Join the discussion @ p2p.wrox.com

Wrox Programmer to Programmer™

Mac OS X and iOS Internals

To the Apple's Core

Jonathan Levin

MAC OS® X AND iOS INTERNALS

INTRODUCTION xxv

▶ PART I FOR POWER USERS

CHAPTER 1 Darwinism: The Evolution of OS X 3

CHAPTER 2 E Pluribus Unum: Architecture of OS X and iOS 17

CHAPTER 3 On the Shoulders of Giants: OS X and iOS Technologies 55

CHAPTER 4 Parts of the Process: Mach-O, Process, and Thread Internals 91

CHAPTER 5 Non Sequitur: Process Tracing and Debugging 147

CHAPTER 6 Alone in the Dark: The Boot Process: EFI and iBoot 183

CHAPTER 7 The Alpha and the Omega — launchd 227

▶ PART II THE KERNEL

CHAPTER 8 Some Assembly Required: Kernel Architectures 261

CHAPTER 9 From the Cradle to the Grave — Kernel Boot and Panics 299

CHAPTER 10 The Medium Is the Message: Mach Primitives 343

CHAPTER 11 Tempus Fugit — Mach Scheduling 389

CHAPTER 12 Commit to Memory: Mach Virtual Memory 447

CHAPTER 13 BS”D — The BSD Layer 501

CHAPTER 14 Something Old, Something New: Advanced BSD Aspects 539

CHAPTER 15 Fee, FI-FO, File: File Systems and the VFS 565

CHAPTER 16 To B (-Tree) or Not to Be — The HFS+ File Systems 607

CHAPTER 17 Adhere to Protocol: The Networking Stack 649

CHAPTER 18 Modu(lu)s Operandi — Kernel Extensions 711

CHAPTER 19 Driving Force — I/O Kit 737

APPENDIX Welcome to the Machine 773

INDEX 793

Mac OS® X and iOS Internals

TO THE APPLE'S CORE

Jonathan Levin

WILEY

John Wiley & Sons, Inc.

Mac OS® X and iOS Internal

Published by
John Wiley & Sons, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256
www.wiley.com

Published by John Wiley & Sons, Inc., Indianapolis, Indiana

Published simultaneously in Canada

ISBN: 978-1-11805765-0
ISBN: 978-1-11822225-6 (ebk)
ISBN: 978-1-11823605-5 (ebk)
ISBN: 978-1-11826094-4 (ebk)

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

Library of Congress Control Number: 2011945020

To Steven Paul Jobs: From Mac OS's very first incarnation, to the present one, wherein the legacy of NeXTSTEP still lives, his relationship with Apple is forever entrenched in OS X (and iOS). People focus on his effect on Apple as a company. No less of an effect, though hidden to the naked eye, is on its architecture. I resisted the pixie dust for 25 years, but he finally made me love Mac OS... Just as soon as I got my shell prompt.

— Jonathan Levin

CREDITS

ACQUISITIONS EDITOR
Mary James

SENIOR PROJECT EDITOR
Adaobi Obi Tulton

DEVELOPMENT EDITOR
Sydney Argenta

TECHNICAL EDITORS
Arie Haenel
Dwight Spivey

PRODUCTION EDITOR
Christine Mugnolo

COPY EDITORS
Paula Lowell
Nancy Rapoport

EDITORIAL MANAGER
Mary Beth Wakefield

FREELANCER EDITORIAL MANAGER
Rosemarie Graham

ASSOCIATE DIRECTOR OF MARKETING
David Mayhew

MARKETING MANAGER
Ashley Zurcher

BUSINESS MANAGER
Amy Knies

PRODUCTION MANAGER
Tim Tate

VICE PRESIDENT AND EXECUTIVE GROUP PUBLISHER
Richard Swadley

VICE PRESIDENT AND EXECUTIVE PUBLISHER
Neil Edde

ASSOCIATE PUBLISHER
Jim Minatel

PROJECT COORDINATOR, COVER
Katie Crocker

PROOFREADER
James Saturnio, Word One New York

INDEXER
Robert Swanson

COVER DESIGNER
Ryan Sneed

COVER IMAGE
© Matt Jeacock / iStockPhoto

ABOUT THE AUTHOR

JONATHAN LEVIN is a seasoned technical trainer and consultant focusing on the internals of the "Big Three" (Windows, Linux, and Mac OS) as well as their mobile derivatives (Android and iOS). Jonathan has been spreading the gospel of kernel engineering and hacking for 15 years, and has given technical talks at DefCON as well as other technical conferences. He is the founder and CTO of Technologeeks.com, a partnership of expert like-minded individuals, devoted to propagating knowledge through technical training, and solving tough technical challenges through consulting. Their areas of expertise cover real-time and other critical aspects of software architectures, system/kernel-level programming, debugging, reverse engineering, and performance optimizations.

ABOUT THE TECHNICAL EDITORS

ARIE HAENEL is a security and internals expert at NDS Ltd. (now part of Cisco). Mr. Haenel has vast experience in data and device security across the board. He holds a Bachelor of Science Engineering in Computer Science from the Jerusalem College of Technology, Israel and an MBA from the University of Poitiers, France. His hobbies include learning Talmud, judo, and solving riddles. He lives in Jerusalem, Israel.

DWIGHT SPIVEY is the author of several Mac books, including *OS X Mountain Lion Portable Genius* and *OS X Lion Portable Genius*. He is also a product manager for Konica Minolta, where he has specialized in working with Mac operating systems, applications, and hardware, as well as color and monochrome laser printers. He teaches classes on Mac usage, writes training and support materials for Konica Minolta, and is a member of the Apple Developer Program. Dwight lives on the Gulf Coast of Alabama with his beautiful wife Cindy and their four amazing children, Victoria, Devyn, Emi, and Reid. He studies theology, draws comic strips, and roots for the Auburn Tigers ("War Eagle!") in his ever-decreasing spare time.

ACKNOWLEDGMENTS

"Y'KNOW, JOHNNY," said my friend Yoav, taking a puff from his cigarette on a warm summer night in Shanghai, "Why don't *you* write a book?"

And that's how it started. It was Yoav (Yobo) Chernitz who planted the seed to write my own book, for a change, after years of reading others'. From that moment, in the Far, Middle, and US East (and the countless flights in between), the idea began to germinate, and this book took form. I had little idea it would turn into the magnum opus it has become, at times taking on a life of its own, and becoming quite the endeavor. With so many unforeseen complications and delays, it's hard to believe it is now done. I tried to illuminate the darkest reaches of this monumental edifice, to delineate them, and leave no stone unturned. Whether or not I have succeeded, you be the judge. But know, I couldn't have done it without the following people:

> Arie Haenel, my longtime friend — a natural born hacker, and no small genius. Always among my harshest critics, and an obvious choice for a technical reviewer.
>
> Moshe Kravchik — whose insights and challenging questions as the book's first reader hopefully made it a lot more readable for all those who follow.
>
> Yuval Navon — from down under in Melbourne, Australia, who has shown me that friendship knows no geographical bounds.

And last, but hardly least, to my darling Amy, who was patient enough to endure my all-too-frequent travels, more than understanding enough to support me to no end, and infinitely wise enough to constantly remind me not only of the important deadlines and obligations. I had with this book, but of the things that are truly the most important in life.

— Jonathan Levin

CONTENTS

INTRODUCTION *xxv*

PART I: FOR POWER USERS

CHAPTER 1: DARWINISM: THE EVOLUTION OF OS X 3

The Pre-Darwin Era: Mac OS Classic 3
The Prodigal Son: NeXTSTEP 4
Enter: OS X 4
OS X Versions, to Date 5
10.0 — Cheetah and the First Foray 5
10.1 — Puma — a Stronger Feline, but . . . 6
10.2 — Jaguar — Getting Better 6
10.3 — Panther and Safari 6
10.4 — Tiger and Intel Transition 6
10.5 — Leopard and UNIX 7
10.6 — Snow Leopard 7
10.7 — Lion 8
10.8 — Mountain Lion 9
iOS — OS X Goes Mobile 10
1.*x* — Heavenly and the First iPhone 11
2.*x* — App Store, 3G and Corporate Features 11
3.*x* — Farewell, 1st gen, Hello iPad 11
4.*x* — iPhone 4, Apple TV, and the iPad 2 11
5.*x* — To the iPhone 4S and Beyond 12
iOS vs. OS X 12
The Future of OS X 15
Summary 16
References 16

CHAPTER 2: E PLURIBUS UNUM: ARCHITECTURE OF OS X AND IOS 17

OS X Architectural Overview 17
The User Experience Layer 19
Aqua 19
Quicklook 20
Spotlight 21

Darwin — The UNIX Core **22**
The Shell 22
The File System 23
UNIX System Directories **24**
OS X–Specific Directories 25
iOS File System Idiosyncrasies 25
Interlude: Bundles **26**
Applications and Apps **26**
Info.plist 28
Resources 30
NIB Files 30
Internationalization with .lproj Files 31
Icons (.icns) 31
CodeResources 31
Frameworks **34**
Framework Bundle Format 34
List of OS X and iOS Public Frameworks 37
Libraries **44**
Other Application Types **46**
System Calls **48**
POSIX 48
Mach System Calls 48
A High-Level View of XNU **51**
Mach 51
The BSD Layer 51
libkern 52
I/O Kit 52
Summary **52**
References **53**

CHAPTER 3: ON THE SHOULDERS OF GIANTS: OS X AND IOS TECHNOLOGIES **55**

BSD Heirlooms **55**
sysctl 56
kqueues 57
Auditing (OS X) 59
Mandatory Access Control 62
OS X- and iOS-Specific Technologies **65**
User and Group Management (OS X) 65
System Configuration 67

Logging 69
Apple Events and AppleScript 72
FSEvents 74
Notifications 78
Additional APIs of interest 79
OS X and iOS Security Mechanisms 79
Code Signing 80
Compartmentalization (Sandboxing) 81
Entitlements: Making the Sandbox Tighter Still 83
Enforcing the Sandbox 89
Summary 90
References 90

CHAPTER 4: PARTS OF THE PROCESS: MACH-O, PROCESS, AND THREAD INTERNALS 91

A Nomenclature Refresher 91
Processes and Threads 91
The Process Lifecycle 92
UNIX Signals 95
Executables 98
Universal Binaries 99
Mach-O Binaries 102
Load Commands 106
Dynamic Libraries 111
Launch-Time Loading of Libraries 111
Runtime Loading of Libraries 122
dyld Features 124
Process Address Space 130
The Process Entry Point 130
Address Space Layout Randomization 131
32-Bit (Intel) 132
64-Bit 132
32-Bit (iOS) 133
Experiment: Using vmmap(1) to Peek Inside a Process's Address Space 135
Process Memory Allocation (User Mode) 138
Heap Allocations 139
Virtual Memory — The sysadmin Perspective 140
Threads 143
Unraveling Threads 143
References 146

CHAPTER 5: NON SEQUITUR: PROCESS TRACING AND DEBUGGING 147

DTrace 147
The D Language 147
dtruss 150
How DTrace Works 152
Other Profiling mechanisms 154
The Decline and Fall of CHUD 154
AppleProfileFamily: The Heir Apparent 155
Process Information 156
sysctl 156
proc_info 156
Process and System Snapshots 159
system_profiler(8) 159
sysdiagnose(1) 159
allmemory(1) 160
stackshot(1) 160
The stack_snapshot System Call 162
kdebug 165
kdebug-based Utilities 165
kdebug codes 166
Writing kdebug messages 168
Reading kdebug messages 169
Application Crashes 170
Application Hangs and Sampling 173
Memory Corruption Bugs 174
Memory Leaks 176
heap(1) 177
leaks(1) 177
malloc_history(1) 178
Standard UNIX Tools 178
Process listing with ps(1) 179
System-Wide View with top(1) 179
File Diagnostics with lsof(1) and fuser(1) 180
Using GDB 181
GDB Darwin Extensions 181
GDB on iOS 182
LLDB 182
Summary 182
References and Further Reading 182

CHAPTER 6: ALONE IN THE DARK: THE BOOT PROCESS: EFI AND IBOOT 183

Traditional Forms of Boot 183
EFI Demystified 185
Basic Concepts of EFI 186
The EFI Services 188
NVRAM Variables 192
OS X and boot.efi 194
Flow of boot.efi 195
Booting the Kernel 201
Kernel Callbacks into EFI 203
Boot.efi Changes in Lion 204
Boot Camp 204
Count Your Blessings 204
Experiment: Running EFI Programs on a Mac 206
iOS and iBoot 210
Precursor: The Boot ROM 210
Normal Boot 211
Recovery Mode 212
Device Firmware Update (DFU) Mode 213
Downgrade and Replay Attacks 213
Installation Images 214
OS X Installation Process 214
iOS File System Images (.ipsw) 219
Summary 225
References and Further Reading 225

CHAPTER 7: THE ALPHA AND THE OMEGA — LAUNCHD 227

launchd 227
Starting launchd 227
System-Wide Versus Per-User launchd 228
Daemons and Agents 229
The Many Faces of launchd 229
Lists of LaunchDaemons 241
GUI Shells 246
Finder (OS X) 247
SpringBoard (iOS) 248
XPC (Lion and iOS) 253
Summary 257
References and Further Reading 258

PART II: THE KERNEL

CHAPTER 8: SOME ASSEMBLY REQUIRED: KERNEL ARCHITECTURES 261

Kernel Basics 261
Kernel Architectures 262
User Mode versus Kernel Mode 266
Intel Architecture — Rings 266
ARM Architecture: CPSR 267
Kernel/User Transition Mechanisms 268
Trap Handlers on Intel 269
Voluntary kernel transition 278
System Call Processing 283
POSIX/BSD System calls 284
Mach Traps 287
Machine Dependent Calls 292
Diagnostic calls 292
XNU and hardware abstraction 295
Summary 297
References 297

CHAPTER 9: FROM THE CRADLE TO THE GRAVE — KERNEL BOOT AND PANICS 299

The XNU Sources 299
Getting the Sources 299
Making XNU 300
One Kernel, Multiple Architectures 302
The XNU Source Tree 305
Booting XNU 308
The Bird's Eye View 309
OS X: vstart 310
iOS: start 310
[i386|arm]_init 311
i386_init_slave() 313
machine_startup 314
kernel_bootstrap 314
kernel_bootstrap_thread 318
bsd_init 320
bsdinit_task 325
Sleeping and Waking Up 328
Boot Arguments 329

Kernel Debugging **332**
"Don't Panic" 333
Implementation of Panic 334
Panic Reports 336
Summary **340**
References **341**

CHAPTER 10: THE MEDIUM IS THE MESSAGE: MACH PRIMITIVES 343

Introducing: Mach **344**
The Mach Design Philosophy 344
Mach Design Goals 345
Mach Messages **346**
Simple Messages 346
Complex messages 347
Sending Messages 348
Ports 349
The Mach Interface Generator (MIG) 351
IPC, in Depth **357**
Behind the Scenes of Message Passing 359
Synchronization Primitives **360**
Lock Group Objects 361
Mutex Object 362
Read-Write Lock Object 363
Spinlock Object 364
Semaphore Object 364
Lock Set Object 366
Machine Primitives **367**
Clock Object 378
Processor Object 380
Processor Set Object 384
Summary **388**
References **388**

CHAPTER 11: TEMPUS FUGIT — MACH SCHEDULING 389

Scheduling Primitives **389**
Threads 390
Tasks 395
Task and Thread APIs 399
Task APIs 399
Thread APIs 404

Scheduling **408**
The High-Level View 408
Priorities 409
Run Queues 412
Mach Scheduler Specifics **415**
Asynchronous Software Traps (ASTs) 423
Scheduling Algorithms 427
Timer Interrupts **431**
Interrupt-Driven Scheduling 431
Timer Interrupt Processing in XNU 432
Exceptions **436**
The Mach Exception Model 436
Implementation Details 437
Experiment: Mach Exception Handling 440
Summary **446**
References **446**

CHAPTER 12: COMMIT TO MEMORY: MACH VIRTUAL MEMORY **447**

Virtual Memory Architecture **447**
The 30,000-Foot View of Virtual Memory 448
The Bird's Eye View 449
The User Mode View 452
Physical Memory Management **462**
Mach Zones **467**
The Mach Zone Structure 468
Zone Setup During Boot 470
Zone Garbage Collection 471
Zone Debugging 473
Kernel Memory Allocators **473**
kernel_memory_allocate() 473
kmem_alloc() and Friends 477
kalloc 477
OSMalloc 479
Mach Pagers **480**
The Mach Pager interface 480
Universal Page Lists 484
Pager Types 486
Paging Policy Management **494**
The Pageout Daemon 495
Handling Page Faults 497
The dynamic_pager(8) (OS X) 498

Summary 499
References 500

CHAPTER 13: BS"D — THE BSD LAYER 501

Introducing BSD 501
One Ring to Bind Them 502
What's in the POSIX Standard? 503
Implementing BSD 503
XNU Is *Not* Fully BSD 504
Processes and Threads 504
BSD Process Structs 504
Process Lists and Groups 507
Threads 508
Mapping to Mach 510
Process Creation 512
The User Mode Perspective 512
The Kernel Mode Perspective 513
Loading and Executing Binaries 516
Mach-O Binaries 522
Process Control and Tracing 525
ptrace (#26) 525
proc_info (#336) 527
Policies 527
Process Suspension/Resumption 529
Signals 529
The UNIX Exception Handler 529
Hardware-Generated Signals 534
Software-Generated Signals 535
Signal Handling by the Victim 536
Summary 536
References 537

CHAPTER 14: SOMETHING OLD, SOMETHING NEW: ADVANCED BSD ASPECTS 539

Memory Management 539
POSIX Memory and Page Management System Calls 540
BSD Internal Memory Functions 541
Memory Pressure 545
Jetsam (iOS) 546
Kernel Address Space Layout Randomization 548
Work Queues 550

BSD Heirlooms Revisited 552
Sysctl 552
Kqueues 555
Auditing (OS X) 556
Mandatory Access Control 558
Apple's Policy Modules 560
Summary 563
References 563

CHAPTER 15: FEE, FI-FO, FILE: FILE SYSTEMS AND THE VFS 565

Prelude: Disk Devices and Partitions 565
Partitioning Schemes 567
Generic File System Concepts 577
Files 577
Extended Attributes 577
Permissions 577
Timestamps 578
Shortcuts and Links 578
File Systems in the Apple Ecosystem 579
Native Apple File Systems 579
DOS/Windows File Systems 580
CD/DVD File Systems 581
Network-Based File Systems 582
Pseudo File Systems 583
Mounting File Systems (OS X only) 587
Disk Image Files 589
Booting from a Disk Image (Lion) 590
The Virtual File System Switch 591
The File System Entry 591
The Mount Entry 592
The vnode Object 595
FUSE — File Systems in USEr Space 597
File I/O from Processes 600
Summary 605
References and Further Reading 605

CHAPTER 16: TO B (-TREE) OR NOT TO BE — THE HFS+ FILE SYSTEMS 607

HFS+ File System Concepts 607
Timestamps 607
Access Control Lists 608

Extended Attributes 608
Forks 611
Compression 612
Unicode Support 617
Finder integration 617
Case Sensitivity (HFSX) 619
Journaling 619
Dynamic Resizing 620
Metadata Zone 620
Hot Files 621
Dynamic Defragmentation 622
HFS+ Design Concepts 624
B-Trees: The Basics 624
Components 630
The HFS+ Volume Header 631
The Catalog File 633
The Extent Overflow 640
The Attribute B-Tree 640
The Hot File B-Tree 641
The Allocation File 642
HFS Journaling 642
VFS and Kernel Integration 645
fsctl(2) integration 645
sysctl(2) integration 646
File System Status Notifications 647
Summary 647
References 648

CHAPTER 17: ADHERE TO PROTOCOL: THE NETWORKING STACK 649

User Mode Revisited 650
UNIX Domain Sockets 651
IPv4 Networking 651
Routing Sockets 652
Network Driver Sockets 652
IPSec Key Management Sockets 654
IPv6 Networking 654
System Sockets 655
Socket and Protocol Statistics 658
Layer V: Sockets 660
Socket Descriptors 660
mbufs 661
Sockets in Kernel Mode 667

Layer IV: Transport Protocols 668
Domains and Protosws 669
Initializing Domains 673
Layer III: Network Protocols 676
Layer II: Interfaces 678
Interfaces in OS X and iOS 678
The Data Link Interface Layer 680
The ifnet Structure 680
Case Study: utun 682
Putting It All Together: The Stack 686
Receiving Data 686
Sending Data 690
Packet Filtering 693
Socket Filters 694
ipfw(8) 696
The PF Packet Filter (Lion and iOS) 697
IP Filters 698
Interface Filters 701
The Berkeley Packet Filter 701
Traffic Shaping and QoS 705
The Integrated Services Model 706
The Differentiated Services Model 706
Implementing dummynet 706
Controlling Parameters from User Mode 707
Summary 707
References and Further Reading 708

CHAPTER 18: MODU(LU)S OPERANDI — KERNEL EXTENSIONS 711

Extending the Kernel 711
Securing Modular Architecture 712
Kernel Extensions (Kexts) 713
Kext Structure 717
Kext Security Requirements 718
Working with Kernel Extensions 719
Kernelcaches 719
Multi-Kexts 723
A Programmer's View of Kexts 724
Kernel Kext Support 725
Summary 735
References 735

CHAPTER 19: DRIVING FORCE — I/O KIT 737

Introducing I/O Kit 738
Device Driver Programming Constraints 738
What I/O Kit Is 738
What I/O Kit Isn't 741
LibKern: The I/O Kit Base Classes 742
The I/O Registry 743
I/O Kit from User Mode 746
I/O Registry Access 747
Getting/Setting Driver Properties 749
Plug and Play (Notification Ports) 750
I/O Kit Power Management 751
Other I/O Kit Subsystems 753
I/O Kit Diagnostics 753
I/O Kit Kernel Drivers 755
Driver Matching 755
The I/O Kit Families 757
The I/O Kit Driver Model 761
The IOWorkLoop 764
Interrupt Handling 765
I/O Kit Memory Management 769
BSD Integration 769
Summary 771
References and Further Reading 771

APPENDIX: WELCOME TO THE MACHINE 773

INDEX *793*

INTRODUCTION

EVEN MORE THAN TEN YEARS AFTER ITS INCEPTION, there is a dearth of books discussing the architecture of OS X, and virtually none about iOS. While there is plentiful documentation on Objective-C, the frameworks, and Cocoa APIs of OS X, it often stops short of the system-call level and implementation specifics. There is some documentation on the kernel (mostly by Apple), but it, too, focuses on building drivers (with I/O Kit), and shows only the more elegant parts, and virtually nothing on the Mach core that is foundation of XNU. XNU is open source, granted, but with over a million lines of source (and comments) with some dating as far back to 1987, it's not exactly a fun read.

This is not the case with other operating systems. Linux, being fully open source, has no shortage of books, including the excellent series by O'Reilly. Windows, though closed, is exceptionally well documented by Microsoft (and its source has been "liberated" on more than one occasion). This book aims to do for XNU what Bovet & Cesati's *Understanding the Linux Kernel* does for Linux, and Russinovich's *Windows Internals* does for Windows. Both are superb books, clearly explaining the architectures of these incredibly complex operating systems. With any luck, the book you are holding (or downloaded as a PDF) will do the same to expound on the inner workings of Apple's operating systems.

A previous book on Mac OS — Amit Singh's excellent *OS X Internals*: *A Systems Approach* is an amazing reference, and provides a vast wealth of valuable information. Unfortunately, it is PowerPC oriented, and is only updated up until Tiger, circa 2006. Since then, some six years have passed. Six long years, in which OS X has abandoned PowerPC, has been fully ported to Intel, and has progressed by almost four versions. Through Leopard, Snow Leopard, Lion and, most recently Mountain Lion, the wild cat family is expanding, and many more features have been added. Additionally, OS X has been ported anew. This time to the ARM architecture, as iOS, (which is, by some counts, the world's leading operating system in the mobile environments). This book, therefore, aims to pick up where its predecessor left off, and discuss the new felines in the Apple ecosystem, as well as the various iOS versions.

Apple's operating systems have proven to be moving targets. This book was originally written to target iOS 5 and Lion, but both have gone on evolving. iOS is, at the time this book goes to print, at 5.1.1 with hints of iOS 6. OS X is still at Lion (10.7.4), but Mountain Lion (10.8) is in advanced developer previews, and this book will hit the shelves coinciding with its release. Every attempt has been made to keep the information as updated as possible to reflect all the versions, and remain relevant going forward.

OVERVIEW AND READING SUGGESTION

This is a pretty large book. Initially, it was not designed to be this big and detailed, but the more I delved into OS X I uncovered more of the abstruse, for which I could find no detailed explanation or documentation. I therefore found myself writing about more and more aspects. An operating system is a full eco-system with its own geography (hardware), atmosphere (virtual memory), flora and fauna (processes). This book tries to methodically document as much as it can, while not sacrificing clarity for detail (or vice versa). No mere feat.

Architecture at a Glance

OS X and iOS are have a complex architecture, which is a hybrid of several very different technologies: The UI and APIs of the legacy OS 9 (for OS X) with NextSTEP's Cocoa, the system calls and kernel layer of BSD, and the kernel structure of NeXTSTEP. Though an amalgam, it still maintains a relatively clean separation between its components. Figure I-1 shows a bird's eye view of the architecture, and maps the components to the corresponding chapters in this book.

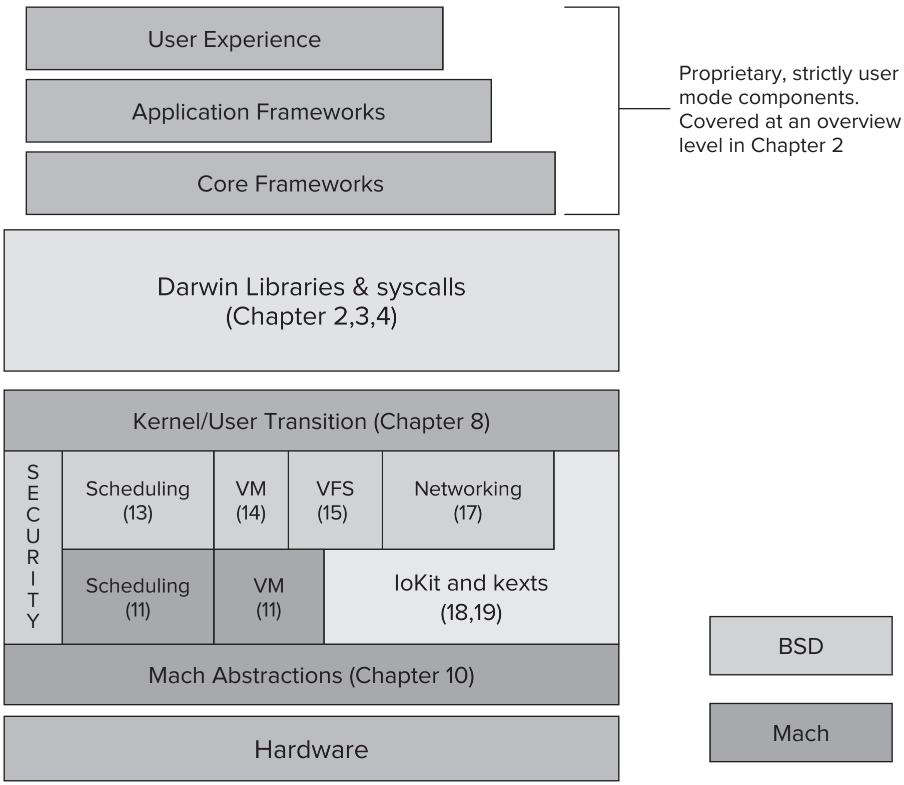


FIGURE I-1: OS X Architecture, and its mapping to chapters in this book

This book additionally contains chapters on non-architectural, yet very important topics, such as debugging (5), firmware (6) and user mode startup (7), kernel-mode startup (9), and kernel modules (18). Lastly, there are two appendices: The first, providing a quick reference for POSIX system calls and Mach traps, and the second, providing a gentle high-level introduction to the assembly of both Intel and ARM architectures.

Target Audience

There are generally four types of people who might find this tome, or its parts, interesting:

- Power users and system administrators who want to get a better idea of how OS X works. Mac OS adoption grows steadily by the day, as market claws back market share that was, for years, denied by the utter hegemony of the PC. Macs are steadily growing more popular in corporate environments, and overshadowing PCs in academia.

- User mode developers who find the vast playground of Objective-C insufficient, and want to see how their programs are really executed at the system level.
- Kernel mode developers who revel in the vast potential of kernel-mode low-level programming of drivers, kernel enhancements, or file system and network hooks.
- Hackers and jailbreakers who aren't satisfied with jailbreaking with a ready-made tool, exploit or patch, and want to understand how and what exactly is being patched, and how the system can be further tweaked and bent to their will. Note, that in this context, the target audience refers to people who delve deeper into internals for the fun, excitement, and challenge, and not for any illicit or evil purposes.

Choose your own adventure

While this book can be read cover to cover, let's not forget it is a technical book, after all. The chapters are therefore designed to be read individually, as a detailed explanation or as a quick reference. You have the option of reading chapters in sequential or random access, skimming or even skipping over some chapters, and coming back to them later for a more thorough read. If a chapter refers to a concept or function discussed in a previous chapter, it is clearly noted.

You are also welcome to employ a reading strategy which reflects the type of target reader you classify yourself as. For example, the chapters of the first part of this book can therefore be broken into the flow shown in Figure I-2:

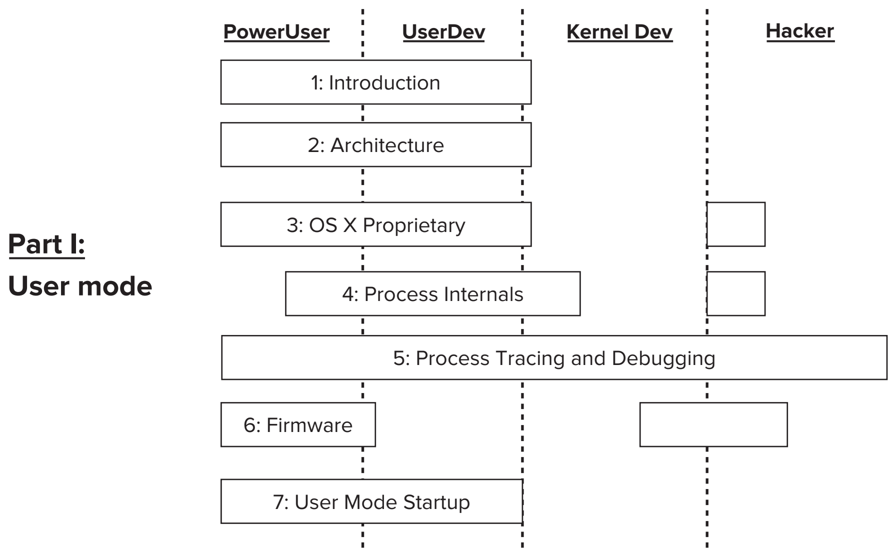


FIGURE I-2: Reading suggestion for the first part of this book, which focuses on user mode architecture

In Figure I-2, a full bar implies the chapter contents are of interest to the target reader, and a partial bar implies at least some interest. Naturally, every reader's interest will vary. This is why every chapter starts with a brief introduction, discussing what the chapter is about. Likewise, just by looking at the section headers in the table of contents you can figure out if the section merits a read or just a quick skim.

The second part of this book could actually have been a volume by itself. It focuses on the XNU kernel architecture, and is considerably more complicated than the first. This cannot be avoided; by their very nature, kernels are subject to a more complicated, real-time, and hardware constrained environment. This part shows many more code listings, and (thankfully, rarely) even has to go into snippets of code implemented in assembly. Reading suggestions for this part of the book are shown in Figure I-3.

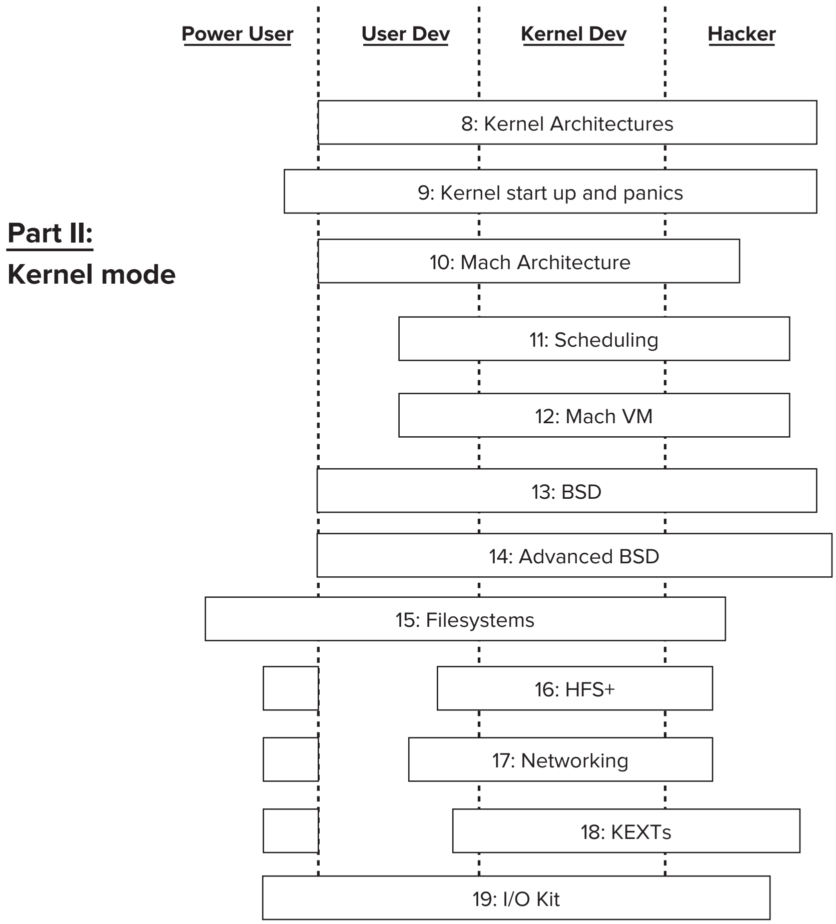


FIGURE I-3: Reading suggestion for the second part of this book, which focuses on the kernel

EXPERIMENTS

Most chapters in this book contain "experiments," which usually involve running a few shell commands, and sometimes custom sample programs. They are classified as "experiments" because they demonstrate aspects of the operating system which can vary, depending on OS version, or on configuration. Normally, the results of these experiments are demonstrated in detail, but you are more than encouraged to try the experiments on your own system, and witness the results. Like UNIX, which it implements, Mac OS X can truly be experienced and absorbed through the fingers, not the eyes or ears.

In some cases, some parts of the experiments have been left out as an exercise for the reader. Even though the book's companion website will have the solutions — i.e. fully working versions of the exercises in question — you are encouraged to try to complete those parts yourself. Careful reading of the book, with a modicum of common sense, should provide you with everything you need to do so.

TOOLS

The book also makes use of a few tools, which were developed by the author to accompany the book. The tools, true to the UNIX heritage, are command line tools, and are meant to be both easily readable as well as `grep(1)`-able, making them useful not just for manual usage, but also in scripts.

filemon

Chapter 3 presents a tool called "filemon," to display real time file system activity on OS X and iOS. An homage to Russinovich's tool of the same name, this simple utility relies on the FSEvents device, present in OS X and iOS 5, to follow file system related events, such as creation and deletion of files.

psx

Chapter 4 presents a tool called psx, an extended ps-like command which can display pretty much any tidbit of information one could possibly require about processes and threads in OS X. It is particularly useful for this chapter, which deals with process internals, and demonstrates using an undocumented system call, `proc_info`. The tool requires no special permissions if you are viewing your own processes, but will require root permissions otherwise. The tool can be freely downloaded from the book's companion website, with full source code.

jtool

While for most binary function one can use the OS X built-in `otool(1)`, it leaves much to be desired in analyzing data section and can get confused when displaying ARM binaries due to the two modes of assembly in the ARM architecture. jtool aims to improve on `otool`, by addressing these shortcomings, and offering useful new features for static binary analysis. The tool comes in handy in Chapter 4, which details the Mach-O file format, as well as later in this book, due to its many useful features, like finding references in files and limited disassembly skills. The tool can be freely downloaded from the book's companion website, but is closed source.

dEFI

This is a simple program to dump the firmware (EFI) variables on an Intel Mac and to display registered EFI providers. This tool demonstrates the basics of EFI programming — interfacing with the boot and runtime services. This tool can be freely downloaded, along with its source code. It is presented in Chapter 6.

joker

The joker tool, presented in Chapter 8, is a simple tool created to play with the kernel (specifically, in iOS). The tool can find and display the system call and Mach trap tables of iOS and OS X kernels, show sysctl structures, and look for particular patterns in the binary. This tool is highly useful for reverse engineers and hackers alike, as the trap and system call symbols are no longer exported.

corerupt

Chapter 11 discusses the low-level APIs of the Mach virtual memory manager. To demonstrate just how powerful (and dangerous) these APIs are, the book provides the corerupt tool. This tool enables you to dump any process's virtual memory map to a file in a core-compatible format, similar to Windows' Create Dump File option, and much like the gcore tool in this book's predecessor. It further improves on its precursor, by providing support for ARM and allowing invasive operations on the vm map, such as modifying its pages.

HFSleuth

A key tool used in the book is HFSleuth, a command line all-in-one utility for viewing the supporting structures of HFS+ file systems, which are the native OS X file system type. The tool was developed because there really are no alternative ways to demonstrate the inner workings of this rather complicated file system. Singh's book, *Mac Os X Internals*: *A Systems Approach* (Addison-Wesley; 2006) also included a similar, though less feature-ful tool called hfsdebug, but the tool was only provided for PowerPC, and was discontinued in favor of a commercial tool, fileXRay.

To use HFSleuth on an actual file system, you must be able to read the file system. One option is to simply be root. HFSleuth's functions are nearly all read-only, so rest assured it is perfectly safe. But access permissions to the underlying block (and sometimes, character) devices on which the file systems are usually `rw-r-----`, meaning the devices are not readable by plebes. If you generally distrust root and adhere to least privilege (a wise choice!), an equally potent alternative is to `chmod(1)` the permissions on the HFS+ partition devices, making them readable to your user (usually, this involves an `o+r`). Advanced functions (such as repair, or HFS+/HFSX conversion) will require write access.

HFSleuth can be freely downloaded from the book's companion website and will remain freely available, period. Like its predecessor, however, it is not open source.

lsock

The much needed functionality of `netstat -o`, which shows the processes owning the various sockets in the system, is missing from OS X. It exists in `lsof(1)`, but the latter makes it somewhat cumbersome to weed out sockets from other open files. Another functionality missing is the ability to display socket connections as they are created, much like Windows' TCPMon. This tool, introduced in Chapter 17, uses an undocumented kernel control protocol called `com.apple.network.statistics` to obtain real-time notifications of sockets as they are created. The tool is especially easy to incorporate into scripts, making it handy for use as a connection event handler.

jkextstat

The last tool used in the book is jkextstat, a `kextstat(8)`-compatible utility to list kernel extensions. Unlike the original, it supports verbose mode, and can work on iOS. This makes it invaluable in exploring the iOS kernel hands-on, something which — until this book — was very difficult, as the binary kextstat for iOS uses APIs which are no longer supported. The tool improves on its original inspiration by allowing more detailed output, focusing on particular kernel extensions, as well as output to XML format.

> *All the tools mentioned here are made available for free, and will remain free, whether you buy (or copy) the book. This is because they are generally useful, and fill many advanced functions, which are either lacking, or present but well hidden, in Apple's own tools.*

CONVENTIONS USED IN THIS BOOK

To make it easier to follow along the book and not be bogged down by reiterating specific background for example code and programs, this book adopts a few conventions, which are meant to subtly remind you of the context of the given listings.

Dramatis Personae

The demos and listings in this book have naturally been produced and tested on various versions of Apple computers and i-Devices. As is in the habit of sysadmins to name their boxes, each host has his or her own "personality" and name. Rather than repeatedly specifying which demo is based on which device and OS, the shell command prompt has been left as is, and by the hostname you can easily figure out which version of OS X or iOS the demo can be reproduced on. (See Table I-1.)

TABLE I-1: Host Name and Version Information for the Book's Demos

HOST NAME	TYPE	OS VERSION	USED FOR
Ergo	MacBook Air, 2010	Snow Leopard , 10.6.8	Generic OS X feature demonstration. Tested in Snow Leopard and later
iPhonoclast	iPhone 4S	iOS 5.1.1	iOS 5 and later features on an A5 (ARM multi-core)
Minion	Mac Mini, 2010	Lion, 10.7.4	Lion specific feature demonstration
Simulacrum	VMWare image	Mountain Lion, 10.8.0 DP3	Mountain Lion (Developer Preview) specific feature demonstration
Padishah	iPad 2	iOS 4.3.3	iOS 4 and later features
Podicum	iPod Touch, 4G	iOS 5.0.1	iOS 5 specific features, on A4 or A5

Further, shell prompts of `root@` demonstrate a command runnable only by the root user. This makes it easy to see which examples will run on which system, with what privileges.

Code Excerpts and Samples

This book contains a considerable number of code samples of two types:

- **Example programs**, which are found mostly in the first part. These usually demonstrate simple concepts and principles that hold in user mode, or specific APIs or libraries. The example programs were all devised by the author, are well commented, and are free for you to try yourself, modify in any way you see fit, or just leave on the page. In an effort to promote the lazy, all these programs are available on the book's website, in both open source and binary form.
- **Darwin code excerpts**, which are found mostly in the second part. These are almost entirely snippets of XNU's code, taken from the latest open source version, i.e. 1699.26.8 (corresponding to Lion 10.7.4). All code is open source, but subject to Apple's Public Source License. The excerpts are provided here for demonstration of the relevant parts in XNU's architecture. While natural language is potentially prone to some ambiguities, code is context free and precise (though unfortunately sometimes less readable), and so at times the most precise explanation comes from reading the code. When code references are provided, they are usually either to the header files (denoted by the standard C < > notation, e.g. `<mach/mach-o.h>`) in `/usr/include`. Other times, they may refer to the Darwin sources, either of XNU or some related package. In those cases, the relative path is used (e.g. `osfmk/kern/spl.c`, relating to where the XNU kernel source is extracted). The related package will always be specified in the section, and in Part II of the book nearly all references are to the XNU kernel source.

XNU and Darwin components are fairly well documented, but this book tries to go the extra step, and sometimes provide additional explanations inline, as comments. To be clear, such annotations, which are not part of the original source code, can be clearly marked by their C++ style comment, rather than the C style comment which is typical in Darwin as in this sample listing:

LISTING I-1: SAMPLE LISTING

```
/* This is a Darwin comment, as it appears in the original source */


// This is an annotation provided by the author, elaborating or explaining
// something which the documentation may or may not leave wanting

// Where the source code is long and tedious, or just obvious, some parts may
// be omitted, and this is denoted by a comment marking ellipsis (...), i.e:

// ...

     important parts of a listing or output may be shown in bold
```

The book distinguishes between *outputs* and *listings*. Listings are verbatim references from files, either program source code or system files. Outputs, on the other hand, are textual captures of user commands, shown for demonstration on OS X, iOS, or — sometimes — both. The book aims to compare and contrast the two systems, so it is not uncommon to find the same sequence of commands shown on both systems. In an output, you will see the user commands that were typed marked in bold, and are encouraged to follow along and try them on your own systems.

In general, the code listings are provided to elucidate, not to confuse. Natural language is not without its ambiguities, but code can only be interpreted one way (even if sometimes that way is not entirely clear). Whenever possible, clear descriptions aided by detailed figures will hopefully enable you to just skim through the code. Fluency in C (and sometimes a little assembly) is naturally helpful for reading the code samples, but is not necessary. The comments — especially the extra annotations — help you understand the gist of the code. More commonly, block diagrams and flow charts are presented, leaving the functions as black boxes. This enables to choose between remaining at an overview level, or delving deeper and seeing the actual variables and functions of the implementations. Be warned, however, that the complexity of the code, being the product of many people and many coding styles, varies greatly throughout XNU.

In the case of iOS, XNU remains closed. iOS versions actually use a version of XNU many revisions ahead of the publicly released versions. Naturally, code samples cannot be shown, but in some cases disassembly (mostly of iOS 5.*x*) is provided. The assembly in question is ARM, and comments there — all provided by the author — aim to explicate its inner workings. For all things assembly, you can refer to the appendix in this book for a quick overview.

Typographic Conventions

Every effort has been made to ensure that these conventions are followed throughout this book:

- Words in `courier font` denote commands, file names, function names, or variable names from the Darwin sources.
- Commands are further specified by their man section (if applicable) in parentheses. Example: `ls(1)` for a user command, `write(2)` for a system call, `printf(3)` for a library call, and `ipfw(8)` for a system administration command. Most commands and system calls shown in this book are usually well documented in the manual page, and the book does not attempt to upstage the fine manual (i.e. RTFM, first). Occasionally, however, the documentation may leave some aspects wanting — or, rarely, undocumented at all — and this is where further information is provided.

THE COMPANION WEBSITE(S)

Both OS X and iOS have rapidly evolved, and continue to do so. I will try to play catch up, and keep an updated companion website for this book at `http://newosxbook.com`. My company, `(http://technologeeks.com)`, also maintains the OS X and iOS Kernel developers group on LinkedIn (alongside those of Windows and Android), with its website of `http://darwin.kerneldevelopers.com` (the name chosen in a forward-compatible view of a post OS X era. The latter site includes a questions and answers forum, which will hopefully become a bustling arena for OS X and iOS related discussions.

On the book's companion website you can find:

- An appendix that lists the various POSIX and Mach system calls.
- The sample programs included in experiments throughout this book — for the enthusiastic to try, yet lazy to code. The programs are provided in source form, but also as binaries (for those even lazier to compile(!) or devoid of XCode).
- The tools introduced in this book, and discussed in this introduction freely downloadable in binary form for both OS X and iOS, and often times with source.
- Updated references and links to other web resources, as they become available.
- Updated articles about new features or enhancements, as time goes by.
- Errata — *Errare est humanum*, and — especially in iOS, where most of the details were eked out by painful disassembly, there may be inaccuracies or version differences that need to be fixed.

This book has been an unbelievable journey, through the looking glass (while playing with kittens), unraveling the very fabric of the reality presented to user mode applications. I truly hope that you, the reader, will find it as illuminating as I have, drawing ideas not just on OS X and iOS, but on operating system architecture and software design in general.

Read on then, ye devout Apple-lyte, and learn.

PART I
For Power Users

- **CHAPTER 1:** Darwinism: The Evolution of OS X
- **CHAPTER 2:** E Pluribus Unum: Architecture of OS X and iOS
- **CHAPTER 3:** On the Shoulders of Giants: OS X and iOS Technologies
- **CHAPTER 4:** Parts of the Process: Mach-O, Process, and Thread Internals
- **CHAPTER 5:** Non Sequitur: Process Tracing and Debugging
- **CHAPTER 6:** Alone in the Dark: The Boot Process: EFI and iBoot
- **CHAPTER 7:** The Alpha and the Omega — launchd

1

Darwinism: The Evolution of OS X

Mac OS has evolved tremendously since its inception. From a niche operating system of a cult crowd, it has slowly but surely gained mainstream share, with the recent years showing an explosion in popularity as Macbooks, Macbook Pros, and Airs become ever more ubiquitous, clawing back market share from the gradually declining PC. Its mobile derivative — iOS — is by some accounts the mobile operating system with the largest market share, head-to-head with Linux's derivative, Android.

The growth, however, did not happen overnight. In fact, it was a long and excruciating process, which saw Mac OS come close to extinction, before it was reborn as "OS X." Simply "reborn" is an understatement, as Mac OS underwent a total reincarnation, with its architecture torn down and rebuilt anew. Even then, Mac OS still faced significant hardship before the big breakthrough — which came with Apple's transition to Intel-based architecture, leaving behind its long history with PowerPC architectures.

The latest and greatest version, OS X 10.7, or Lion, occurred shortly before the release of this book, as did the release of iOS 5.*x*, the most recent version of iOS. To understand their features and the relationship between the two, however, it makes sense to take a few steps back and understand how the architecture unifying both came to be.

The following is by no means a complete listing of features, but rather a high-level perspective. Apple has been known to add hundreds of features between releases, mostly in GUI and application support frameworks. Rather, more emphasis is placed on design and engineering features. For a comprehensive treatise on Mac OS versions to date, see Amit Singh's work on the subject[1], or check Ars Technica's comprehensive reviews[2]. Wikipedia also maintains a fairly complete list of changes[3].

THE PRE-DARWIN ERA: MAC OS CLASSIC

Mac OS Classic is the name given the pre-OS X era of Mac OS. The operating system then was nothing much to boast about. True, it was novel in that it was an all-GUI system (earlier versions did not have a command line like today's "Terminal" app). Memory management was poor, however, and multitasking was cooperative, which — by today's standards — is considered primitive. Cooperative multitasking involves processes voluntarily yielding their CPU timeslice, and works reasonably well when processes are well behaved. If even one process refuses to cooperate, however, the entire system screeches to a halt. Nonetheless, Mac OS Classic laid some of the foundations for the contemporary Mac OS, or OS X. Primarily, those foundations include the "Finder" GUI, and the file system support for "forks" in the first generation HFS file system. These affect OS X to this very day.

THE PRODIGAL SON: NEXTSTEP

While Mac OS experienced its growing pains in the face of the gargantuan PC, its founder Steve Jobs left Apple (by some accounts was ousted) to get busy with a new and radically different company. The company, NeXT, manufactured specialized hardware, the NeXT computer and NeXTstation, with a dedicated operating system called NeXTSTEP.

NeXTSTEP boasted some avant-garde features for the time:

- NeXTSTEP was based on the Mach microkernel, a little-known kernel developed by Carnegie Mellon University (CMU). The concept of a microkernel was, itself, considered a novelty, and remains rarely implemented even today.
- The development language used was *Objective-C*, a superset of C, which — unlike C++ — is heavily object-oriented.
- The same object-orientation was prevalent all throughout the operating system. The system offered frameworks and kits, which allowed for rapid GUI development using a rich object library, based on the `NSObject`.
- The device driver environment was an object-oriented framework as well, known as DriverKit. Drivers could subclass other drivers, inheriting from them and extending their functionality.
- Applications and libraries were distributed in self-contained *bundles*. Bundles consisted of a fixed directory structure, which was used to package software, along with its dependencies and related files, so installing and uninstalling could be as easy as moving around a folder.
- *PostScript* was heavily used in the system, including a variant called "display postscript," which enabled the rendering of display images as postscript. Printing support was thus 1:1, unlike other operating systems, which needed to convert to a printer-friendly format.

NeXTSTEP went down the road of better operating systems (remember OS/2?), and is nowadays extinct, save for a GNUStep port. Yet, its legacy lives on to the present day. One winter day in 1997, Apple — with an OS that wasn't going anywhere — ended up acquiring NeXT, bringing its intellectual property into Apple, along with Steve Jobs. And the rest, as they say, is history.

ENTER: OS X

As a result of the acquisition of NeXT, Apple gained access to Mach, Objective-C, and the other aspects of the NeXTSTEP architecture. While NeXTSTEP was discontinued as a result, these components live on in OS X. In fact, OS X can be considered as a fusion of Mac OS Classic and

NeXTSTEP, mostly the latter absorbing the former. The transition wasn't immediate, and Mac OS passed through an interim operating system called Rhapsody, which never really went public. It was Rhapsody, however, that eventually evolved into the first version of Mac OS X, and its kernel became the core of what is now known as Darwin.

Mac OS X is closer in its design and implementation to NeXTSTEP than it is to any other operating system, including Apple's own OS 9. As you will see, the core components of OS X — Cocoa, Mach, IOKit, the XCode Interface Builder, and others — are all direct descendants of NeXTSTEP. The fusion of two fringe, niche operating systems — one with a great GUI and poor design, the other with great design but lackluster GUI — resulted in a new OS that has become far more popular than the both of them combined.

OS X VS. DARWIN

There is sometimes some confusion between OS X and Darwin regarding the definitions of the two terms, and the relationship between them. Let's attempt to clarify this:

OS X is the name given, collectively, to the entire operating system. As discussed in the next chapter, the operating system contains many components, of which Darwin is but one.

Darwin is the UNIX-like core of the operating system, which is itself comprised of the kernel, XNU (an acronym meaning "X is Not UNIX", similar to GNU's recursive acronym) and the runtime. Darwin is open source (save for its adaptation to ARM in iOS, discussed later), whereas other parts of OS X — Apple's frameworks — are not.

There exists a straightforward correlation between the version of OS X and the version of Darwin. With the exception of OS X 10.0, which utilized Darwin 1.3. *x*, all other versions follow a simple equation:

```
If (OSX.version == 10.x.y)
 Darwin.version = (4+x).y
```

So, for example, the upcoming Mountain Lion, being 10.8.0, is Darwin 12.0. The last release of Snow Leopard, 10.6.8, is Darwin 10.8. It's a little bit confusing, but at least it's consistent.

OS X VERSIONS, TO DATE

Since its inception, Mac OS X has gone through several versions. From a novel, but — by some accounts — immature operating system, it has transformed into the feature-rich platform that is Lion. The following section offers an overview of the major features, particularly those which involve architectural or kernel mode changes.

10.0 — Cheetah and the First Foray

Mac OS X 10.0, known as Cheetah, is the first public release of the OS X platform. About a year after a public beta, Kodiak, Apple released 10.0 in March 2001. It marks a significant departure from the old-style Mac OSes with the integration of features from NeXT/Openstep, and the layered architecture we will discuss shortly. It is a total rewrite of the MacOS 9, and shares little in common, save for maybe the Carbon interface, which is used to maintain compatibility with OS 9 APIs. 10.0 ran five sub-versions (10.0 through 10.0.4) with relatively minor modifications. The version of the core OS packages, called Darwin, were 1.3.1 in all. XNU was version 123.

10.1 — Puma — a Stronger Feline, but . . .

While definitely novel, OS 10.0 was considered to be immature and unstable, not to mention slow. Although it boasted preemptive multitasking and memory protection, like all its peer operating systems, it still left much to be desired. Some six months later, Mac OS X 10.1, known as Puma, was released to address stability and performance issues, as well as add more user experience features. This also led shortly thereafter to Apple's public abandonment of Mac OS 9, and focus on OS X as the new operating system of choice. Puma ran six sub-versions (10.1 through 10.1.5). In version 10.1.1, Darwin (the core OS) was renumbered from v1.4.1 to 5.1, and since then has followed the OS X numbers consistently by being four numbers ahead of the minor version, and aligning its own minor with the sub-version. XNU was version 201.

10.2 — Jaguar — Getting Better

A year later saw the introduction of Mac OS X 10.2, known as Jaguar, a far more mature OS with myriad UX feature enhancements, and the introduction of the "Quartz Extreme" framework for faster graphics. Another addition was Apple's Bonjour (then called Rendezvous), which is a form of ZeroConf, a uPNP-like protocol (Universal Plug and Play) allowing Apple devices to find one another on a local area network (discussed later in this book). Darwin was updated to 6.0. 10.2 ran nine sub-versions (10.2 through 10.2.8, Darwin 6.0 through 6.8, respectively). XNU was version 344.

10.3 — Panther and Safari

Yet another year passed, and in 2003 Apple released Mac OS X 10.3, Panther, enhancing the OS with yet more UX features such as Exposé. Apple created its own web browser, Safari, displacing Internet Explorer for Mac as it distanced itself from Microsoft.

Another noteworthy improvement in Panther is FileVault, which allows for transparent disk encryption. Mac OS X 10.3 stayed current for a year and a half, and ran 10 sub-versions (10.3 through 10.3.9) with Darwin 7.x (7.0 through 7.9). XNU was version 517.

10.4 — Tiger and Intel Transition

The next update to Mac OS was announced in May 2004, but it took almost a year until Mac OS X 10.4 (Tiger) was officially released. This version sported, as usual, many new GUI features, such as spotlight and dashboard widgets, but also significant architectural changes, most important of which was the foray into the Intel x86 processor space, with 10.4.4. Until that point, Mac OS required a PowerPC architecture. 10.4.4 was also the first OS to introduce the concept of *universal binaries* that could operate on both PPC and x86 architectures. The kernel was significantly improved, allowing for 64-bit pointers.

Other important developer features in this release included four important frameworks: Core Data, Image, Video, and Audio. Core Data handled data manipulation (undo/redo/save). Core Image and Core Video accelerated graphics by exploiting GPUs, and Core Audio built audio right into the OS — allowing for Mac's text-to-speech engine, Voice Over, and the legendary "say" command ("Isn't it nice to have a computer that talks to you?").

Tiger reigned for over two years and a dozen sub-versions — 10.4.0 (Darwin 8.0) through 10.4.11 (Darwin 8.11). XNU was 792.

10.5 — Leopard and UNIX

Leopard was over a year in the making. Announced in June 2006, but not released until October 2007, it boasted hundreds of new features. Chief among them from the developer perspective were:

- Core Animation, which offloaded animation tasks to the framework
- Objective-C 2.0
- OpenGL 2.1
- Improved scripting and new languages, including Python and Ruby
- Dtrace (ported from Solaris 10) and its GUI, Instruments
- FSEvents, allowing for Linux's inotify-like functionality (file system/directory notifications)
- Leopard is also fully UNIX/POSIX-compliant

Leopard ran 10.5 through 1.0.5.8; Darwin 9.0 through 9.8. XNU leapt forward to version 1228.

10.6 — Snow Leopard

Snow Leopard introduced quite a few changes, but mostly under the hood. Following what now was somewhat of a tradition, it took over a year from its announcement in June 2008 to its release in August 2009 From the UX perspective, changes are minimal, although all its applications were ported to 64-bit. The developer perspective, however, revealed significant changes, including:

- **Full 64-bit functionality**: Both in user space libraries and kernel space (K64).
- **File system–level compression**: Incorporated very quietly, as most commands and APIs still report the files' real sizes. In actuality, however, most files — specifically those of the OS — are transparently compressed to save disk space.
- **Grand Central Dispatch**: Enabled multi-core programming through a central API.
- **OpenCL**: Enabled the offloading of computations to the GPU, utilizing the ever-increasing computational power of graphics adapters for non-graphic tasks. Apple originally developed the standard, and still maintains the trademark over the name. Development has been handed over to the Khronos group (`www.khronos.org`), a consortium of industry leaders (including AMD, Intel, NVidia, and many others), who also host OpenGL (for graphics) and OpenSL (for sound).

Snow Leopard finished the process of migration started in 10.4.4 — from PPC to x86/x64 architectures. It no longer supports PowerPCs so universal binaries to support that architecture are no longer needed, saving much disk space by thinning down binaries. In practice, however, most binaries still contain multiple architectures for 32-bit and 64-bit Intel.

The most current version of Snow Leopard is 10.6.8 (Darwin 10.8.0), released July 2011. XNU is version 1504.

10.7 — Lion

Lion is Apple's latest incarnation of OS X at the time of this writing. (More accurately, the latest one publicly available, as Mountain Lion has been released as a developer preview as this book goes to print.) It is a relatively high-end system, requiring Intel Core 2 Duo or better to run on (although successfully virtualized by now).

While it provides many features, most of them are in user mode. Several of the new features have been heavily influenced from iOS (the mobile port of OS X for i-Devices, as we discuss later). These features include, to name but a few:

- **iCloud**: Apple's new cloud-based storage is tightly integrated into Lion, enabling applications to store documents in the cloud directly from the Objective-C runtime and NSDocument.
- **Tighter security**: Drawing on a model that was started in iOS, of application sandboxing and privilege separation.
- **Improvements in the built-in applications**: Such as Finder, Mail, and Preview, as well as porting of apps from iOS, notably FaceTime and the iOS-like LaunchPad.
- **Many framework features**: From overlay scrollbars and other GUI enhancements, through voice over, text auto-correction similar to iOS, to linguistic and part-of-speech tagging to enable Natural Language Processing–based applications.
- **Core Storage**: Allowing logical volume support, which can be used for new partitioning features. A particularly useful feature is extending file systems onto more than one partition.
- **FileVault 2**: Used for encryption of the filesystem, down to the root volume level — marking Apple's entry into the Full Disk Encryption (FDE) realm. This builds on Core Storage's encryption capabilities at the logical volume level. The encryption is AES-128 in XTS mode, which is especially optimized for hard drive encryption. (Both Core Storage and File Vault are discussed in Chapter 15 of this book, "Files and Filesystems.")
- **Air Drop**: Extends Apple's already formidable peer-finding abilities (courtesy of Bonjour) to allow for quick file sharing between hosts over WiFi.
- **64-bit mode**: Enabled by default on more Mac models. Snow Leopard already had a 64-bit kernel, but still booted 32-bit kernels on non-Pro Macbooks.

At the time of this writing, the most recent version of Lion is 10.7.3, XNU version 1699.24.23. With the announcement of Mountain Lion (destined to be 10.8), it seems that Lion will be especially short lived.

10.8 — Mountain Lion

In February 2012, just days before this book was finalized and sent off to print, Apple surprised the world with the announcement of OS X 10.8, Mountain Lion. This is quite unusual, as Apple's OS lifespan is usually longer a year, especially for a cat as big as a Lion, which many believed would end the feline species. The book makes every attempt to also include the most up-to-date material so as to cover Mountain Lion, but the operating system will only be available to the public much later, sometime around the summer of 2012.

Mountain Lion aims to bring iOS and OS X closer together, as was actually speculated in this book (see "The Future of OS X," later in this chapter). Continuing the trend set by Lion, 10.8 further brings features from iOS to OS X, as boasted by its tagline — "Inspired by iPad, reimagined for Mac." The features advertised by Apple are mostly user mode. Interestingly enough, however, the kernel seems to have undergone major revisions as well, as is hinted by its much higher version number — 2050. One notable feature is kernel address space randomization, a feature that is expected to make OS X far more resilient to rootkits and kernel exploitation. The kernel will also likely be 64-bit only, dropping support for 32-bit APIs. The sources for Darwin 12 (and, with them, XNU) will not be available until Mountain Lion is officially released.

Using uname(1)

Throughout this book, many UNIX and OS X-specific commands will be presented. It is only fitting that `uname(1)`, which shows the UNIX system name, be the first of them. Running `uname` will give you the details on the architecture, as well as the version information of Darwin. It has several switches, but `-a` effectively uses all of them. The following code snippets shownin Outputs 1-1a through c demonstrate using `uname` on two different OS X systems:

OUTPUT 1-1A: Using uname(1) to view Darwin version on Snow Leopard 10.6.8, a 32-bit system

```
morpheus@ergo (~) uname -a
Darwin Ergo 10.8.0 Darwin Kernel Version 10.8.0: Tue Jun  7 16:33:36 PDT 2011; root:xnu-
1504.15.3~1/RELEASE_I386 i386
```

OUTPUT 1-1B: Using uname(1) to view Darwin version on Lion 10.7.3, a 64-bit system

```
morpheus@Minion (~) uname -a
Darwin Minion.local 11.3.0 Darwin Kernel Version 11.3.0: Thu Jan 12 18:47:41 PST 2012;
root:xnu-1699.24.23~1/RELEASE_X86_64 x86_64
```

If you use `uname(1)` on Mountain Lion (in the example below, the Developer Preview) you will see an even newer version

OUTPUT 1-1C: Using uname(1) to view Darwin version on Mountain Lion 10.8 (DP3), a 64-bit system

```
morpheus@Simulacrum (~) uname -a
Darwin Simulacrum.local 12.0.0 Darwin Kernel Version 12.0.0: Sun Apr  8 21:22:58 PDT
2012; root:xnu-2050.3.19~1
```

OS X ON NON-APPLE HARDWARE

À la Apple, running OS X on any hardware other than the Apple line of Macs constitutes a violation of the EULA. Apple wages a holy war against Mac clones, and has sued (and won against) companies like Psystar, who have attempted to commercialize non-Apple ports of OS X. This has not deterred many an enthusiast, however, from trying to port OS X to the plain old PC, and — recently — to run under virtualization.

The OpenDarwin/PureDarwin projects take the open source Darwin environment and make of it a fully bootable and installable ISO image. This is carried further by the OSX86 project, which aims to fully port OS X onto PCs, laptops, and even netbooks (this is commonly referred to as "Hackintosh"). With the bootable ISO images, it is possible to circumvent the OS X installer protections and install the system on non-Apple hardware. The hackers (in the good sense of the word) emulate the EFI environment (which is the default on Mac hardware, but still scarce on PC) using a boot loader (Chameleon) based on Apple's Boot-132, which was a temporary boot loader used by Apple back in Tiger v10.4.8. Originally, some minor patches to the kernel were needed, as well — which were feasible since XNU remains open source.

With the rise of virtualization and the accessibility of excellent products such as VMWare, users can now simply download a pre-installed VM image of a fully functioning OS X system. The first images made available were of the later Leopards, and are hard to come by, but now images of the latest Lion and even Mountain Lion are readily downloadable from some sites.

While still in violation of the EULA, Apple does not seem as adamant (yet?) in pursuing the non-commercial ports. It has added features to Lion which require an Internet connection to install (i.e. "Verify the product with Apple"), but still don't manage to snuff the Hackintosh flame. Then again, what people do in the privacy of their own home is their business.

IOS — OS X GOES MOBILE

Windows has its Windows Mobile, Linux has Android, and OS X, too, has its own mobile derivative — the much hyped iOS. Originally dubbed iPhone OS (until mid-2010), Apple (following a short trademark dispute with Cisco), renamed the operating system iOS to reflect the unified nature of the operating system which powers all its i-Devices: the iPhone, iPod, iPad, and Apple TVs.

iOS, like OS X, also has its version history, with its current release at the time of writing being iOS 5.1. Though all versions have code names, they are private to Apple and are usually known only to the jailbreaking community.

1.*x* — Heavenly and the First iPhone

This release ran from the iPhone's inception, in mid-2007, through mid-2008. Version numbers were 1.0 through 1.02, then 1.1 through 1.1.5. The only device supported was initially the iPhone, but the iPod Touch soon followed. The original build was known as "Alpine" (which is also the default root password on i-Devices), but the released version was "Heavenly."

From the jailbreakers' perspective, this release was heavenly, indeed. Full of debug symbols, unencrypted, and straightforward to disassemble. Indeed, many versions later, many jailbreakers still rely on the symbols and function-call graphs extracted from this version.

2.*x* — App Store, 3G and Corporate Features

iPhoneOS 2.0 (known as BigBear) was released along with the iPhone 3G, and both became an instant hit. The OS boasted features meant to make the iPhone more compatible with corporate needs, such as VPN and Microsoft Exchange support. This OS also marked the iPhone going global, with support for a slew of other languages.

More importantly, with this release Apple introduced the App Store, which became the largest software distribution platform in the world, and helped generate even more revenue for Apple as a result of its commission model. (This is so successful that Apple has been trying this, with less success, with the Mac App Store, as of late Snow Leopard).

2.*x* ran 2.0–2.02, 2.1 (SugarBowl), 2.2–2.2.1 (Timberline), until early 2009, and the release of 3.*x*. The XNU version in 2.0.0 is 1228.6.76, corresponding to Darwin 9.3.1.

3.*x* — Farewell, 1st gen, Hello iPad

The 3.*x* versions of iOS brought along the much-longed-for cut/paste, support for lesser used languages, spotlight searches, and many other enhancements to the built-in apps. On the more technical front, it was the first iOS to allow tethering, and allowed the plugging in of Nike+ receivers, demonstrating that the i-Devices could not only be clients but hosts for add-on devices themselves.

3.0 (KirkWood) was quickly superseded by 3.1 (NorthStar), which ran until 3.1.3, the final version supported by the "first generation" devices. Version 3.2 (WildCat) was introduced in April of 2010, especially for the (then mocked) tablet called the iPad. After its web-based jailbreak by Comex (Star 2.0), it was patched to 3.2.2, which was its last version. The Darwin version in 3.1.2 was 10.0.0d3, and XNU was at 1357.5.30.

4.*x* — iPhone 4, Apple TV, and the iPad 2

The 4.*x* versions of iOS brought along many more features and apps, such as FaceTime and voice control, with 4.0 introduced in late June 2010, along with the iPhone 4. 4.*x* versions were the first to support true multitasking, although jailbroken 3.*x* offered a crude hack to that extent.

iOS 4 was the longest running of the iOS versions, going through 4.0–4.0.2 (Apex), 4.1 (Baker or Mohave, which was the first Apple TV version of iOS), and 4.2–4.2.10 (Jasper). Version 4.3

(Durango) brought support for the (by then well respected) iPad 2 and its new dual-core A5 chip. Another important new feature was Address Space Layout Randomization (ASLR, discussed later in this book), which was unnoticeable by users, but — Apple hoped — would prove insurmountable to hackers. Hopes aside, by version 4.3.3 ASLR succumbed to "Saffron" hack when jailbreaker Comex then released his ingenious "Star 3.0" jailbreak for the till-then-unbreakable iPad 2. Apple quickly released 4.3.4 to fix this bug (discussed later in this book as well), and figured the only way to discourage future jailbreaks is to go after the jailbreaker himself — assimilating him. The last release of 4.3.*x* was 4.3.5, which incorporated another minor security fix.

The Darwin version in 4.3.3 is 11.0.0, same as Lion. The XNU kernel, however, is at 1735.46.10 — way ahead of Lion.

5.*x* — To the iPhone 4S and Beyond

iOS is, at the time of this writing, in its fifth incarnation: Telluride (5.0.0 and 5.0.1) and Hoodoo (5.1), named after ski resorts. Initially released as iOS 5.0, it coincided with the iPhone 4S, and introduced (for that phone only) Apple's natural language-based voice control, Siri. iOS5 also boasts many new features, such as much requested notifications, NewsStand (an App Store for digital publications), and some features iOS users never knew they needed, like Twitter integration. Another major enhancement is iCloud (also supported in Lion).

As a result of complaints concerning poor battery life in 5.0, Apple rushed to release 5.0.1, although some complaints persisted. Version 5.1 was released March 2012, coinciding with the iPad 3.

As this book goes to print, the iPhone 4S is the latest and greatest model, and the iPad 3 has just been announced, boasting the improved A5X with quad-core graphics. If Apple's pattern repeats itself, it seems more than likely that it will be followed by the highly anticipated iPhone 5. Apple's upgrade cycles have, thus far, been first for iPad, then iPhone, and finally iPod. From the iOS perspective this matters fairly little — the device upgrades have traditionally focused on better hardware, and fairly few software feature enablers.

Darwin is still at 11.0.0, but XNU is even further ahead of Lion with the version being 1878.11.8 in iOS 5.1.

iOS vs. OS X

Deep down, iOS is really Mac OS X, but with some significant differences:

- The architecture for which the kernel and binaries are compiled is ARM-based, rather than Intel i386 or x86_64. The processors may be different (A4, A5, A5X, etc), but all are based on designs by ARM. The main advantage of ARM over Intel is in power management, which makes their processor designs attractive for mobile operating systems such as iOS, as well as its arch-nemesis, Android.
- The kernel sources remain closed — even though Apple promised to maintain XNU, the OS X Kernel, as open source, it apparently frees itself from that pledge for its mobile version. Occasionally, some of the iOS modifications leak into the publicly available sources (as can be seen by various `#ifdef`, `__arm__`, and `ARM_ARCH` conditionals), though these generally diminish in number with new kernel versions.

- The kernel is compiled slightly differently, with a focus on embedded features and some new APIs, some of which eventually make it to OS X, whereas others do not.
- The system GUI is Springboard, the familiar touch-based application launcher, rather than Aqua, which is mouse-driven and designed for windowing. SpringBoard proved so popular it has actually been (somewhat) back ported into OS X with Lion's LaunchPad.
- Memory management is much tighter, as there is no nigh-infinite swap space to fall on. As a consequence, programmers have to adapt to harsher memory restrictions and changes in the programming model.
- The system is hardened, or "jailed," so as not to allow any access to the underlying UNIX APIs (i.e. Darwin), nor root access, nor any access to any directory but the application's own. Only Apple's applications enjoy the full power of the system. App Store apps are restricted and subject to Apple's scrutiny.

The last point is really the most important: Apple has done its utmost to keep iOS closed, as a specialized operating system for its mobile platforms. In effect, this strips down the operating system to allow developers only the functionality Apple deems as "safe" or "recommended," rather than allow full use of the hardware, which — by itself — is comparable to any decent desktop computer. But these limitations are artificial — at its core, iOS can do nearly everything that OS X can. It doesn't make sense to write an OS from scratch when a good one already exists and can simply be ported. What's more, OS X had already been ported once, from PPC to x86 — and, by induction, could be ported again.

Whether or not you possess an i-Device, you have no doubt heard the much active buzz around the "jailbreaking" procedure, which allows you to overcome the Apple-imposed limitations. Without getting into the legal implications of the procedure (some claim Apple employs more lawyers than programmers), suffice it to say it is possible and has been demonstrated (and often made public) for all i-Devices, from the very first iPhone to the iPhone 4S. Apple seems to be playing a game of cat and mouse with the jailbreakers, stepping up the challenge considerably from version to version, yet there's always "one more thing" that the hackers find, much to Apple's chagrin.

Most of the examples shown in this book, when applied to iOS, require a jailbroken device. Alternatively, you can obtain an iOS software update — which is normally encrypted to prevent any prying eyes such as yours — but can easily be decrypted with well-known decryption keys obtained from certain iPhone-dedicated Wiki sites. Decrypting the iOS image enables you to peek at the file system and inspect all the files, but not run any processes for yourself. For this reason, jailbreaking proves more advantageous. Jailbreaking is about as harmful (if you ask Apple) as open source is bad for your health (if you ask Microsoft). Apple went so far as to "get the facts" and published HT3743[4] about the terrible consequences of "unauthorized modification of iOS." This book will not teach you how to jailbreak, but many a website will happily share this information.

If you were to, say, jailbreak your device, the procedure would install an alternate software package called Cydia, with which you can install third-party apps, that are not App Store approved. While there are many, the ones you'll need to follow along with the examples in this book are:

- **OpenSSH**: Allows you to connect to your device remotely, via the SSH protocol, from any client, OS X, Linux (wherein ssh is a native command line app), or Windows (which has a plethora of SSH clients — for example, PuTTY).

- **Core Utilities:** Packaging the basic utilities you can expect to find in a UNIX /`bin` directory.
- **Adv-cmds and top:** Advanced commands, such as `ps` to view processes.

SSHing to your device, the first command to try would be the standard UNIX `uname` which you saw earlier in the context of OS X. If you try this on an iPad 2 running iOS 4.3.3, for example, you would see something similar to the following:

OUTPUT 1-2A: uname(1) on an iOS 4 iPad 2

```
root@Padishah (/) # uname -a
Darwin Padishah 11.0.0 Darwin Kernel Version 11.0.0: Wed Mar 30 18:52:42 PDT 2011;
root:xnu-1735.46~10/RELEASE_ARM_S5L8940X iPad2,3 arm K95AP Darwin
```

And on an iPod running iOS 5:, you would see the following:

OUTPUT 1-2B: uname(1) on a 4th-generation iPod running iOS 5.0

```
root@Podicum (/) # uname -a
Darwin Podicum 11.0.0 Darwin Kernel Version 11.0.0: Thu Sep 15 23:34:16 PDT 2011;
root:xnu-1878.4.43~2/RELEASE_ARM_S5L8930X iPod4,1 arm N81AP Darwin
```

So, from the kernel perspective, this is (almost) the same kernel, but the architecture is ARM. (S5L8940X is the processor on iPad, commonly known as A5, whereas S5L8930X is the one known as A4. The new iPad is reported as iPad3.1, and its processor, A5X, is identified as S5L8945X).

Table 1-1 partially maps OS X and iOS, in some of their more modern incarnations, to the respective version of XNU. As you can see, until 4.2.1, iOS was using largely the same XNU version as its corresponding OS X at the time. This made it fairly easy to reverse engineer its compiled kernel (and with a fairly large number of debug symbols still present!). With iOS 4.3, however, it has taken off in terms of kernel enhancements, leaving OS X behind. Mountain Lion seems to put OS X back in the lead, but this might very well change if and when iOS 6 comes out.

TABLE 1-1: Mapping of OS X and iOS to their corresponding kernel versions, and approximate release dates.

OPERATING SYSTEM	RELEASE DATE	KERNEL VERSION
Puma (10.1.x)	Sep 2001	201.*.*
Jaguar (10.2.x)	Aug 2002	344.*.*
Panther (10.3.x)	Oct 2003	517.*.*
Tiger (10.4.x)	April 2005	792.*.*
iOS 1.1	June 2007	933.0.0.78
Leopard (10.5.4)	October 2007	1228.5.20

OPERATING SYSTEM	RELEASE DATE	KERNEL VERSION
iOS 2.0.0	July 2008	1228.6.76
iOS 3.1.2	June 2009	1357.5.30
Snow Leopard (10.6.8)	August 2009	1504.15.3
iOS 4.2.1	November 2010	1504.58.28
iOS 4.3.1	March 2011	1735.46
Lion (10.7.0)	August 2011	1699.22.73
iOS 5	October 2011	1878.4.43
Lion (10.7.3)	February 2012	1699.24.23
iOS 5.1	March 2012	1878.11.8
Mountain Lion (DP1)	March 2012	2050.1.12

THE FUTURE OF OS X

At the time of writing, the latest publicly available Mac OS X is Lion, OS X 10.7, with Mountain Lion — OS X 10.8 — lurking in the bushes. Given that the minor version of the latter is already at 8, and the supply of felines has been exhausted, it is also likely to be the last "OS X" branded operating system (although this is, of course, a speculation).

OS X has matured over the past 10 years and has evolved into a formidable operating system. Still, from an architectural standpoint, it hasn't changed that much. The great transition (to Intel architectures) and 64-bit changes aside, the kernel has changed relatively little in the past couple of versions. What, then, may one expect from OS XI?

➤ **The eradication of Mach**: The Mach APIs in the kernel, on which this book will elaborate greatly, are an anachronistic remnant of the NeXTSTEP days. These APIs are largely hidden from view, with most applications using the much more popular BSD APIs. The Mach APIs are, nonetheless, critical for the system, and virtually all applications would break down if they were to be suddenly removed. Still, Mach is not only inconvenient — but also slower. As you will see, its message-passing microkernel-based architecture may be elegant, but it is hardly as effective as contemporary monolithic kernels (in fact, XNU tends toward the monolithic than the microkernel architecture, as is discussed in Chapter 8). There is much to be gained by removing Mach altogether and solidifying the kernel to be fully BSD, though this is likely to be no mere feat.

➤ **ELF binaries**: Another obstacle preventing Mac OS from fully joining the UN*X sorority is its insistence on the Mach-O binary format. Whereas virtually all other UN*X support ELF, OS X does not, basing its entire binary architecture on the legacy Mach-O. If Mach is removed, Mach-O will lose its raison d'etre, and the road to ELF will be paved. This, along with the POSIX compatibility OS X already boasts, could provide both source code and binary compatibility, allowing migrating applications from Solaris, BSD, and Linux to run with no modifications.

- **ZFS**: Much criticism is pointed at HFS+, the native Mac OS file system. HFS+ is itself a patchwork over HFS, which was used in OS 8 and 9. ZFS would open up many features that HFS+ cannot. Core Storage was a giant stride forward in enabling logical volumes and multi-partition volumes, but still leaves much to be desired.
- **Merger with iOS**: At present, features are tried out in OS X, and then sometimes ported to iOS, and sometimes vice versa. For example, Launchpad and gestures, both now mainstream in Lion, originated in iOS. The two systems are very much alike in many regards, but the supported frameworks and features remain different. Lion introduced some UI concepts borrowed from iOS, and iOS 5.0 brings some frameworks ported from OS X. As mobile platforms become stronger, it is not unlikely that the two systems will eventually become closer still, paving the way for running iOS apps, for example, on OS X. Apple has already implemented an architecture translation mechanism before — with Rosetta emulating the PPC architecture on Intel.

SUMMARY

Over the years, Mac OS evolved considerably. It has turned from being the underdog of the operating system world — an OS used by a small but devoted population of die-hard fans — into a mainstream, modern, and robust OS, gaining more and more popularity. iOS, its mobile derivative, is one of the top mobile operating systems in use today.

The next chapters take you through a detailed discussion of OS X internals: Starting with the basic architecture, then diving deeper into processes, threads, debugging, and profiling.

REFERENCES

[1] Amit Singh's Technical History of Apple's Operating Systems: `http://osxbook.com/book/bonus/chapter1/pdf/macosxinternals-singh-1.pdf`

[2] ARS Technica: `http://arstechnica.com`

[3] Wikipedia's Mac OS X entry: `http://en.wikipedia.org/wiki/Mac_OS_X`

[4] "Unauthorized modification of iOS has been a major source of instability, disruption of services, and other issues": `http://support.apple.com/kb/HT3743`

2

E Pluribus Unum: Architecture of OS X and iOS

OS X and iOS are built according to simple architectural principles and foundations. This chapter presents these foundations, and then focuses further on the user-mode components of the system, in a bottom-up approach. The Kernel mode components will be discussed with greater equal detail, but not until the second part of this book.

We will compare and contrast the two architectures — iOS and OS X. As you will see, iOS is in essence, a stripped down version of the full OS X with two notable differences: The architecture is ARM-based (as opposed to Intel x86 or x86_64), and some components have either been simplified or removed altogether, to accommodate for the limitations and/or features of mobile devices. Concepts such as GPS, motion-sensing, and touch — which are applicable at the time of this writing only to mobile devices — have made their debut in iOS, and are progressively being merged into the mainstream OS X in Lion.

OS X ARCHITECTURAL OVERVIEW

When compared to its predecessor, OS 9, OS X is a technological marvel. The entire operating system has been redesigned from its very core, and entirely revamped to become one of the most innovative operating systems available. Both in terms of its Graphical User Interface (GUI) and its underlying programmer APIs, OS X sports many features that are still novel, although are quickly being ported (not to say copied) into Windows and Linux.

Apple's official OS X and iOS documentation presents a very elegant and layered approach, which is somewhat overly simplified:

- **The User Experience layer:** Wherein Apple includes Aqua, Dashboard, Spotlight, and accessibility features. In iOS, the UX is entirely up to SpringBoard, and Spotlight is supported as well.
- **The Application Frameworks layer:** Containing Cocoa, Carbon, and Java. iOS, however, only has Cocoa (technically, Cocoa Touch, a derivative of Cocoa)
- **The Core Frameworks:** Also sometimes called the Graphics and Media layer. Contains the core frameworks, Open GL, and QuickTime.
- **Darwin:** The OS core — kernel and UNIX shell environment.

Of those, Darwin is fully open sourced and serves as the foundation and low-level APIs for the rest of the system. The top layers, however, are closed-source, and remain Apple proprietary.

Figure 2-1 shows a high level architectural overview of these layers. The main difference from Apple's official figure, is that this rendition is tiered in a stair-like manner. This reflects the fact that applications can be written so as to interface directly with lower layers, or even exist solely in them. Command line applications, for example, have no "User Experience" interaction, though they can interact with application or core frameworks.

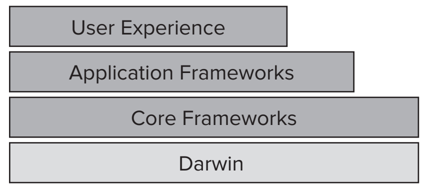


FIGURE 2-1: OS X and iOS architectural diagram

At this high level of simplification, the architecture of both systems conforms to the above figure. But zooming in, one would discover subtle differences. For example, the User Experience of the two systems is different: OS X uses Aqua, whereas iOS uses SpringBoard. The frameworks are largely very similar, though iOS contains some that OS X doesn't, and vice versa.

While Figure 2-1 is nice and clean, it is far too simplified for our purposes. Each layer in it can be further broken down into its constituents. The focus of this book is on Darwin, which is itself not a single layer, but its own tiered architecture, as shown in Figure 2-2.

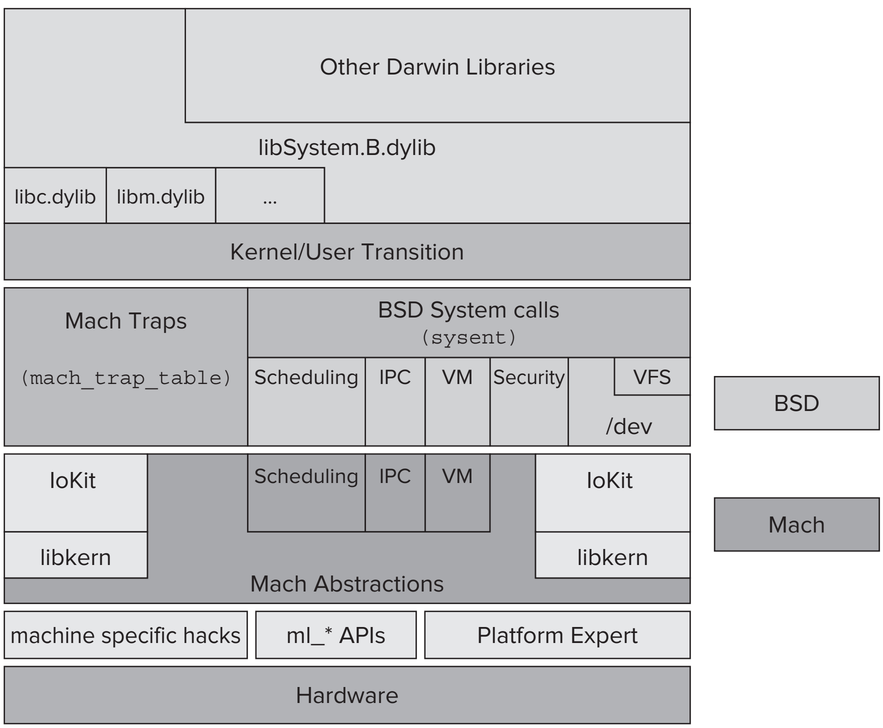


FIGURE 2-2: Darwin Architecture

Figure 2-2 is much closer to depicting the real structure of the Darwin, and particularly its kernel, XNU (though it, too, is somewhat simplified). It reveals an inconvenient truth: XNU is really a hybrid of two technologies: Mach and BSD, with several other components — predominantly IOKit, thrown in for good measure. Unsurprisingly, Apple's neat figures and documentation don't get to this level of unaesthetic granularity. In fact, Apple barely acknowledges Mach.

The good news in all this is that, to some extent, ignorance is bliss. Most user-mode applications, especially if coded in Objective-C, need only interface with the frameworks — primarily Cocoa, the preferred application framework, and possibly some of its core frameworks. Most OS X and iOS developers therefore remain agnostic of the lower layers, Darwin, and most certainly of the kernel. Still, each of the user-mode layers is individually accessible by applications. In the kernel, quite a few components are available to device driver developers. We therefore wade into greater detail in the sections that follow. In particular, we focus on the Darwin shell environment. The second part of this book delves into the kernel.

THE USER EXPERIENCE LAYER

In OS X parlance, the user interface is the User Experience. OS X prides itself on its innovative features, and with good reason. The sleek interface, that debuted with Cheetah and has evolved since, has been a target for imitation, and has influenced other GUI-based operating systems, such as Vista and Windows 7.

Apple lists several components as part of the User Experience layer:

- Aqua
- Quick Look
- Spotlight
- Accessibility options

> *iOS architecture, while basically the same at the lower layers, is totally different at the User Experience level. SpringBoard (the familiar touch driven UI) is entirely responsible for all user interface tasks (as well as myriad other ones). SpringBoard is covered in greater detail in chapter 6.*

Aqua

Aqua is the familiar, distinctive GUI of OS X. Its features, such as translucent windows and graphics effects, are well known but are of less interest in the context of the discussion here. Rather, the focus is how it is actually maintained.

The system's first user-mode process, launchd (which is covered in great depth in Chapter 6) is responsible for starting the GUI. The main process that maintains the GUI is the `WindowServer`. It is intentionally undocumented, and is part of the Core Graphics frameworks buried deep within another framework, Application Services. Thus, the full path to it is `/System/Library/Frameworks/ApplicationServices.framework/Frameworks/CoreGraphics.framework/Resources/WindowServer`.

The window server is started with the `-daemon` switch. Its code doesn't really do anything — all the work is done by the `CGXServer` (Core Graphics X Server) of the CoreGraphics framework. `CGXServer` checks whether it is running as a daemon and/or as the main console getty. It then forks itself into the background. When it is ready, the `LoginWindow` (also started by `launchd`) starts the interactive login process.

> *It is possible to get the system to boot in text console mode, just like the good ol' UNIX days. The setting which controls loginWindow is in* `/etc/ttys`*, under console defined as:*
>
> ```
> root@Ergo (/)# cat /etc/ttys | grep console
> #console     "/usr/libexec/getty std.57600"       vt100   on
> secure
> console "/System/Library/CoreServices/loginwindow.app/Contents/
> MacOS/
> loginwindow" vt100 on secure onoption="/usr/libexec/getty
> std.9600"
> ```
>
> *Uncommenting the first console line will make the system boot into single-user mode. Alternatively, by setting Display Login Window as Name and Password from System Settings ➪ Accounts ➪ Login options, the system console can be accessed by logging in with* `">console"` *as the user name, and no password. If you want back to GUI, a simple CTRL-D (or an exit from the login shell) will resume the Window Server. You can also try* `">sleep"` *and* `">reboot"`

Quicklook

Quicklook is a feature that was introduced in Leopard (10.5) to enable a quick preview from inside the Finder, of various file types. Instead of double-clicking to open a file, it can be QuickLook-ed by pressing the spacebar. It is an extensible architecture, allowing most of the work to be done by plugins. These plugins are bundles with a `.qlgenerator` extension, which can be readily installed by dropping them into the QuickLook directory (system-wide at `/System/Library/QuickLook`; or per user, at `~/Library/QuickLook`).

> *Bundles are a fundamental software deployment architecture in OS X, which we cover in great detail later in this chapter. For now, suffice it to consider a bundle as a directory hierarchy conforming to a fixed structure.*

The actual plug-in is a specially compiled program — but not a standalone executable. Instead of the traditional `main()` entry point, it implements a `QuickLookGeneratorPluginFactory`. A separate configuration file associates the plugin with the file. The file type is specified in what Apple calls UTI, Uniform Type Identifier, which is essentially just reverse DNS notation.

> **REVERSE DNS NOTATION — WHY?**
>
> There is good reasoning for using reverse DNS name as identifiers of software packages. Specifically,
>
> - The Internet DNS format serves as a globally unique hierarchical namespace for host names. It forms a tree, rooted in the null domain (.), with the top-level domains being `.com`, `.net`, `.org`, and so on.
> - The idea of using the same namespace for software originated with Java. To prevent namespace conflict, Sun (now Oracle) noted that DNS can be used — albeit in reverse — to provide a hierarchy that closely resembles a file system.
> - Apple uses reverse DNS format extensively in OS X, as you will see throughout this book.

`quicklookd(8)` is the system "QuickLook server," and is started upon login from the file `/System/Library/LaunchAgents/com.apple.quicklook.plist`. The daemon itself resides within the QuickLook framework and has no GUI. The `qlmanage(1)` command can be used to maintain the plugins and control the daemon, as is shown in Output 2-1:

OUTPUT 2-1: Demonstrating qlmanage(1)

```
morpheus@Ergo (/) % qlmanage -m
 living for 4019s (5 requests handled - 0 generated thumbnails) -
 instant off: yes - arch: X86_64 - user id: 501
memory used: 1 MB (1132720 bytes)
last burst: during 0.010s - 1 requests - 0.000s idle
plugins:
 org.openxmlformats.wordprocessingml.document ->
/System/Library/QuickLook/Office.qlgenerator (26.0)
 com.apple.iwork.keynote.sffkey -> /Library/QuickLook/iWork.qlgenerator
 (11)
   ..
 org.openxmlformats.spreadsheetml.template ->
/System/Library/QuickLook/Office.qlgenerator (26.0)
 com.microsoft.word.stationery -> /System/Library/QuickLook/Office.qlgenerator (26.0)
 com.vmware.vm-package -> /Library/QuickLook/VMware Fusion
 QuickLook.qlgenerator (282344)
 com.microsoft.powerpoint.pot -> /System/Library/QuickLook/Office.qlgenerator (26.0)
```

Spotlight

Spotlight is the quick search technology that Apple introduced with Tiger (10.4). In Leopard, it has been seamlessly integrated into Finder. It has also been ported into iOS, beginning with iOS 3.0. In OS X, the user interacts with it by clicking the magnifying glass icon that is located at the right corner of the system's menu bar. In iOS, a finger swipe to the left of the home screen will bring up a similar window.

The brain behind spotlight is an indexing server, `mds`, located in the MetaData framework, which is part of the system's core services. (`/System/Library/Frameworks/CoreServices.framework/Frameworks/Metadata.framework/Support/mds`). This is a daemon with no GUI. Every time a file operation occurs — creation, modification, or deletion — the kernel notifies this daemon. This notification mechanism, called `fsevents`, is discussed later in this chapter.

When `mds` receives the notification, it then imports, via a Worker process (`mdworker`), various metadata information into the database. The `mdworker` can launch a specific Spotlight Importer to extract the metadata from the file. System-provided importers are in `/System/Library/Spotlight`, and user-provided ones are in `/Library/Spotlight`. Much like QuickLook, they are plugins, implementing a fixed API (which can be generated boilerplate by XCode when a MetaData Importer project is selected).

Spotlight can be accessed from the command line using the following commands:

- `mdutil`: Manages the `MetaData` database
- `mdfind`: Issues spotlight queries
- `mdimport`: Configures and test spotlight plugins
- `mdls`: Lists metadata attributes for file
- `mdcheckschema`: Validates metadata schemata
- `Mddiagnose`: Added in Lion, this utility provides a full diagnostic of the spotlight subsystem (`mds` and `mdworker`), as well as additional data on the system.

Another little documented feature is controlling Spotlight (particularly, `mds`) by creating files in various paths: For example, creating a `.metadata_never_index` hidden file in a directory will prevent its indexing (originally designed for removable media).

DARWIN — THE UNIX CORE

OS X's Darwin is a full-fledged UNIX implementation. Apple makes no attempt to hide it, and in fact takes pride in it. Apple maintains a special document highlighting Darwin's UNIX features[2]. Leopard (10.5) was the first version of OS X to be UNIX-certified. For most users, however, the UNIX interface is entirely hidden: The GUI environment hides the underlying UNIX directories very well. Because this book focuses on the OS internals, most of the discussion, as well as the examples, will draw on the UNIX command line.

The Shell

Accessing the command line is simple — the Terminal application will open a terminal emulator with a UNIX shell. By default this is `/bin/bash`, the GNU "Bourne Again" shell, but OS X provides quite the choice of shells:

- **`/bin/sh` (the Bourne shell):** The basic UNIX shell, created by Stephen Bourne. Considered the standard as of 1977. Somewhat limited.
- **`/bin/bash` (Bourne Again shell):** Default shell. Backward compatible with the basic Bourne shell, but far more advanced. Considered the modern standard on many operating systems, such as Linux and Solaris.

- **`/bin/csh` (C-shell):** An alternative basic shell, with C-like syntax.
- **`/bin/tcsh` (TC-shell):** Like the C-shell, but with more powerful aliasing, completion, and command line editing features.
- **`/bin/ksh` (Korn shell):** Another standard shell, created by David Korn in the 1980s. Highly efficient for scripting, but not too friendly in the command-line environment.
- **`/bin/zsh` (Z-Shell):** A slowly emerging standard, developed at `http://www.zsh.org`. Fully Bourne/Bourne Again compatible, with even more advanced features.

The command line in OS X (and iOS) can also be accessed remotely, over telnet or SSH. Both are disabled by default, and the former (telnet) is highly discouraged as it is inherently insecure and unencrypted. SSH, however, is used as a drop-in replacement (as well as for the former Berkeley "R-utils," such as `rcp/rlogin/rsh`).

Either telnet or SSH can be easily enabled on OS X by editing the appropriate property list file (`telnet.plist`, or `ssh.plist`) in `/System/Library/LaunchDaemons`. Simply set the Disabled key to false, (or remove it altogether). To do so, however, you will need to assume root privileges first — by using `sudo bash` (or another shell of your choice).

On iOS, SSH is disabled by default as well, but on jailbroken systems it is installed and enabled during the jailbreak process. The two users allowed to log in interactively are root (naturally) and mobile. The default root password is alpine, as was the code name for the first version of iOS.

The File System

Mac OS X uses the Hierarchical File System Plus (or HFS+) file system. The "Plus" denotes that HFS+ is a successor to an older Hierarchical File System, which was commonly used in pre-OS X days.

HFS+ comes in four varieties:

- **Case sensitive/insensitive:** HFS+ is always case preserving, but may or may not also be case-sensitive. When set to be case sensitive, HFS+ is referred to as HFSX. HFSX was introduced around Panther, and — while not used in OS X — is the default on iOS.
- **Optional journaling:** HFS+ may optionally employ a journal, in which case it is commonly referred to as JHFS (or JHFSX). A journal enables the file system to be more robust in cases of forced dismounting (for example, power failures), by using a journal to record file system transactions until they are completed. If the file system is mounted and the journal contains transactions, they can be either replayed (if complete) or discarded. Data may still be lost, but the file system is much more likely to be in a consistent state.

In a case-insensitive file system in OS X, files can be created in any uppercase-lowercase combination, and will in fact be displayed in the exact way they were created, but can be accessed by any case combination. As a consequence, two files can never share the same name, irrespective of case. However, accidentally setting caps lock wouldn't affect file system operations. To see for yourself, try LS /ETC/PASSWD.

In iOS, being the case sensitive HFSX by default, case is not only preserved, but allows for multiple files to have the same name, albeit with different case. Naturally, case sensitivity means typos produce a totally different command or file reference, often a wrong one.

The HFS file systems have unique features, like extended attributes and transparent compression, which are discussed in depth in chapter 15. Programmatically, however, the interfaces to the HFS+ and HFSX are the same as other file systems, as well — The APIs exposed by the kernel are actually provided through a common file system adaptation layer, called the Virtual File system Switch (VFS). VFS is a uniform interface for all file systems in the kernel, both UNIX based and foreign. Likewise, both HFS+ and HFSX offer the user the "default" or common UNIX file system user experience — permissions, hard and soft links, file ownership and types are all like other UNIX.

UNIX SYSTEM DIRECTORIES

As a conformant UNIX system, OS X works with the well-known directories that are standard on all UNIX flavors:

- **`/bin`**: Unix binaries. This is where the common UNIX commands (for example, `ls`, `rm`, `mv`, `df`) are.
- **`/sbin`**: System binaries. These are binaries used for system administration, such as file-system management, network configuration, and so on.
- **`/usr`**: The User directory. This is not meant for users, but is more like Windows' program files in that third-party software can install here.
- **`/usr`**: Contains in it `bin`, `sbin`, and `lib`. `/usr/lib` is used for shared objects (think, Windows DLLs and `\windows\system32`). This directory also contains the `include/` subdirectory, where all the standard C headers are.
- **`/etc`**: Et Cetera. A directory containing most of the system configuration files; for example, the password file (`/etc/passwd`). In OS X, this is a symbolic link to `/private/etc`.
- `/dev`: BSD device files. These are special files that represent hardware devices on the system (character and block devices).
- **`/tmp`**: Temporary directory. The only directory in the system that is world-writable (permissions: `rwxrwxrwx`). In OS X, this is a symbolic link to `/private/tmp`.
- **`/var`**: Various. A directory for log files, mail store, print spool, and other data. In OS X, this is a symbolic link to `/private/var`.

The UNIX directories are invisible to Finder. Using BSD's `chflags(2)` system call, a special file attribute of "hidden" makes them hidden from the GUI view. The non-standard option `-O` to `ls`, however, reveals the file attributes, as you can see in Output 2-2. Other special file attributes, such as compression, are discussed in Chapter 14.

OUTPUT 2-2: Displaying file attributes with the non standard "-O" option of ls

```
morpheus@Ergo (/) % ls -lO /
drwxrwxr-x+ 39 root      admin     -           1326 Dec  5 02:42 Applications
drwxrwxr-x@ 17 root      admin     -            578 Nov  5 23:40 Developer
drwxrwxr-t+ 55 root      admin     -           1870 Dec 29 17:23 Library
drwxr-xr-x@  2 root      wheel     hidden        68 Apr 28  2010 Network
```

```
drwxr-xr-x   4 root      wheel     -           136 Nov 11 09:52 System
drwxr-xr-x   6 root      admin     -           204 Nov 14 21:07 Users
drwxrwxrwt@  3 root      admin     hidden      102 Feb  6 11:17 Volumes
drwxr-xr-x@ 39 root      wheel     hidden     1326 Nov 11 09:50 bin
drwxrwxr-t@  3 root      admin     hidden      102 Jan 21 02:40 cores
dr-xr-xr-x   3 root      wheel     hidden     4077 Feb  6 11:17 dev
...
```

OS X–Specific Directories

OS X adds its own special directories to the UNIX tree, under the system root:

- **`/Applications`**: Default base for all applications in system.
- **`/Developer`**: If XCode is installed, the default installation point for all developer tools.
- **`/Library`**: Data files, help, documentation, and so on for system applications.
- **`/Network`**: Virtual directory for neighbor node discovery and access.
- **`/System`**: Used for System files. It contains only a Library subdirectory, but this directory holds virtually every major component of the system, such as frameworks (`/System/Library/Frameworks`), kernel modules (`/System/Library/Extensions`), fonts, and so on.
- **`/Users`**: Home directory for users. Every user has his or her own directory created here.
- **`/Volumes`**: Mount point for removable media and network file systems.
- **`/Cores`**: Directory for core dumps, if enabled. Core dumps are created when a process crashes, if the `ulimit(1)` command allows it, and contain the core virtual memory image of the process. Core dumps are discussed in detail in Chapter 4, "Process Debugging."

iOS File System Idiosyncrasies

From the file system perspective, iOS is very similar to OS X, with the following differences:

- The file system (HFSX) is case-sensitive (unlike OS X's HFS+, which is case preserving, yet insensitive). The file system is also encrypted in part.
- The kernel is already prepackaged with its kernel extensions, as a `kernelcache` (in `/System/Library/Caches/com.apple.kernelcaches`). Unlike OS X kernel caches (which are compressed images), iOS kernel caches are encrypted Img3. This is described in chapter 5.

> *Kernel caches are discussed in Chapter 18, but for now you can simply think of them as a preconfigured kernel.*

- **`/Applications`** may be a symbolic link to `/var/stash/Applications`. This is a feature of the jailbreak, not of iOS.
- There is no `/Users`, but a `/User` — which is a symbolic link to `/var/mobile`

- There is no `/Volumes` (and no need for it, or for disk arbitration, as iOS doesn't have any way to add more storage to a given system)
- `/Developer` is populated only if the i-Device is selected as "Use for development" from within XCode. In those cases, the `DeveloperDiskImage.dmg` included in the iOS SDK is mounted onto the device.

INTERLUDE: BUNDLES

Bundles are a key idea in OS X, which originated in NeXTSTEP and, with mobile apps, has become the de facto standard. The bundle concept is the basis for applications, but also for frameworks, plugins, widgets, and even kernel extensions all packaged into bundles. It therefore makes sense to pause and consider bundles before going on to discuss the particulars of applications as frameworks.

> *The term "bundle" is actually used to describe two different terms in Mac OS: The first is the directory structure described in this section (also sometimes called "package"). The second is a file object format of a shared-library object which has to be explicitly loaded by the process (as opposed to normal libraries, which are implicitly loaded). This is also sometimes referred to as a plug-in.*

Apple defines bundles as "a standardized hierarchical structure that holds executable code and the resources used by that code."[1]. Though the specific type of bundle may differ and the contents vary, all bundles have the same basic directory structure, and every bundle type has the same directories. OS X Application bundles, for example, look like the following code shown in Listing 2-1:

LISTING 2-1: The bundle format of an application

```
Contents/
     CodeResources/
     Info.plist          Main package manifest files
     MacOS/              Binary contents of package
     PkgInfo             Eight character identifier of package
     Resources/          .nib files (GUI) and .lproj files
     Version.plist       Package version information
     _CodeSignature/
          CodeResources
```

Cocoa provides a simple programmatic way to access and load bundles using the NSBundle object, and CoreFoundation's CFBundle APIs.

APPLICATIONS AND APPS

OS X's approach to applications is another legacy of its NeXTSTEP origins. Applications are neatly packaged in bundles. An application's bundle contains most of the files required for the application's runtime: The main binary, private libraries, icons, UI elements, and graphics. The user remains largely oblivious to this, as a bundle is shown in Finder as a single icon. This allows for the easy installation experience in Mac OS — simply dragging an application icon into the Applications folder. To peek inside an application, one would have to use (the non-intuitive) right click.

In OS X, applications are usually located in the `/Applications` folder. Each application is in its own directory, named *AppName*`.app`. Each application adheres quite religiously to a fixed format, discussed shortly — wherein resources are grouped together according to class, in separate sub-directories.

In iOS, apps deviate somewhat from the neat structure — they are still contained in their own directories, but do not adhere as zealously to the bundle format. Rather, the app directory can be quite messy, with all the app files thrown in the root, though sometimes files required for internationalization ("i18n") are in subdirectories (*xxx*.lproj directories, where *xxx* is the language, or ISO language code).

Additionally, iOS distinguishes between the default applications provided by Apple, which reside in `/Applications` (or `/var/stash/Applications` in older jailbreak-versions of iOS), and App Store purchased ones, which are in `/var/mobile/Applications`. The latter is installed in a directory with a specific 128-bit GUID, broken up into a more manageable structure of 4-2-2-2-4 (e.g: `A8CB4133-414E-4AF6-06DA-210490939163` — each hex digit representing 4 bits).

In the GUID-named directory, you can find the usual `.app` directory, along with several additional directories:

This special directory structure, shown in Table 2-1 is required because iOS Apps are `chroot(2)`-ed to their own application directory — the GUID encoded one — and cannot escape it and access the rest of the file system. This ensures that non-Apple applications are so limited that they can't even see what other applications are installed side by side — contributing to the user's privacy and Apple's death grip on the operating system (Jailbreaking naturally changes all that). An application therefore treats its own GUID directory as the root, and when it needs a temporary directory, `/tmp` points to its `GUID/tmp`.

TABLE 2-1: Default directory structure of an iOS app.

IOS APP COMPONENT	USED FOR
`Documents`	Data files saved by the applications (saved high scores for games, documents, notes..)
`iTunesArtwork`	The app's high resolution icon. This is usually a JPG image.
`iTunesMetaData.plist`	The property list of the app, in binary plist format (more on plists follows shortly)
`Library/`	Miscellaneous app files. This is further broken down into Caches, Cookies, Preferences, and sometimes WebKit (for apps with built-in browsing)
`Tmp`	Directory for temporary files

When downloaded from the App Store (or elsewhere), applications are packaged as an `.ipa` file — this is really nothing more than a `zip` file (and may be opened with `unzip(1)`), in which the application directory contents are compressed, under a `Payload/` directory. If you do not have a jail-broken device, try to `unzip -t` an `.ipa` to get an idea of application structure. The `.ipas` are stored locally in `Music/iTunes/iTunes Media/Mobile Applications/`.

Info.plist

The `Info.plist` file, which resides in the `Contents/` subdirectory of `Applications` (and of most other bundles), holds the bundle's metadata. It is a required file, as it supplies information necessary for the OS to determine dependencies and other properties.

The property list format, or `plist`, is well-documented in its own manual page — `plist(5)`. Property lists are stored in one of three formats:

- **XML:** These human-readable lists are easily identified by the XML signature and document type definition (DTD) found in the beginning of the file. All elements of the property list are contained in a `<plist>` element, which in turn defines an array or a dictionary (`<dict>`) — an associative array of keys/values. This is the common format for property lists on OS X.
- **Binary:** Known as `bplists` and identified by the magic of `bplist` at the beginning of the file, these are compiled `plists`, which are less readable by humans, but far more optimized for the OS, as they do not require any complicated XML parsing and processing. Further, it is straightforward to serialize `BPlists`, as data can be simply `memcpy`'d directly, rather than being converted to ASCII. `BPLists` have been introduced with OS X v10.2 and are much more common on iOS than on OS X.
- **JSON:** Using JavaScript Object Notation, the keys/values are stored in a format that is both easy to read, as well as to parse. This format is not as common as either the XML or the Binary.

All three of these formats are, of course, supported natively. In fact, the Objective-C runtime enables developers to be entirely agnostic about the format. In Cocoa, it is simple to instantiate a `Plist` by using the built-in dictionary or array object without having to specify the file format:

```
NSDictionary *dictionary = [NSDictionary dictionaryWithContentsOfURL:plistURL];
NSArray *array = [NSArray arrayWithContentsOfURL:plistURL];
```

Naturally, humans would prefer the XML format. Both OS X and iOS contain a console mode program called `plutil(1)`, which enables you to convert between the various representations. Output 2-3 shows the usage of `plutil(1)` for the conversion:

OUTPUT 2-3: Displaying the Info.plist of an app, after converting it to a more human readable form

```
morpheus@ergo (~) $ cd  ~/Music/iTunes/iTunes\ Media/Mobile\ Applications/

# Note the .ipa is just a zipfile..
morpheus@ergo(Mob..) $ file someApp.ipa
someApp.ipa: Zip archive data, at least v1.0 to extract
```

```
# Use unzip -j to "junk" subdirs and just inflate the file, without directory
structure

morpheus@ergo (Mob..) $ unzip -j someApp.ipa Payload/someApp.app/Info.plist
Archive:  someApp.ipa
 inflating: Info.plist

# Resulting file is a binary plist:

morpheus@ergo (Mob..) $ file Info.plist
Payload/someApp.app/Info.plist: Apple binary property list

# .. which can be converted using plutil..

morpheus@ergo (Mob..) $ plutil -convert xml1 - -o - < Info.plist  > converted.Info.plist

# .. and the be displayed:

morpheus@ergo (Mob..) $ more converted.Info.plist
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>BuildMachineOSBuild</key>
        <string>10K549</string>
        <key>CFBundleDevelopmentRegion</key>
        <string>English</string>
        <key>CFBundleDisplayName</key>
... (output truncated for brevity)...
```

A standard `Info.plist` contains the following entries:

- `CFBundleDevelopmentRegion`: Default language if no user-specific language can be found.
- `CFBundleDisplayName`: The name that is used to display this bundle to the user.
- `CFBundleDocumentTypes`: Document types this will be associated with. This is a dictionary, with the values specifying the file extensions this bundle handles. The dictionary also specifies the display icons used for the associated documents.
- `CFBundleExecutable`: The actual executable (binary or library) of this bundle. Located in `Contents/MacOS`.
- `CFBundleIconFile`: Icon shown in Finder view.
- `CFBundleIdentifier`: Reverse DNS form.
- `CFBundleName`: Name of bundle (limited to 16 characters).
- `CFBundlePackageType`: Specifying a four letter code, for example, `APPL` = Application, `FRMW` = Framework, `BNDL` = Bundle.
- `CFBundleSignature`: Four-letter short name of the bundle.
- `CFBundleURLTypes`: URLs this bundle will be associated with. This is a dictionary, with the values specifying which URL scheme to handle, and how.

All of the keys in the preceding list have the `CF` prefix, as they are defined and handled by the Core Foundation framework. Cocoa applications can also contain `NS` keys, defining application scriptability, Java requirements (if any), and system preference pane integration. Most of the `NS` keys are available only in OS X, and not in iOS.

Resources

The `Resources` directory contains all the files the application requires for its use. This is one of the great advantages of the bundle format. Unlike other operating systems, wherein the resources have to be compiled into the executables, bundles allow the resources to remain separate. This not only makes the executable a lot thinner, but also allows for selective update or addition of a resource, without the need for recompilation.

The resources are very application-dependent, and can be virtually any type of file. It is common, however, to find several recurring types. I describe these next.

NIB Files

`.nib` files are binary plists which contain the positioning and setup of GUI components of an application. They are built using XCode's Interface Builder, which edits the textual versions as `.xib`, before packaging them in binary format (from which point on they are no longer editable). The `.nib` extension dates back to the days of the NEXT Interface Builder, which is the precursor to XCode's. This, too, is a property list, and is in binary form on both OS X and iOS.

The `plutil(1)` command can be used to partially decompile a `.nib` back to its XML representation, although it still won't have as much information as the `.xib` from which it originated (shown in the following code). This is no doubt intentional, as .nib files are not meant to be editable; if they had been, the UI of an application could have been completely malleable externally.

.XIB FILE

```
<?xml version="1.0" encoding="UTF-8"?>
<archive type="com.apple.InterfaceBuilder3.CocoaTouch.XIB" version="7.10">
        <data>
                <int key="IBDocument.SystemTarget">1056</int>
                <string key="IBDocument.SystemVersion">10J869</string>
                <string key="IBDocument.InterfaceBuilderVersion">1306</string>
                <string key="IBDocument.AppKitVersion">1038.35</string>
                <string key="IBDocument.HIToolboxVersion">461.00</string>
                <object class="NSMutableDictionary" key=
                "IBDocument.PluginVersions">
                ...
                        <string key="NS.key.0">com.apple.InterfaceBuilder
                         .IBCocoaTouchPlugin</string>
                        <string key="NS.object.0">301</string>
                </object>
                <object class="NSArray" key="IBDocument
                         .IntegratedClassDependencies">
                        <bool key="EncodedWithXMLCoder">YES</bool>
```

```
                                <string>IBUIButton</string>
                                <string>IBUIImageView</string>
                                <string>IBUIView</string>
                                <string>IBUILabel</string>
                                <string>IBProxyObject</string>
                        </object>
```

.NIB FILE

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/
PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>$archiver</key>
        <string>NSKeyedArchiver</string>
        <key>$objects</key>
        <array>
                <string>$null</string>
                <dict>
                        <key>$class</key>
                        <dict>
                                <key>CF$UID</key>
                                <integer>135</integer>
                        </dict>
                        <key>NS.objects</key>
                        <array>
                                <dict>
                                        <key>CF$UID</key>
                                        <integer>2</integer>
                                </dict>
```

Internationalization with .lproj Files

Bundles have, by design, internationalization support. This is accomplished by subdirectories for each language. Language directories are suffixed with an `.lproj` extension. Some languages are with their English names (English, Dutch, French, etc), and the rest are with their country and language code (e.g. zh_CN for Mandarin, zh_TW for Cantonese). Inside the language directories are string files, `.nib` files and multimedia which are localized for the specific language.

Icons (.icns)

An application usually contains one or more icons for visual display. The application icon is used in the Finder, dock, and in system messages pertaining to the application (for example, Force Quit).

The icons are usually laid out in a single file, *appname*`.icns`, with several resolutions — from 32 × 32 all the way up to a huge 512 × 512.

CodeResources

The last important file an application contains is `CodeResources`, which is a symbolic link to `_CodeSignature/CodeResources`. This file is a property list, containing a listing of all other files in the bundle. The property list is a single entry, `files`, which is a dictionary whose keys are the file names, and whose values are usually hashes, in Base64 format. Optional files have a subdictionary as a value, containing a hash key, and an optional key (whose value is, naturally, a Boolean true).

The CodeResources file helps determine if an application is intact or damaged, as well as prevent accidental modification or corruption of its resources.

Application default settings

Unlike other well known operating systems, neither OS X nor iOS maintain a registry for application settings. This means that an Application must turn to another mechanism to store user preferences, and various default settings.

The mechanism Apple provides is known as defaults, and is yet again, a legacy of NeXTSTEP. The idea behind it is simple: Each application receives its own namespace, in which it is free to add, modify, or remove settings as it sees fit. This namespace is known as the application's domain. Additionally, there is a global domain (NSGlobalDomain) common to all applications.

The application defaults are (usually) stored in property lists. Apple recommends the reverse DNS naming conventions for the `plists`, which are (again, usually) binary, are maintained on a per-user basis, in `~/Library/Preferences`. Additionally, applications can store system-wide (i.e. common to all users) preferences in `/Library/Preferences`. NSGlobalDomain is maintained in a hidden file, `.GlobalPreferences.plist`, which can also exist in both locations.

A system administrator or power user can access and manipulate defaults using the `defaults(1)` command — a generally preferable approach to direct editing of the `plist` files. The command also accepts a `-host` switch, which enables it to set different default settings for the same application on different hosts.

Note, that the defaults mechanism only handles the logistics of storing and retrieving settings. What applications choose to use this mechanism for is entirely up to them. Additionally, some applications (such as VMWare Fusion) deviate from the `plist` requirement and naming convention.

Applications are seldom self-contained. As any developer knows, an application cannot reinvent the wheel, and must draw on operating system supplied functionality and APIs. In UNIX, this mechanism is known as shared libraries. Apple builds on this the idiosyncratic concept of frameworks.

Launching Default Applications

Like most GUI operating systems, OS X keeps an association of file types to their registered applications. This provides for a default application that will be started (or, in Apple-speak, "launched") when a file is double clicked, or a submenu of the registered applications, if the Open With option is selected from the right click menu. This is also useful from a terminal, wherein the `open(1)` command can be used to start the default application associated with the file type.

Windows users are likely familiar with its registry, in which this functionality is implemented (specifically, in subkeys of HKEY_CLASSES_ROOT). OS X provides this functionality a framework called LaunchServices. This framework (which bears no relation to `launchd(1)`, the OS X boot process), is part of the Core Services framework (described later in this chapter).

The launch services framework contains a binary called `lsregister`, which can be used to dump (and also reset) the launch services database, as shown in Listing 2-2:

LISTING 2-2: Using lsregister to view the type registry

```
morpheus@Ergo (~)$ cd /System/Library/Frameworks/CoreServices.Framework
morpheus@Ergo (../Core..work)$ cd Frameworks/LaunchServices.framework/Support
morpheus@Ergo (../Support)$ ./lsregister -dump
Checking data integrity......done.
Status: Database is seeded.
Status: Preferences are loaded.
-----------------------------------------------------------------------------
...  // some lines omitted here for brevity...
bundle id:            1760
      path:          /System/Library/CoreServices/Archive Utility.app
      name:          Archive Utility
      category:
      identifier:    com.apple.archiveutility (0x8000bd0c)
      version:       58
      mod date:      5/5/2011 2:16:50
      reg date:      5/19/2011 10:04:01
      type code:     'APPL'
      creator code:  '????'
      sys version:   0
      flags:         apple-internal  display-name  relative-icon-path  wildcard
      item flags:    container  package  application  extension-hidden  native-app  i386
  x86_64
      icon:          Contents/Resources/bah.icns
      executable:    Contents/MacOS/Archive Utility
      inode:         37623
      exec inode:    37629
      container id:  32
      library:
      library items:
       --------------------------------------------------------
       claim id:            8484
             name:
             rank:          Default
             roles:         Viewer
             flags:         apple-internal  wildcard
             icon:
             bindings:      '****', 'fold'
      --------------------------------------------------------
      claim  id:            8512
            name:          PAX archive
            rank:          Default
            roles:         Viewer
            flags:         apple-default  apple-internal  relative-icon-path
            icon:          Contents/Resources/bah-pax.icns
            bindings:      public.cpio-archive, .pax
       --------------------------------------------------------
```

continues

LISTING 2-2 *(continued)*

```
claim id:           8848
      name:         bzip2 compressed archive
      rank:         Default
      roles:        Viewer
      flags:        apple-default  apple-internal  relative-icon-path
      icon:         Contents/Resources/bah-bzip2.icns
      bindings:     .bzip2
  ...
// many more lines omitted for brevity
```

A common technique used when the Open With menu becomes too overwhelming (often due to the installation of many application), is to rebuild the database with the command: `lsregister -kill -r -domain local -domain system -domain user.`

FRAMEWORKS

Another key component of the OS X landscape are frameworks. Frameworks are bundles, consisting of one or more shared libraries, and their related support files.

Frameworks are a lot like libraries (in fact having the same binary format), but are unique to Apple's systems, and are therefore not portable. They are also not considered to be part of Darwin: As opposed to the components of Darwin, which are all open source, Apple keeps most frameworks in tightly closed source. This is because the frameworks are responsible (among other things) for providing the unique look-and-feel, as well as other advanced features that are offered only by Apple's operating systems — and which Apple certainly wouldn't want ported. The "traditional" libraries still exist in Apple's systems (and, in fact, provide the basis on top of which the frameworks are implemented). The frameworks do, however, provide a full runtime interface, and — especially in Objective-C — serve to hide the underlying system and library APIs.

Framework Bundle Format

Frameworks, like applications (and most other files on OS X), are bundles. Thus, they follow a fixed directory structure:

```
CodeResources/        Symbolic link to Code Signature/CodeResources plist
     Headers/         Symbolic link to Miscellaneous .h files provided by this
 framework
     Resources/       .nib files (GUI), .lproj files, or other files required by
 framework
     Versions/        Subdirectory to allow versioning
          A/          Letter directories denoting version of this framework
          Current/    Symbolic link to preferred framework version
     Framework -name   Symbolic link to framework binary, in preferred version
```

As you can see, however, framework bundles are a bit different than applications. The key difference is in the built-in versioning mechanism: A framework contains one or more versions of the code, which may exist side-by-side in separate subdirectories, such as `Versions/A` , `Versions/B`, and so on. The preferred version can then easily be toggled by creating a symbolic link (shortcut) called `Current`. The framework files themselves are all links to the selected version files. This approach takes after the UN*X model of symbolically linking libraries, but extends it to headers as well. And, while most frameworks still have only one version (usually `A`, but sometimes `B` or `C`), this architecture allows for both forward and backward compatibility.

The OS X and iOS GCC supports a `-framework` switch, which enables the inclusion of any framework, whether Apple supplied or 3rd party. Using this flag provides to the compiler a hint as to where to find the header files (much like the `-I` switch), and to the linker where to find the library file (similar, but not exactly like the `-l` switch)

Finding Frameworks

Frameworks are stored in several locations on the file system:

- `/System/Library/Frameworks`. Contains Apple's supplied frameworks — both in iOS and OS X
- `/Network/Library/Frameworks` may (rarely) be used for common frameworks installed on the network.
- `/Library/Frameworks` holds 3rd party frameworks (and, as can be expected, the directory is left empty on iOS)
- `~/Library/Frameworks` holds frameworks supplied by the user, if any

Additionally, applications may include their own frameworks. Good examples for this are Apple's GarageBand, iDVD, and iPhoto, all of which have application-specific frameworks in `Contents/Frameworks`.

The framework search may be modified further by user-defined variables, in the following order:

- `DYLD_FRAMEWORK_PATH`
- `DYLD_LIBRARY_PATH`
- `DYLD_FALLBACK_FRAMEWORK_PATH`
- `DYLD_FALLBACK_LIBRARY_PATH`

Apple supplies a fair number of frameworks — over 90 in Snow Leopard, and well past 100 in Lion. Even greater in number, however, are the *private* frameworks, which are used internally by the public ones, or directly by Apple's Applications. These reside in `/System/Library/PrivateFrameworks`, and are exactly the same as the public ones, save for header files, which are (intentionally) not included.

Top Level Frameworks

The two most important frameworks in OS X are known as Carbon and Cocoa:

Carbon

Carbon is the name given to the OS 9 legacy programming interfaces. Carbon has been declared deprecated, though many applications, including Apple's own, still rely on it. Even though many of its interfaces are specifically geared for OS 9 compatibility, many new interfaces have been added into it, and it shows no sign of disappearing.

Cocoa

Cocoa is the preferred application programming environment. It is the modern day incarnation of the NeXTSTEP environment, as is evident by the prefix of many of its base classes — NS, short for NeXTSTEP/Sun. The preferred language for programming with Cocoa is Objective C, although it can be accessed from Java and AppleScript as well.

If you inspect the Cocoa and Carbon frameworks, you will see they are both small, almost tiny binaries — around 40k or so on Snow Leopard. That's unusually small for a framework with such a vast API. It's even more surprising, given that Cocoa is a "fat" binary with all three architectures (including the deprecated PPC). The secret to this is that they are built on top of other frameworks, and essentially serve as a wrapper for them — by re-exporting their dependencies' symbols as their own.

The "Cocoa" framework just serves to include three others: AppKit, CoreData and Foundation, which can be seen directly, in its `Headers/cocoa.h`*. In Apple-speak, a framework encapsulating others is often referred to as an* umbrella *framework. The term applies whether the framework merely* `#imports`*, as Cocoa does, or actually contains nested frameworks, as the Application and Core Services frameworks do. This can be seen in the following code:*

```
/*
        Cocoa.h
        Cocoa Framework
        Copyright (c) 2000-2004, Apple Computer, Inc.
        All rights reserved.

        This file should be included by all Cocoa application
source files for easy building.  Using this file is preferred
over importing individual files because it will use a precompiled
version.

        Tools with no UI and no AppKit dependencies may prefer to
include just <Foundation/Foundation.h>.
*/

#import <Foundation/Foundation.h>
#import <AppKit/AppKit.h>
#import <CoreData/CoreData.h>
```

List of OS X and iOS Public Frameworks

Table 2-2 lists the frameworks in OS X and iOS, including the versions in which they came to be supported. The version numbers are from the Apple official documentation [3, 4], wherein similar (and possibly more up to date tables) tables can be found. There is a high degree of overlap in the frameworks, with many frameworks from OS X being ported to iOS, and some (like CoreMedia) making the journey in reverse. This is especially true in the upcoming Mountain Lion, which ports several frameworks like Game Center and Twitter from iOS. Additionally, quite a few of the OS X frameworks exist in iOS as private ones.

TABLE 2-2: Public frameworks in Mac OS X and iOS

FRAMEWORK	OS X	IOS	USED FOR
`AGL`	10.0	--	Carbon interfaces for OpenGL
`Accounts`	10.8	5.0	User account database — Single sign on support
`Accelerate`	10.3	4.0	Accelerated Vector operations
`AddressBook`	10.2	2.0	Address Book functions
`AddressBookUI`	--	2.0	Displaying contact information (iOS)
`AppKit`	10.0	--	One of Cocoa's main libraries (relied on by Cocoa. Framework), and in itself, an umbrella for others. Also contains XPC (which is private in iOS)
`AppKitScripting`	10.0	--	Superseded by Appkit
`AppleScriptKit`	10.0	--	Plugins for AppleScript
`AppleScriptObjC`	10.0	--	Objective-C based plugins for AppleScript
`AppleShareClientCore`	10.0	--	AFP client implementation
`AppleTalk`	10.0	--	Core implementation of the AFP protocol
`ApplicationServices`	10.0	--	Umbrella (headers) for CoreGraphics, CoreText, ColorSync, and others, including SpeechSynthesis (the author's favorite)
`AudioToolBox`	10.0	2.0	Audio recording/handling and others
`AssetsLibrary`	--	4.0	Photos and Videos
`AudioUnit`	10.0	2.0	Audio Units (plug-ins) and Codecs
`AudioVideoBridging`	10.8	--	AirPlay
`AVFoundation`	10.7	2.2	Objective-C support for Audio/Visual media. Only recently ported into Lion

continues

TABLE 2-2 *(continued)*

FRAMEWORK	OS X	IOS	USED FOR
`Automator`	10.4	--	Automator plug-in support
`CalendarStore`	10.5	--	iCal support
`Carbon`	10.0	--	Umbrella (headers) for Carbon, the legacy OS 9 APIs
`Cocoa`	10.0	--	Umbrella (headers) for Cocoa APIs — AppKit, CoreData and Foundation
`Collaboration`	10.5	--	The CBIdentity* APIs
`CoreAudio`	10.0	2.0	Audio abstractions
`CoreAudioKit`	10.4	--	Objective-C interfaces to Audio
`CoreBlueTooth`	--	5.0	BlueTooth APIs
`CoreData`	10.4	3.0	Data model — NSEntityMappings, etc.
`CoreFoundation`	10.0	2.0	Literally, the core framework supporting all the rest through primitives, data structures, etc. (the CF* classes)
`CoreLocation`	10.6	2.0	GPS Services
`CoreMedia`	10.7	4.0	Low-level routines for audio/video
`CoreMediaIO`	10.7	--	Abstraction layer of CoreMedia
`CoreMIDI`	10.0	--	MIDI client interface
`CoreMIDIServer`	10.0	--	MIDI driver interface
`CoreMotion`	--	4.0	Accelerometer/gyroscope
`CoreServices`	10.0	--	Umbrella for AppleEvents, Bonjour, Sockets, Spotlight, FSEvents, and many other services (as sub-frameworks)
`CoreTelephony`	--	4.0	Telephony related data
`CoreText`	10.5	3.2	Text, fonts, etc. On OS X this is a sub framework of ApplicationServices.
`CoreVideo`	10.5	4.0	Video format support used by other libs
`CoreWifi`	10.8	P	Called "MobileWiFi" and private in iOS
`CoreWLAN`	10.6	--	Wireless LAN (WiFi)
`DVComponentGlue`	10.0	--	Digital Video recorders/cameras

FRAMEWORK	OS X	IOS	USED FOR
`DVDPlayback`	10.3	--	DVD playing
`DirectoryService`	10.0	--	LDAP Access
`DiscRecording`	10.2	--	Disc Burning libraries
`DiscRecordingUI`	10.2	--	Disc Burning libraries, and user interface
`DiskArbitration`	10.4	--	Interface to DiskArbitrationD, the system volume manager
`DrawSprocket`	10.0	--	Sprocket components
`EventKit`	10.8	4.0	Calendar support
`EventKitUI`	--	4.0	Calendar User interface
`ExceptionHandling`	10.0	--	Cocoa exception handling
`ExternalAccessory`	--	3.0	Hardware Accessories (those that plug in to iPad/iPod/iPhone)
`FWAUserLib`	10.2	--	FireWire Audio
`ForceFeedback`	10.2	--	Force Feedback enabled devices (joysticks, game-pads, etc)
`Foundation`	10.0	2.0	underlying data structure support
`GameKit`	10.8	3.0	Peer-to-peer connectivity for gaming
`GLKit`	10.8	5.0	OpenGLES helper
`GLUT`	10.0	--	OpenGL Utility framework
`GSS`	10.7	5.0	Generic Security Services API (RFC2078), flavored with some private Apple extensions
`iAd`	--	4.0	Apple's mobile advertisement distribution system
`ICADevices`	10.3	--	Scanners/Cameras (like TWAIN)
`IMCore`	10.6	--	Used internally by InstantMessaging
`ImageCaptureCore`	10.6	P	Supersedes the older ImageCapture
`ImageIO`	--	4.0	Reading/writing graphics formats
`IMServicePlugin`	10.7	--	iChat service providers
`InputMethodKit`	10.5	--	Alternate input methods

continues

TABLE 2-2 *(continued)*

FRAMEWORK	OS X	IOS	USED FOR
`InstallerPlugins`	10.4	--	Plug-ins for system installer
`InstantMessage`	10.4	M	Instant Messaging and iChat
`IOBluetooth`	10.2	--	BlueTooth support for OS X
`IOBluetoothUI`	10.2	--	BlueTooth support for OS X
`IOKit`	10.0	2.0	User-mode components of device drivers
`IOSurface`	10.6	P	Shares graphics between applications
`JavaEmbedding`	10.0-10.7	--	Embeds Java in Carbon. No longer supported in Lion and later
`JavaFrameEmbedding`	10.5	--	Embeds Java in Cocoa
`JavaScriptCore`	10.5	5.0	The Javascript interpreter used by Safari and other WebKit programs.
`JavaVM`	10.0	--	Apple's port of the Java runtime library
`Kerberos`	10.0	--	Kerberos support (required for Active Directory integration and some UNIX domains)
`Kernel`	10.0	--	Required for Kernel Extensions
`LDAP`	10.0	P	Original LDAP support. Superseded by OpenDirectory
`LatentSemanticMapping`	10.5	--	Latent Semantic Mapping
`MapKit`	--	4.0	Embedding maps and geocoding data
`MediaPlayer`	--	2.0	iPod player interface and movies
`MediaToolbox`	10.8	P	
`Message`	10.0	P	Email messaging support
`MessageUI`	--	3.0	UI Resources for messaging and the `Mail.app` (ComposeView and friends)
`MobileCoreServices`	--	3.0	Core Services, light
`Newsstandkit`	--	5.0	Introduced with iOS 5.0's "Newsstand"
`NetFS`	10.6	--	Network File Systems (AFP, NFS)
`OSAKit`	10.4	--	OSA Scripting integration in Cocoa
`OpenAL`	10.4	2.0	Cross platform audio library

FRAMEWORK	OS X	IOS	USED FOR
`OpenCL`	10.6	P	GPU/Parallel Programming framework
`OpenDirectory`	10.6	--	Open Directory (LDAP) objective-C bindings
`OpenGL`	10.0	--	OpenGL — 3D Graphics. Links with OpenCL on supported chipsets.
`OpenGLES`	--	2.0	Embedded OpenGL — replaces OpenGL in iOS
`PCSC`	10.0	--	SmartCard support
`PreferencePanes`	10.0	--	System Preference Pane support. Actual panes are bundles in the `/System/Library/PreferencePanes` folder
`PubSub`	10.5	--	RSS/Atom support
`Python`	10.3	--	The Python scripting language
`QTKit`	10.4	--	QuickTime support
`Quartz`	10.4	--	An umbrella framework containing PDF support, ImageKit, QuartzComposer, QuartzFilters, and QuickLookUI.Responsible for most of the 2D graphics in the system
`QuartzCore`	10.4	2.0	Interface between Quartz and Core frameworks
`QuickLook`	10.5	4.0	Previewing and thumbnailing of files
`QuickTime`	10.0	--	Quicktime embedding
`Ruby`	10.5	--	The popular Ruby scripting language
`RubyCocoa`	10.5	--	Ruby Cocoa bindings
`SceneKit`	10.8	--	3D rendering. Available as a private framework of Lion, but made into a public one in Mountain Lion
`ScreenSaver`	10.0	--	Screen saver APIs
`Scripting`	10.0	--	The original scripting framework. Now superseded
`ScriptingBridge`	10.5	--	Scripting adapters for Objective-C
`Security`	10.0	3.0	Certificates, Keys and secure random numbers
`SecurityFoundation`	10.0	--	SF* Authorization
`SecurityInterface`	10.3	--	SF* headers for UI of certificates, authorization and keychains
`ServerNotification`	10.6	--	Notficiation support

continues

TABLE 2-2 *(continued)*

FRAMEWORK	OS X	IOS	USED FOR
`ServiceManagement`	10.6	--	Interface to launchD
`StoreKit`	10.7	3.0	In-App purchases
`SyncServices`	10.4	--	Sync calendars with .mac
`System`	10.0	2.0	Internally used by other frameworks
`SystemConfiguration`	10.0, 10.3	2.0	SCNetwork, SCDynamicStore
`TWAIN`	10.2	--	Scanner support
`Twitter`	10.8	5.0	Twitter support (in iOS 5)
`Tcl`	10.3	--	TCL Interpreter
`Tk`	10.4	--	Tk Toolkits
`UIKit`	--	2.0	Cocoa Touch — replaces AppKit
`VideoDecodeAcceleration`	10.6.3	--	H.264 acceleration via GPU (TN2267)
`VideoToolkit`	10.8	P	Replaces QuickTime image compression manager and provides video format support
`WebKit`	10.2	P	HTML rendering (Safari Core)
`XgridFoundation`	10.4–10.7	--	Clustering (removed in Mountain Lion)
`vecLib`	10.0	--	Vector calculations (sub framework of Accelerate)

Exercise: Demonstrating the Power of Frameworks

OS X's frameworks really are technological marvels. By any standards, their ingenuity and reusability stands out. There are many stunning examples one can bring using graphical frameworks, but a really useful, and equally impressive example is the `SpeechSynthesis.Framework`.

This framework allows the quick and easy embedding of Text-to-Speech features by drawing on complicated logic which has already been developed (and, to a large part, perfected) by Apple. The `/System/Library/Speech` directory contains the Synthesizers (currently, only one — MacinTalk) which are Mach-O binary bundles, that can be loaded, like libraries, into virtually any process. Additionally, there are quite a few pre-programmed voices (in the `Voices/` subdirectory), and Recognizers (for Speech-to-Text). The voices encode the pitch and other speech parameters, in a proprietary binary form. There is ample documentation about this in the Apple Developer document "The Speech Synthesis API," and a cool utility to customize speech (which is part of XCode) called "Repeat After Me" (`/Developer/Applications/Utilities/Speech/Repeat After Me`).

The average developer, however, needn't care about all this. The Speech Synthesizer can be accessed (among other ways) through the `SpeechSynthesis.Framework`, which itself is under ApplicationServices (Carbon) or AppKit (Cocoa). This enables a C or Objective-C application to enable Text-To-Speech — in one of the many voices on the system — in a matter of several lines of code, as is demonstrated in the following example. The example shows a quick and dirty example of drawing on OS X's text-to-speech.

To not get into the quite messy Objective-C syntax, the next example, shown in Listing 2-3 is in C, and therefore uses the ApplicationServices framework, rather than AppKit.

LISTING 2-3: Demonstrating a very simple (partial) implementation of the say(1) utility

```
#include <ApplicationServices/ApplicationServices.h>

// Quick and dirty (partial) implementation of OS X's say(1) command
// Compile with -framework ApplicationServices

void main (int argc, char **argv)
{

        OSErr rc;
        SpeechChannel channel;
        VoiceSpec vs;
        int voice;
        char *text = "What do you want me to say?";

        if (!argv[1]) { voice = 1; } else { voice = atoi(argv[1]); }

        if (argc == 3) { text = argv[2]; }

        // GetIndVoice gets the voice defined by the (positive) index
        rc= GetIndVoice(voice, // SInt16       index,
                        &vs);  // VoiceSpec *  voice)

        // NewSpeechChannel basically makes the voice usable
        rc = NewSpeechChannel(&vs,// VoiceSpec * voice, /* can be NULL */
                              &channel);


        // And SpeakText... speaks!
        rc = SpeakText(channel,      // SpeechChannel   chan,
                      text,          // const void *    textBuf,
                      strlen(text)); //unsigned long   textBytes)

        if (rc) { fprintf (stderr,"Unable to speak!\n"); exit(1);}

        // Because speech is asynchronous, wait until we are done.
        // Objective-C has much nicer callbacks for this.

        while (SpeechBusy()) sleep(1);
        exit(0);
}
```

The speech framework can also be tapped by other means. There are various bridges to other languages, such as Python and Ruby, and for non-programmers, there is the command line of `say(1)` (which the example mimics), and/or Apple's formidable scripting language, Applescript (accessible via `osascript(1)`). To try this for yourself, have some fun with either command (which can be an inexhaustible font of practical jokes, or other creative uses, as is shown in the comic in Figure 2-3)

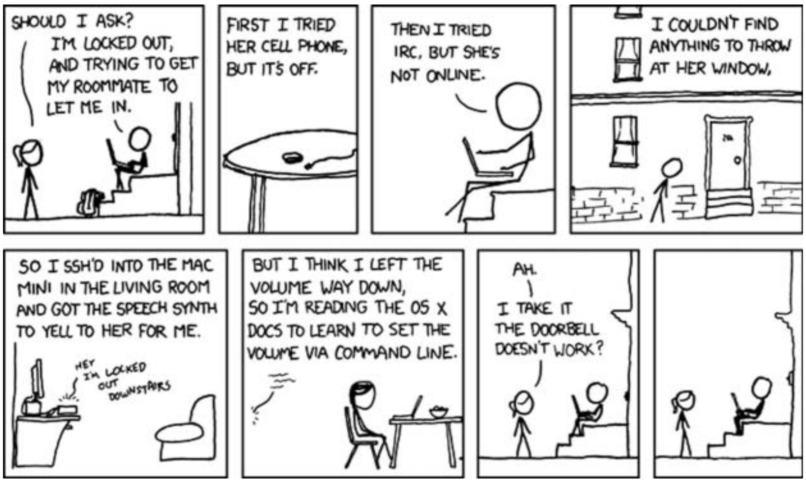


FIGURE 2-3: Other creative uses of OS X Speech, from the excellent site, http://XKCD.com/530 (incidentally, osascript -e "set Volume 10" is what he is looking for)

As stated, an application may be entirely dependent only on the frameworks, which is indeed the case for many OS X and iOS apps. The frameworks themselves, however, are dependent on the operating system libraries, which are discussed next.

LIBRARIES

Frameworks are just a special type of libraries. In fact, framework binaries *are* libraries, as can be verified with the `file(1)` command. Apple still draws a distinction between the two terms, and frameworks tend to be more OS X (and iOS) specific, as opposed to libraries, which are common to all UNIX systems.

OS X and iOS store their "traditional" libraries in `/usr/lib` (there is no `/lib`). The libraries are suffixed with a `.dylib` extension, rather than the customary `.so` (shared object) of ELF on other UNIX. Aside from the different extension (and the different binary format, which is incompatible with `.so`), they are still conceptually the same. You can still find your favorite libraries from other UNIX here, albeit with the `.dylib` format.

If you try to look around the iOS file system — either on a live, jailbroken system, or through an iOS software update image (`.ipsw`), you will see that many of the libraries (and, for that matter, also frameworks), are missing! This is due to an optimization (and possibly obfuscation) technique of library caching, which is discussed in the next chapter. It's easier, therefore to look at the iPhone SDK, wherein the files can be found under `/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS#.#.sdk/`.

The core library — `libc` — has been absorbed into Apple's own `libSystem.B.dylib`. This library also provides the functionality traditionally offered by the math library (`libm`), and PThreads (`libpthread`) — as well as several others, which are all just symbolic links to `libSystem`, as you can see in Output 2-4:

OUTPUT 2-4: Libraries in /usr/lib which are all implemented by libSystem.dylib

```
morpheus@Minion (/)$ ls -l /usr/lib | grep ^l | grep libSystem.dylib
lrwxr-xr-x   1 root  wheel       17 Sep 26 02:08 libSystem.dylib -> libSystem.B.dylib
lrwxr-xr-x   1 root  wheel       15 Sep 26 02:08 libc.dylib -> libSystem.dylib
lrwxr-xr-x   1 root  wheel       15 Sep 26 02:08 libdbm.dylib -> libSystem.dylib
lrwxr-xr-x   1 root  wheel       15 Sep 26 02:08 libdl.dylib -> libSystem.dylib
lrwxr-xr-x   1 root  wheel       15 Sep 26 02:08 libinfo.dylib -> libSystem.dylib
lrwxr-xr-x   1 root  wheel       15 Sep 26 02:08 libm.dylib -> libSystem.dylib
lrwxr-xr-x   1 root  wheel       15 Sep 26 02:08 libpoll.dylib -> libSystem.dylib
lrwxr-xr-x   1 root  wheel       15 Sep 26 02:08 libproc.dylib -> libSystem.dylib
lrwxr-xr-x   1 root  wheel       15 Sep 26 02:08 libpthread.dylib -> libSystem.dylib
lrwxr-xr-x   1 root  wheel       15 Sep 26 02:08 librpcsvc.dylib -> libSystem.dylib
```

Yet, libSystem itself relies on several libraries internal to it — which are found in `/usr/lib/system`. It imports these libraries, and then re-exports their public symbols as if they are its own. In Snow Leopard, there are fairly few such libraries. In Lion and iOS 5, there is a substantial number. This is shown in Output 2-5, which demonstrates using XCode's `otool(1)` to show library dependencies. Note, that because `libSystem` is cached (and therefore not present in the iOS filesystem), it's easier to run it on the iPhone SDK's copy of the library.

OUTPUT 2-5: Dependencies of iOS 5's libSystem using otool(1).

```
morpheus@ergo (.../Developer/SDKs/iPhoneOS5.0.sdk/usr/lib)$  otool -L libSystem.B.dylib
libSystem.B.dylib (architecture armv7):
 /usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version 161.0.0)
 /usr/lib/system/libcache.dylib (compatibility version 1.0.0, current version 49.0.0)
 /usr/lib/system/libcommonCrypto.dylib (compatibility version 1.0.0, current version 40142.0.0)
 /usr/lib/system/libcompiler_rt.dylib (compatibility version 1.0.0, current version 16.0.0)
```

continues

OUTPUT 2-5 *(continued)*

```
/usr/lib/system/libcopyfile.dylib (compatibility version 1.0.0, current version 87.0.0)
/usr/lib/system/libdispatch.dylib (compatibility version 1.0.0, current version 192.1.0)
/usr/lib/system/libdnsinfo.dylib (compatibility version 1.0.0, current version 423.0.0)
/usr/lib/system/libdyld.dylib (compatibility version 1.0.0, current version 199.3.0)
/usr/lib/system/libkeymgr.dylib (compatibility version 1.0.0, current version 25.0.0)
/usr/lib/system/liblaunch.dylib (compatibility version 1.0.0, current version 406.4.0)
/usr/lib/system/libmacho.dylib (compatibility version 1.0.0, current version 806.2.0)
/usr/lib/system/libnotify.dylib (compatibility version 1.0.0, current version 87.0.0)
/usr/lib/system/libremovefile.dylib (compatibility version 1.0.0, current version 22.0.0)
/usr/lib/system/libsystem_blocks.dylib (compatibility version 1.0.0, current version 54.0.0)
/usr/lib/system/libsystem_c.dylib (compatibility version 1.0.0, current version 770.4.0)
/usr/lib/system/libsystem_dnssd.dylib (compatibility version 1.0.0, current version 1.0.0)
/usr/lib/system/libsystem_info.dylib (compatibility version 1.0.0, current version 1.0.0)
/usr/lib/system/libsystem_kernel.dylib (compatibility version 1.0.0, current version 1878.4.20)
/usr/lib/system/libsystem_network.dylib (compatibility version 1.0.0, current version 1.0.0)
/usr/lib/system/libsystem_sandbox.dylib (compatibility version 1.0.0, current version 1.0.0)
/usr/lib/system/libunwind.dylib (compatibility version 1.0.0, current version 34.0.0)
/usr/lib/system/libxpc.dylib (compatibility version 1.0.0, current version 89.5.0)
```

The OS X loader, `dyld(1)`, is also referred to as the Mach-O loader. This is discussed in great detail in the next chapter, which offers an inside view on process loading and execution from the user mode perspective.

OS X contains out-of-box many other open source libraries, which have been included in Darwin (and in iOS). OpenSSL, OpenSSH, libZ, libXSLT, and many other libraries can either be obtained from Apple's open source site, or downloaded from SourceForge and other repositories, and compiled. Ironically enough, it's not the first (nor last) time these open source libraries were the source of iOS jailbreaks (libTiff? FreeType, anyone?)

OTHER APPLICATION TYPES

The Application and App bundles discussed so far aren't the only types of applications that can be created. OS X (and, to a degree iOS) supports several other types of Applications as well.

Java (OS X only)

OS X includes a fully Java 1.6 compliant Java virtual machine. Just like other systems, Java applications are provided as `.class` files. The `.class` file format is not native to OS X — meaning one still needs to use the `java(1)` command-line utility to execute it, just like anywhere else. The JVM implementation, however, is maintained by Apple. The java command line utilities (`java`, `javac`, and `friends`) are all part of the public `JavaVM.framework`. Two other frameworks, `JavaEmbedding.framework` and `JavaFrameEmbedding.framework`, are used to link with and embed Java in Objective-C.

The actual launching of the Java VM process is performed by the private `JavaLaunching.framework`, and `JavaApplicationLauncher.framework`. iOS does not, at present, support Java.

Widgets

Dashboard widgets (or, simply, Widgets) are HTML/Javascript mini-pages, which can be presented by dashboard. These mini-apps are very easy to program (as they are basically the same as web pages), and are becoming increasingly popular.

Widgets are stored in `/Library/Widgets`, as bundles with the `.wdgt` extension. Each such bundle is loosely arranged, containing:

➤ An HTML file (*`widgetname.html`*) which is the Widget's UI. The UI is marked up just like normal HTML, usually with two `<div>` elements — displaying the front and back of the widget, respectively.

➤ A Javascript (JS) file (*`widgetname.js`*) which is the Widget's "engine," providing for its interactivity

➤ A Cascading Style Sheet (CSS) file (*`widgetname.css`*), which provides styles, fonts, etc.

➤ Language directories, like other bundles, containing localized strings

➤ Any images or other files, usually stored in an `Images`/ subdirectory

➤ Any binary plugins, required when the widget cannot be fully implmeneted in Javascript. This is optional (for example, Calculator.wdgt does not have one) and, if present, contains another bundle, with a binary plugin (with a Mach-O binary subtype of "bundle"). These can be loaded into Dashboard itself to provide complicated functionality that needs to break out of the browser environment, for example to access local files.

BSD/Mach Native

Though the preferred language for both iOS and OS X is Objective-C, native applications may be coded in C/C++, and may choose to forego frameworks, working directly with the system libraries and the low-level interfaces of BSD and Mach instead. This allows for the relatively straightforward porting of UNIX code bases, such as PHP, Apache, SSH, and numerous other open-source products. Additionally, initiatives such as "MacPorts" and "fink" go the extra step by packaging these sources, once compiled, into packages akin to Linux's RPM/APT/DEB model, for quick binary installation.

OS X's POSIX compliance makes it very easy to port applications to it, by relying on the standard system calls, and the libraries discussed earlier. This also holds true for iOS, wherein developers have ported everything but the kitchen sink, available through Cydia. There is, however, another subset of APIs — Mach Traps, which remains OS X (and GNUStep) specific, and which coexists with that of BSD. Both of these are explained from the user perspective next.

SYSTEM CALLS

As in all operating systems, user programs are incapable of directly accessing system resources. Programs can manipulate the general-purpose registers and perform simple calculations, but in order to achieve any significant functionality, such as opening a file or a socket, or even outputting a simple message — they must use *system calls*. These are entry points into predefined functions exported by the kernel and accessible in user mode by linking against `/usr/lib/libSystem.B.dylib`. OS X system calls are unusual in that the system actually exports two distinct "personalities" — that of Mach and that of POSIX.

POSIX

Starting with Leopard (10.5), OS X is a certified UNIX implementation. This means that it is fully compliant with the Portable Operating System Interface, more commonly known as POSIX. POSIX is a standard API that defines, specifically:

- **System call prototypes:** All POSIX system calls, regardless of underlying implementation, have the same prototype — i.e., the same arguments and return value. `Open(2)`, for example, is defined on all POSIX systems as:

  ```
  int    open(const char *path, int oflag, ...);
  ```

  `path` is the name of the file name to be opened, and `oflags` is a bitwise `OR` of flags defined in `<fcntl.h>` (for example, `O_RDONLY`, `O_RDWR`, `O_EXCL`).

  This ensures that POSIX-compatible code can be ported — at the source level — between any POSIX compatible operating system. Code from OS X can be ported to Linux, FreeBSD, and even Solaris — as long as it relies on nothing more than POSIX calls and the C/C++ standard libraries.

- **System call numbers:** The key POSIX functions, in addition to the fixed prototype, have well-defined system call numbers. This enables(to a limited extent) *binary portability* — meaning that a POSIX-compiled binary can be ported between POSIX systems of the same underlying architecture (for example, Solaris can run native Linux binaries — both are ELF). OS X *does not* support this, however, because its object format, Mach-O, is incompatible with ELF. What's more, its system call numbers deviate from those of the standard.

The POSIX compatibility is provided by the BSD layer of XNU. The system-call prototypes are in `<unistd.h>`. We discuss their implementations in Chapter 8.

Mach System Calls

Recall that OS X is built upon the Mach kernel, a legacy of NeXTSTEP. The BSD layer wraps the Mach kernel, but its native system calls are still accessible from user mode. In fact, without Mach system calls, common commands such as `top` wouldn't work.

In 32-bit systems, Mach system calls are negative. This ingenious trick enables both POSIX and Mach system calls to exist side by side. Because POSIX only defines non-negative system calls, the negative space is left undefined, and therefore usable by Mach.

In 64-bit systems, Mach system calls are positive, but are prefixed with `0x2000000` — which clearly separates and disambiguates them from the POSIX calls, which are prefixed with `0x1000000`.

The online appendix at `http://newosxbook.com` lists the various POSIX and Mach system calls. We will further cover the transition to Kernel mode in Chapter 8, and the Kernel perspective of system calls and traps in Chapters 9 and 13.

Experiment: Displaying Mach and BSD system calls

System calls aren't called directly, but via thin wrappers in `libSystem.B.dylib`. Using `otool(1)`, the default Mach-O handling tool and disassembler on OS X, you can disassemble (with the `-tV` switch) any binary, and peek inside libSystem. This will enable you to see how the system call interface in OS X works with both Mach and BSD calls.

On a 32-bit system, a Mach system call would look something like this:

```
Morpheus@Ergo (/) % otool -arch i386 -tV /usr/lib/libSystem.B.dylib | more
/usr/lib/libSystem.B.dylib:
(__TEXT,__text) section
_mach_reply_port:
000010c0        movl    $0xffffffe6,%eax    ; Load system call # into EAX
000010c5        calll   __sysenter_trap
000010ca        ret
000010cb        nop                         ; padding to 32-bit boundary
_thread_self_trap:
000010cc        movl    $0xffffffe5,%eax    ; Load system call # into EAX…
000010d1        calll   __sysenter_trap
000010d6        ret
000010d7        nop                         ; padding to 32-bit boundary
__sysenter_trap:
000013d8        popl    %edx
000013d9        movl    %esp,%ecx
000013db        sysenter                    ; Actually execute sysenter
000013dd        nopl    (%eax)
```

The system call number is loaded into the EAX register. Note the number is specified as `0xFFFFxxxx`. Treated as a signed integer, the Mach API calls would be negative. Looking at a BSD system call:

```
Ergo (/) % otool -arch i386 -tV /usr/lib/libSystem.B.dylib -p _chown | more
/usr/lib/libSystem.B.dylib:
(__TEXT,__text) section
_chown:
0005d350        movl    $0x000c0010,%eax    ; load system call -
0005d355        calll   0x00000dd8          ; jump to __sysenter_trap
0005d35a        jae     0x0005d36a          ; if return code >= 0: jump to ret
0005d35c        calll   0x0005d361
0005d361        popl    %edx
0005d362        movl    0x0014c587(%edx),%edx
0005d368        jmp     *%edx
0005d36a        ret
0005d87c        calll   0x0005d881          ; on error…
```

```
0005d881        popl    %edx
0005d882        movl    0x0014c063(%edx),%edx
0005d888        jmp     *%edx
0005d88a        ret
```

The same example, on a 64-bit architecture, reveals a slightly different implementation:

```
Ergo (/) % otool -arch x86_64 -tV /usr/lib/libSystem.B.dylib | more
/usr/lib/libSystem.B.dylib:
(__TEXT,__text) section
_mach_reply_port:
00000000000012a0        movq    %rcx,%r10
00000000000012a3        movl    $0x0100001a,%eax        ; Load system call 0x1a with
                                                        ; flag 0x01
00000000000012a8        syscall                         ; call syscall directly
00000000000012aa        ret
00000000000012ab        nop
```

And, for a POSIX (BSD) system call:

```
Ergo (/) % otool -arch x86_64 -tV /usr/lib/libSystem.B.dylib -p _chown | more
/usr/lib/libSystem.B.dylib:
(__TEXT,__text) section
___chown:
0000000000042f20        movl    $0x02000010,%eax        # Load system call (0x10),
                                                        # with flag 0x02
0000000000042f25        movq    %rcx,%r10
0000000000042f28        syscall                         # call syscall directly
0000000000042f2a        jae     0x00042f31              # if >=0, jump to ret
0000000000042f2c        jmp     cerror                  # else jump to cerror
                                                        # (return -1, set errno)
0000000000042f31        ret
```

If you continue this example and try the ARM architecture (for iOS) as well, you'll see a similar flow, with the system call number loaded into r12, the intra-procedural register, and executed using the `svc` (also sometimes decoded by assemblers as `swi`, or SoftWare Interrupt) command. In the example below (using GDB, though `otool(1)` would work just as well), BSD's `chown(2)` and Mach's `mach_reply_port` are disassembled. Note the latter is loaded with "`mvn`" — Move Negative. The return code is, as usual in ARM, in R0.

```
 (gdb) disass chown
0x30d2ad54 <chown>:        mov    r12, #16              ; 0x10
0x30d2ad58 <chown+4>:      svc    0x00000080
0x32f9c758 <chown+8>:      bcc    0x32f9c770 <chown+32> ; jump to exit on >= 0
0x32f9c75c <chown+12>:     ldr    r12, [pc, #4]         ; 0x32f9c768 <chown+24>
0x32f9c760 <chown+16>:     ldr    r12, [pc, r12]
0x32f9c764 <chown+20>:     b      0x32f9c76c <chown+28>
0x32f9c768 <chown+24>:     bleq   0x321e2a50            ; to errno setting
0x32f9c76c <chown+28>:     bx     r12
0x32f9c770 <chown+32>:     bx     lr
(gdb) disass mach_reply_port
Dump of assembler code for function mach_reply_port:
0x32f99bbc <mach_reply_port+0>:  mvn    r12, #25        ; 0x19
0x32f99bc0 <mach_reply_port+4>:  svc    0x00000080
0x32f99bc4 <mach_reply_port+8>:  bx     lr
```

A HIGH-LEVEL VIEW OF XNU

The core of Darwin, and of all of OS X, is its Kernel, XNU. XNU (allegedly an infinitely recursive acronym for XNU's Not UNIX) is itself made up of several components:

- The Mach microkernel
- The BSD layer
- `libKern`
- I/O Kit

Additionally, the kernel is modular and allows for pluggable *Kernel Extensions* (KExts) to be dynamically loaded on demand.

The bulk of this book — its entire second part — is devoted to explaining XNU in depth. Here, however, is a quick overview of its components.

Mach

The core of XNU, its atomic nucleus, if you will, is Mach. Mach is a system that was originally developed at Carnegie Mellon University (CMU) as a research project into creating a lightweight and efficient platform for operating systems. The result was the Mach microkernel, which handles only the most primitive responsibilities of the operating system:

- Process and thread abstractions
- Virtual memory management
- Task scheduling
- Interprocess communication and messaging

Mach itself has very limited APIs and was not meant to be a full-fledged operating system. Its APIs are discouraged by Apple, although — as you will see — they are fundamental, and without them nothing would work. Any additional functionality, such as file and device access, has to be implemented on top of it — and that is exactly what the BSD layer does.

The BSD Layer

On top of Mach, but still an inseparable part of XNU, is the BSD layer. This layer presents a solid and more modern API that provides the POSIX compatibility discussed earlier. The BSD layer provides higher-level abstractions, including, among others:

- The UNIX Process model
- The POSIX threading model (`Pthread`) and its related synchronization primitives
- UNIX Users and Groups
- The Network stack (BSD Socket API)

➤ File system access

➤ Device access (through the `/dev` directory)

XNU's BSD implementation is largely compatible with FreeBSD's, but does have some noteworthy changes. After covering Mach, this book turns to BSD, focusing on the implementations of the BSD core, and providing specific detail about the virtual file system switch and the networking stack in dedicated chapters.

libkern

Most kernels are built solely in C and low level Assembly. XNU, however, is different. Device drivers — called I/O Kit drivers, and discussed next, can be written in C++. In order to support the C++ runtime and provide the base classes, XNU includes `libkern`, which is a built-in, self-contained C++ library. While not exporting APIs directly to user mode, `libkern` is nonetheless a foundation, without which a great deal of advanced functionality would not be possible.

I/O Kit

Apple's most important modification to XNU was the introduction of the I/O Kit device-driver framework. This is a complete, self-contained execution environment in the kernel, which enables developers to quickly create device drivers that are both elegant and stable. It achieves that by establishing a restricted C++ environment (of `libkern`), with the most important functionality offered by the language — inheritance and overloading.

Writing an I/O Kit driver, then, becomes a greatly simplified matter of finding an existing driver to use as a superclass, and inheriting all the functionality from it in runtime. This alleviates the need for boilerplate code copying, which could lead to stability bugs, and also makes driver code very small — always a good thing under the tight memory constraints of kernel space. Any modification in functionality can be introduced by either adding new methods to the driver or overloading/hiding existing ones.

Another benefit of the C++ environment is that drivers can operate in an object-oriented environment. This makes OS X drivers profoundly different than any other device drivers on other operating systems, which are both limited to C and require hefty code for even the most basic functionality. I/O Kit forms an almost self-contained system in XNU, with a rich environment consisting of many drivers. It could easily be covered in a book of its own (and, in fact, is, in a recent book), though this book dedicates chapter 18 to its architecture.

SUMMARY

This chapter explained the architecture of OS X and iOS. Though the two operating systems are designed for different platforms, they are actually quite similar, with the gaps between them growing narrower still with every new release of either.

The chapter provided a detailed overview, yet still remained at a fairly high level, getting into code samples as little as possible. The next chapter goes deeper and discusses OS X specific APIs — with plenty of actual code samples you can try.

REFERENCES

[1] Apple Developer — Bundle Programming Guide

[2] "OS X for UNIX Users" (Lion version): `http://images.apple.com/macosx/docs/OSX_for_UNIX_Users_TB_July2011.pdf`

[3] Apple Developer — OS X Technology Overview: (details all the frameworks): `http://developer.apple.com/library/mac/#documentation/MacOSX/Conceptual/OSX_Technology_Overview/SystemFrameworks/SystemFrameworks.html`

[4] Details frameworks for iOS: `http://developer.apple.com/library/ios/#documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/iPhoneOSFrameworks/iPhoneOSFrameworks.html`

3

On the Shoulders of Giants: OS X and iOS Technologies

By virtue of being a BSD-derived system, OS X inherits most of the kernel features that are endemic to that architecture. This includes the POSIX system calls, some BSD extensions (such as kernel queues), and BSD's Mandatory Access Control (MAC) layer.

It would be wrong, however, to classify either OS X or iOS as "yet another BSD system" like FreeBSD and its ilk. Apple builds on the BSD primitive's several elaborate constructs — first and foremost being the "sandbox" mechanism for application compartmentalization and security. In addition, OS X and iOS enhance or, in some cases, completely replace BSD components. The venerable `/etc` files, for example, traditionally used for system configuration, are entirely replaced. The standard UN*X `syslog` mechanism is augmented by the Apple System Log. New technologies such as Apple Events and FSEvents are entirely proprietary.

This chapter discusses these features and more, in depth. We first discuss the BSD-inspired APIs, and then turn our attention to the Apple-specific ones. The APIs are discussed from the user-mode perspective, including detailed examples and experiments to illustrate their usage. For the kernel perspective of these APIs, where applicable, see Chapter 14, "Advanced BSD Aspects."

BSD HEIRLOOMS

While the core of XNU is undeniably Mach, its main interface to user mode is that of BSD. OS X and iOS both offer the set of POSIX compliant system calls, as well as several BSD-specific ones. In some cases, Apple has gone several extra steps, implementing additional features, some of which have been back-ported into BSD and OpenDarwin.

sysctl

The `sysctl(8)` command is somewhat of a standardized way to access the kernel's internal state. Introduced in 4.4BSD, it can also be found on other UN*X systems (notably, Linux, where it is backed by the `/proc/sys` directories). By using this command, an administrator can directly query the value of kernel variables, providing important run-time diagnostics. In some cases, modifying the value of the variables, thereby altering the kernel's behavior, is possible. Naturally, only a fairly small subset of the kernel's vast variable base is exported in this way. Nonetheless, those variables that are made visible play key roles in recording or determining kernel functionality.

The `sysctl(8)` command wraps the `sysctl(3)` library call, which itself wraps the `__sysctl` system call (#202). The exported kernel variables are accessed by their *Management Information Base* (MIB) names. This naming convention, borrowed from the Simple Network Management Protocol (SNMP), classifies variables by namespaces.

XNU supports quite a few hard-coded namespaces, as is shown in Table 3-1.

TABLE 3-1: Predefined sysctl Namespaces

NAMESPACE	NUMBER	STORES
`debug`	5	Various debugging parameters.
`hw`	6	Hardware-related settings. Usually all read only.
`kern`	1	Generic kernel-related settings.
`machdep`	7	Machine-dependent settings. Complements the hw namespace with processor-specific features.
`net`	4	Network stack settings. Protocols are defined in their own sub-namespaces.
`vfs`	3	File system-related settings. The Virtual File system Switch is the kernel's common file system layer.
`vm`	2	Virtual memory settings.
`user`	8	Settings for user programs.

As shown in the table, namespaces are translated to an integer representation, and thus the variable can be represented as an array of integers. The library call `sysctlnametomib(3)` can translate from the textual to the integer representation, though that is often unnecessary, because `sysctlbyname(3)` can be used to look up a variable value by its name.

Each namespace may have variables defined directly in it (for example, `kern.ostype`, 1.1), or in sub-namespaces (for example, `kern.ipc.somaxconn`, 1.32.2). In both cases accessing the variable in question is possible, either by specifying its fully qualified name, or by its numeric MIB specifier. Looking up a MIB number by its name (using `sysctlnametomib(3)`) is possible, but not vice versa. Thus, one can walk the MIBs by number, but not retrieve the corresponding names.

Using `sysctl(8)` you can examine the exported values, and set those that are writable. Due to the preceding limitation, however, you cannot properly "walk" the MIBs — that is, traverse the namespaces and obtain a listing of their registered variables, as one would with SNMP's `getNext()`. The command does have an `-A` switch to list all variables, but this is done by checking a fixed list, which is defined in the `<sys/sysctl.h>` header (`CTL_NAMES` and related macros). This is not a problem with the OS X `sysctl(8)`, because Apple does rebuild it to match the kernel version. In iOS, however, Apple does not supply a binary, and the one available from Cydia (as part of the system-cmds package) misses out on iOS-specific variables.

Kernel components can register additional `sysctl` values, and even entire namespaces, on the fly. Good examples are the security namespace (used heavily by the `sandbox` kext, as discussed in this chapter) and the `appleprofile` namespace (registered by the `AppleProfileFamily` kexts — as discussed in Chapter 5, "Process Tracing and Debugging"). The kernel-level perspective of `sysctl`s are discussed in Chapter 14.

The gamut of `sysctl(3)` variables ranges from various minor debug variables to other read/write variables that control entire subsystems. For example, the kernel's little-known kdebug functionality operates entirely through `sysctl(3)` calls. Likewise, commands such as `ps(1)` and `netstat(1)` rely on `sysctl(2)` to obtain the list of PIDs and active sockets, respectively, though this could be achieved by other means, as well.

kqueues

kqueues are a BSD mechanism for kernel event notifications. A kqueue is a descriptor that blocks until an event of a specific type and category occurs. A user (or kernel) mode process can thus wait on the descriptor, providing a simple but effective method for synchronization of one or more processes.

kqueues and their kevents form the basis for asynchronous I/O in the kernel (and enable the POSIX `poll(2)`/`select(2)`, accordingly). A kqueue can be constructed in user mode by simply calling the `kqueue(2)` system call (#362), with no arguments. Then, the specific events of interest can be specified using the `EV_SET` macro, which initializes a `struct kevent`. Calling the `kevent(2)` or `kevent64(2)` system calls (#363 or #369, respectively) will set the event filters, and return if they have been satisfied. The system supports several "predefined" filters, as shown in Table 3-2:

TABLE 3-2: Some of the predefined Event Filters in <sys/event.h>

EVENT FILTER CONSTANT	USAGE
`EVFILT_MACHPORT`	Monitors a Mach port or port set and returns if a message has been received.
`EVFILT_PROC`	Monitors a specified PID for `execve(2)`, `exit(2)`, `fork(2)`, `wait(2)`, or signals.
`EVFILT_READ`	For files, returns when the file pointer is not at EOF. For sockets, pipes, and FIFOs, returns when there is data to read (such as `select(2)`).

continues

TABLE 3-2 *(continued)*

EVENT FILTER CONSTANT	USAGE
`EVFILT_SESSION`	Monitors an audit session (described in the next section).
`EVFILT_SIGNAL`	Monitors a specific signal to the process, even if the signal is currently ignored by the process.
`EVFILT_TIMER`	A periodic timer with up to nanosecond resolution.
`EVFILT_WRITE`	For files, unsupported. For sockets, pipes, and FIFOs, returns when data may be written. Returns buffer space available in event data.
`EVFILT_VM`	Virtual memory Notifications. Used for memory pressure handling (discussed in Chapter 14).
`EVFILT_VNODE`	Filters file (`vnode`)-specific system calls such as `rename(2)`, `delete(2)`, `unlink(2)`, `link(2)`, and others.

Listing 3-1 demonstrates using `kevents` to track process-level events on a particular PID:

LISTING 3-1: Using kqueues and kevents to filter process events

```
void main (int argc, char **argv)
{
   pid_t pid;  // PID to monitor
   int kq;     // The kqueue file descriptor
   int rc;     // collecting return values
   int done;
   struct kevent ke;

   pid = atoi(argv[1]);

   kq = kqueue();

   if (kq == -1) { perror("kqueue"); exit(2); }

   // Set process fork/exec notifications

   EV_SET(&ke, pid, EVFILT_PROC, EV_ADD,
       NOTE_EXIT | NOTE_FORK | NOTE_EXEC , 0, NULL);

   // Register event

   rc = kevent(kq, &ke, 1, NULL, 0, NULL);
   if (rc < 0) { perror ("kevent"); exit (3); }

   done = 0;
   while (!done) {
```

```
            memset(&ke, '\0', sizeof(struct kevent));

            // This blocks until an event matching the filter occurs
            rc = kevent(kq, NULL, 0, &ke, 1, NULL);
            if (rc < 0) { perror ("kevent"); exit (4); }

            if (ke.fflags & NOTE_FORK)
                printf("PID %d fork()ed\n", ke.ident);

            if (ke.fflags & NOTE_EXEC)
                printf("pid %d has exec()ed\n", ke.ident);

            if (ke.fflags & NOTE_EXIT)
               {
                 printf("pid %d has exited\n", ke.ident);
                 done++;
               }

        } // end while
}
```

Auditing (OS X)

OS X contains an implementation of the Basic Security Module, or BSM. This auditing subsystem originated in Solaris, but has since been ported into numerous UN*X implementations (as *OpenBSM*), among them OS X. This subsystem is useful for tracking user and process actions, though may be costly in terms of disk space and overall performance. It is, therefore, of value in OS X, but less so on a mobile system such as iOS, which is why it is not enabled in the latter.

Auditing, as the security-sensitive operation that it is, must be performed at the kernel level. In BSD and other UN*X flavors the kernel component of auditing communicates with user space via a special character pseudo-device (for example, `/dev/audit`). In OS X, however, auditing is implemented over Mach messages.

The Administrator's View

Auditing is a self-contained subsystem in OS X. The main user-mode component is the `auditd(8)`, a daemon that is started on demand by `launchd(8)`, unless disabled (in the `com.apple.auditd.plist` file). The daemon does not actually write the audit log records; those are done directly by the kernel itself. The daemon does control the kernel component, however, and so he who controls the daemon controls auditing. To do so, the administrator can use the `audit(8)` command, which can initialize (`-i`) or terminate (`-t`) auditing, start a new log (`-n`), or expire (`-e`) old logs. Normally, `auditd(8)` times out after 60 seconds of inactivity (as specified in its plist `TimeOut` key). Just because `auditd(8)` is not running, therefore, implies nothing about the state of auditing.

Audit logs, unless otherwise stated, are collected in `/var/audit`, following a naming convention of `start_time.stop_time`, with the timestamp accurate to the second. Logs are continuously generated, so (aside from crashes and reboots), the `stop_time` of a log is also a `start_time` of its successor. The latest log can be easily spotted by its `stop_time` of `not_terminated`, or a symbolic link to `current`, as shown in Output 3-1.

OUTPUT 3-1: Displaying logs in the /var/audit directory

```
root@Ergo (/)# ls -ld /var/audit
drwx------  3247 root  wheel  110398 Mar 19 17:44 /var/audit


root@Ergo (/)# ls -l /var/audit
…
-r--r-----  1 root  wheel     749 Mar 19 16:33 20120319203254.20120319203327
-r--r-----  1 root  wheel     337 Mar 19 17:44 20120319203327.20120319214427
-r--r-----  1 root  wheel       0 Mar 19 17:44 20120319214427.not_terminated
lrwxr-xr-x  1 root  wheel      40 Mar 19 17:44 current ->
                                                /var/audit/20120319214427.not_terminated
```

The audit logs are in a compact binary format, which can be deciphered using the `praudit(1)` command. This command can print the records in a variety of human- and machine-readable formats, such as the default CSV or the more elegant XML (using `–x`). To enable searching through audit records, the `auditreduce(1)` command may be used with an array of switches to filter records by event type (`-m`), object access (`-o`), specific UID (`-e`), and more.

Because logs are cycled so frequently, a special character device, `/dev/auditpipe`, exists to allow user-mode programs to access the audit records in real time. The `praudit(1)` command can therefore be used directly on `/dev/auditpipe`, which makes it especially useful for shell scripts. As a quick experiment, try doing so, then locking your screen saver, and authenticating to unlock it. You should see something like Output 3-2.

OUTPUT 3-2: Using praudit(1) on the audit pipe for real-time events

```
root@Ergo (/)# praudit /dev/auditpipe
header,106,11,user authentication,0,Tue Mar 20 02:26:01 2012, + 180 msec
subject,root,morpheus,wheel,root,wheel,38,0,0,0.0.0.0
text,Authentication for user <morpheus>
return,success,0
trailer,106
```

Auditing must be performed at the time of the action, and can therefore have a noticeable impact on system performance as well as disk space. The administrator can therefore tweak auditing using several files, all in `/etc/security`, listed in Table 3-3.

TABLE 3-3: Files in /etc/security Used to Control Audit Policy

AUDIT CONTROL FILE	USED FOR
`audit_class`	Maps event bitmasks to human-readable names, and to the mnemonic classes used in other files for events.
`audit_control`	Specifies audit policy and log housekeeping.

AUDIT CONTROL FILE	USED FOR
audit_event	Maps event identifiers to mnemonic class and human-readable name.
audit_user	Selectively enables/disables auditing of specific mnemonic event classes on a per-user basis. The record format is: *Username:classes_audited:classes_not_audited*
audit_warn	A shell script to execute on warnings from the audit daemon (for example, "audit space low (< 5% free) on audit log file-system"). Usually passes the message to logger(1).

The Programmer's View

If auditing is enabled, XNU dedicates system calls #350 through #359 to enable and control auditing, as shown in Table 3-4 (all return the standard int return value of a system call: 0 on success, or -1 and set errno on error). On iOS, these calls are merely stubs returning –ENOSYS (0x4E).

TABLE 3-4: System Calls Used for Auditing in OS X, BSM-Compliant

#	SYSTEM CALL	USED TO
350	**audit**(const char *rec, u_int length);	Commit an audit record to the log.
359	**auditctl**(char *path);	Open a new audit log in file specified by path (similar to audit -n)
351	**auditon**(int cmd, void *data, u_int length);	Configure audit parameters. Accepts various A_* commands from <bsm/audit.h>.
355 356	**getaudit** (auditinfo_t *ainfo); **setaudit** (auditinfo_t *ainfo);	Get or set audit session state. The auditinfo_t is defined as struct auditinfo { au_id_t ai_auid; **au_mask_t ai_mask;** au_tid_t ai_termid; **au_asid_t ai_asid; };** These system calls are likely deprecated in Mountain Lion.

continues

TABLE 3-4 *(continued)*

#	SYSTEM CALL	USED TO
357	**`getaudit_addr`** `(auditinfo_addr_t *aa, u_int length);`	As `getaudit` or `setaudit`, but with support for >32-bit `termids`, and an additional 64-bit `ai_flags` field.
358	**`setaudit_addr`** `(auditinfo_addr_t *aa, u_int length);`	
353	**`getauid`**`(au_id_t *auid);`	Get or set the audit session ID.
354	**`setauid`**`(au_id_t *auid);`	

Apple deviates from the BSM standard and enhances it with three additional proprietary system calls, tying the subsystem to the underlying Mach system. Unlike the standard calls, these are undocumented save for their open source implementation, as shown in Table 3-5.

TABLE 3-5: Apple-Specific System Calls Used for Auditing

#	SYSTEM CALL	USED FOR
428	**`mach_port_name_t`** `audit_session_self(void);`	Returns a Mach port (send) for the current audit session
429	**`audit_session_join`** `(mach_port_name_t port);`	Joins the audit session for the given Mach port
432	**`audit_session_port`**`(au_asid_t asid, user_addr_t portnamep);`	New in Lion and relocates `fileport_makeport`. Obtains the Mach port (send) for the given audit session `asid`.

Auditing is revisited from the kernel perspective in Chapter 14.

Mandatory Access Control

FreeBSD 5.x was the first to introduce a powerful security feature known as Mandatory Access Control (MAC). This feature, originally part of Trusted BSD[1], allows for a much more fine-grained security model, which enhances the rather crude UN*X model by adding support for object-level security: limiting access to certain files or resources (sockets, IPC, and so on) by specific processes, not just by permissions. In this way, for example, a specific app could be limited so as not to access the user's private data, or certain websites.

A key concept in MAC is that of a *label*, which corresponds to a predefined classification, which can apply to a set of files or other objects in the system (another way to think of this is as sensitivity tags applied to dossiers in spy movies — "Unclassified," "Confidential," "Top Secret," etc). MAC denies access to any object which does not comply with the label (Sun's swan song, Trusted Solaris, actually made such objects invisible!). OS X extends this further to encompass security policies (for example "No network") that can then be applied to various operations, not just objects.

MAC is a framework — not in the OS X sense, but in the architectural one: it provides a solid foundation into which additional components, which do not necessarily have to be part of the kernel proper, may "plug-in" to control system security. By registering with MAC, specialized kernel extensions can assume responsibility for the enforcement of security policies. From the kernel's side, callouts to MAC are inserted into the various system call implementations, so that each system call must first pass MAC validation, prior to actually servicing the user-mode request. These callouts are only invoked if the kernel is compiled with MAC support, which is on by default in both OS X and iOS. Even then, the callouts return 0 (approving the operation) unless a policy module (specialized kernel extension) has registered for them, and provided its own alternate authorization logic. The MAC layer itself makes no decisions — it calls on the registered policy modules to do so.

The kernel additionally offers dedicated MAC system calls. These are shown in Table 3-6. Most match those of FreeBSD's, while a few are Apple extensions (as noted by the shaded rows).

TABLE 3-6: MAC-Specific System Calls

#	SYSTEM CALL	USED FOR
380	`int __mac_execve(char *fname, char **argp, char **envp, struct mac *mac_p);`	As `execve(2)`, but executes the process under a given MAC label
381	`int __mac_syscall(char *policy, int call, user_addr_t arg);`	MAC-enabled Wrapper for `indirect` syscall.
382 383	`int __mac_[get|set]_file (char *path_p, struct mac *mac_p);`	Get or set label associated with a pathname
384 385	`int __mac_[get|set]_link (char *path_p, struct mac *mac_p);`	Get or set label associated with a link
386 387	`int __mac_[get|set]_proc(struct mac *mac_p);`	Retrieve or set the label of the current process
388 389	`int __mac_[get|set]_fd (int fd, struct mac *mac_p);`	Get or set label associated with a file descriptor. This can be a file, but also a socket or a FIFO
390	`int __mac_get_pid(pid_t pid, struct mac *mac_p);`	Get the label of another process, specified by PID
391	`int __mac_get_lcid(pid_t lcid, struct mac *mac_p);`	Get login context ID
392 393	`int __mac_[get|set]_lctx (struct mac *mac_p);`	Get or set login context ID

continues

TABLE 3-6 *(continued)*

#	SYSTEM CALL	USED FOR
424	`int __mac_mount(char *type, char *path, int flags, caddr_t data, struct mac *mac_p);`	MAC enabled `mount(2)` replacement
425	`int __mac_get_mount(char *path, struct mac *mac_p);`	Get Mount point label information
426	`int __mac_getfsstat(user_addr_t buf, int bufsize, user_addr_t mac, int macsize, int flags);`	MAC enabled `getfsstat(2)` replacement

The administrator can control enforcement of MAC policies on the various subsystems using `sysctl(8)`: MAC dynamically registers and exposes the top-level `security` MIB, which contain enforcement flags, as shown in Output 3-3:

OUTPUT 3-3: The security sysctl MIBs exposed by MAC, on Lion

```
morpheus@Minion (/)$ sysctl security
security.mac.sandbox.sentinel: .sb-4bde45ee
security.mac.qtn.sandbox_enforce: 1
security.mac.max_slots: 7
security.mac.labelvnodes: 0
security.mac.mmap_revocation: 0         # Revoke mmap access to files on subject relabel
security.mac.mmap_revocation_via_cow: 0 # Revoke mmap access to files via copy on write
security.mac.device_enforce: 1
security.mac.file_enforce: 0
security.mac.iokit_enforce: 0
security.mac.pipe_enforce: 1
security.mac.posixsem_enforce: 1        # Posix semaphores
security.mac.posixshm_enforce: 1        # Posix shared memory
security.mac.proc_enforce: 1            # Process operation (including code signing)
security.mac.socket_enforce: 1
security.mac.system_enforce: 1
security.mac.sysvmsg_enforce: 1
security.mac.sysvsem_enforce: 1
security.mac.sysvshm_enforce: 1
security.mac.vm_enforce: 1
security.mac.vnode_enforce: 1           # VFS VNode operations (including code signing)
```

The `proc_enforce` and `vnode_enforce` MIBS are the ones which control, among other things, code signing on iOS. A well known workaround for code signing on jailbroken devices was to manually set both to 0 (i.e. disable their enforcement). Apple made those two settings read only in iOS 4.3 and later, but kernel patching and other methods can still work around this.

MAC provides the substrate for OS X's Compartmentalization ("Sandboxing") and iOS's entitlements. Both are unique to OS X and iOS, and are described later in this chapter under "OS X and iOS Security Mechanisms." The kernel perspective of MAC (including an in-depth discussion of its use in OS X and iOS) is described in Chapter 14.

OS X- AND IOS-SPECIFIC TECHNOLOGIES

Mac OS has, over the years, introduced several avant-garde technologies, some of which still remain proprietary. The next section discusses these technologies, particularly the ones that are of interest from an operating-system perspective.

User and Group Management (OS X)

Whereas other UN*X traditionally relies on the age-old password files (`/etc/passwd` and, commonly `/etc/shadow`, used for the password hashes), which are still used in single-user mode (and on iOS), with `/etc/master.passwd` used as the shadow file. In all other cases, however, OS X deprecates them in favor of its own directory service: `DirectoryService(8)` on Snow Leopard, which has been renamed to `opendirectoryd(8)` as of Lion. The daemon's new name reflects its nature: It is an implementation of the OpenLDAP project. Using a standard protocol such as the Lightweight Directory Access Protocol (LDAP) enables integration with non-Apple directory services as well, such as Microsoft's Active Directory. (Despite the "lightweight" moniker, LDAP is a lengthy Internet standard covered by RFCs 4510 through 4519. It is a simplified version of DAP, which is an OSI standard).

The directory service maintains more than just the users and groups: It holds many other aspects of system configuration, as is discussed under "System Configuration" later in the chapter.

To interface with the daemon, OS X supplies a command line utility called `dscl(8)`. You can use this tool, among other things, to display the users and groups on the system. If you try `dscl . -read /Users/username` on yourself (the "." is used to denote the default directory, which is also accessible as `/Local/Default`), you should see something similar to Output 3-4:

OUTPUT 3-4: Running dscl(8) to read user details from the local directory

```
morpheus@ergo(/)$ dscl . -read /Users/ `whoami `
dsAttrTypeNative:_writers_hint: morpheus
dsAttrTypeNative:_writers_jpegphoto: morpheus
dsAttrTypeNative:_writers_LinkedIdentity: morpheus
dsAttrTypeNative:_writers_passwd: morpheus
dsAttrTypeNative:_writers_picture: morpheus
dsAttrTypeNative:_writers_realname: morpheus
dsAttrTypeNative:_writers_UserCertificate: morpheus
AppleMetaNodeLocation: /Local/Default
AuthenticationAuthority: ;ShadowHash; ;Kerberosv5;;morpheus@LKDC:SHA1.3023D12469030DE9DB
FE2C2621A01C121615DC80;LKDC:SHA1.3013D12469030DE9DBFD2C2621A07C123615DC70;
AuthenticationHint:
GeneratedUID: 11E111F7-910C-2410-9BAB-ABB20FE3DF2A
JPEGPhoto:
 ffd8ffe0 00104a46 49460001 01000001 00010000 ffe20238 4943435f 50524f46 494c4500..
```

continues

OUTPUT 3-4 *(continued)*

```
 ... User photo in JPEG format
NFSHomeDirectory: /Users/morpheus
Password: ********
PasswordPolicyOptions:
 <?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/
PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
       <key>failedLoginCount</key>
       <integer>0</integer>
       <key>failedLoginTimestamp</key>
       <date>2001-01-01T00:00:00Z</date>
       <key>lastLoginTimestamp</key>
       <date>2001-01-01T00:00:00Z</date>
       <key>passwordTimestamp</key>
       <date>2011-09-24T20:23:03Z</date>
</dict>
</plist>
Picture:
 /Library/User Pictures/Fun/Smack.tif
PrimaryGroupID: 20
RealName: Me
RecordName: morpheus
RecordType: dsRecTypeStandard:Users
UniqueID: 501
UserShell: /bin/zsh
```

You can also use the `dscl(8)` tool to update the directory and create new users. The shell script in Listing 3-2 demonstrates the implementation of a command-line `adduser`, which OS X does not provide.

LISTING 3-2: A script to perform the function of adduser (to be run as root)

```
#!/bin/bash
# Get username, ID and full name field as arguments from command line
USER=$1
ID=$2
FULLNAME=$3
# Create the user node
dscl . -create /Users/$USER
# Set default shell to zsh
dscl . -create /Users/$USER UserShell /bin/zsh
# Set GECOS (full name for finger)
dscl . -create /Users/$USER RealName "$FULLNAME"
dscl . -create /Users/$USER UniqueID $ID
# Assign user to gid of localaccounts
dscl . -create /Users/$USER PrimaryGroupID 61
# Set home dir (~$USER)
dscl . -create /Users/$USER NFSHomeDirectory /Users/$USER
```

```
# Make sure home directory is valid, and owned by the user
mkdir /Users/$USER
chown $USER /Users/$USER
# Optional: Set the password.
dscl . -passwd /Users/$USER "changeme"
# Optional: Add to admin group
dscl . -append /Groups/admin GroupMembership $USER
```

> *One of Lion's early security vulnerabilities was that* `dscl(8)` *could be used to change passwords of users without knowing their existing passwords, even as a non-root user. If you keep your OS X constantly updated, chances are this issue has been resolved by a security update.*
>
> *The standard UNIX utilities of* `chfn(1)` *and* `chsh(1)`, *which enable the modification of the full name and shell for a given user, respectively, are implemented transparently over directory services by launching the default editor to allow root to type in the fields, rather than bother with* `dscl(8)` *directly. Most administrators, of course, probably use the system configuration GUI — a much safer option, though not as scalable when one needs to create more than a few users.*

System Configuration

Much like it deprecates `/etc` user database files, OS X does away with most other configuration files, which are traditionally used in UN*X as the system "registry."

To maintain system configuration, OS X and iOS use a specialized daemon: – `configd(8)`. This daemon can load additional loadable bundles ("plug-ins") located in the `/System/Library/SystemConfiguration/` directory, which include IP and IPv6 configuration, logging, and other bundles. The average user, of course, is blissfully unaware of this, as the System Preferences application can be used as a graphical front-end to all the configuration tasks.

Command line-oriented power users can employ a specialized tool, `scutil(8)` in order to navigate and query the system configuration. This interactive utility can list and show keys as shown in the following code snippet:

```
root@Padishah (~)# scutil
> list
  subKey [0] = Plugin:IPConfiguration
  subKey [1] = Plugin:InterfaceNamer
  subKey [2] = Setup:
  subKey [3] = Setup:/
  subKey [4] = Setup:/Network/Global/IPv4
  subKey [5] = Setup:/Network/HostNames
  ...
  subKey [50] = com.apple.MobileBluetooth
  subKey [51] = com.apple.MobileInternetSharing
  subKey [52] = com.apple.network.identification

> show com.apple.network.identification
<dictionary> {
  ActiveIdentifiers : <array> {
    0 : IPv4.Router=192.168.1.254;IPv4.RouterHardwareAddress=00:43:a3:f2:81:d9
  }
```

```
  PrimaryIPv4Identifier : IPv4.Router=192.168.1.254;IPv4.RouterHardwareAddress=
00:43:a3:f2:81:d9
  ServiceIdentifiers : <array> {
    0 : 12C4C9CC-7E42-1D2D-ACF6-AAF7FFAF2BFC
  }
}
```

The public `SystemConfiguration.framework` allows programmatic access to the system configuration. Commands such as OS X's `pmset(1)`, which configures power management settings, link with this framework. The framework exists in OS X and iOS, so the program shown in Listing 3-3 can compile and run on both.

LISTING 3-3: Using the SystemConfiguration APIs to query values

```
#include <SystemConfiguration/SCPreferences.h>
// Also implicitly uses CoreFoundation/CoreFoundation.h

void dumpDict(CFDictionaryRef dict){
    // Quick and dirty way of dumping a dictionary as XML
    CFDataRef xml = CFPropertyListCreateXMLData(kCFAllocatorDefault,
                                                (CFPropertyListRef)dict);
    if (xml) {
        write(1, CFDataGetBytePtr(xml), CFDataGetLength(xml));
        CFRelease(xml);
    }
}

void main (int argc, char **argv)
{
  CFStringRef myName = CFSTR("com.technologeeks.SystemConfigurationTest");
  CFArrayRef  keyList;
  SCPreferencesRef prefs = NULL;
  char *val;
  CFIndex i;
  CFDictionaryRef global;

  // Open a preferences session
  prefs = SCPreferencesCreate (NULL,   // CFAllocatorRef allocator,
                               myName, // CFStringRef name,
                               NULL);  // CFStringRef prefsID

  if (!prefs) { fprintf (stderr,"SCPreferencesCreate"); exit(1); }

  // retrieve preference namespaces
  keyList =  SCPreferencesCopyKeyList (prefs);

  if (!keyList) { fprintf (stderr,"CopyKeyList failed\n"); exit(2);}

  // dump 'em
  for (i = 0; i < CFArrayGetCount(keyList); i++) {
       dumpDict(SCPreferencesGetValue(prefs, CFArrayGetValueAtIndex(keyList, i)));
       }

}
```

The dictionaries dumped by this program are naturally maintained in plist files. The default location for these dictionaries is in `/Library/Preferences/SystemConfiguration`. If you compare the output of this program with that of the `preferences.plist` file from that directory, you will see it matches.

Experiment: Using scutil(8) for Network Notifications

You can also use the `scutil(8)` command to watch for system configuration changes, as demonstrated in the following experiment:

1. **Using `scutil(8)`, set a watch on the state of the Airport interface** (if you have one, otherwise the primary Ethernet interface will do):

```
> n.add State:/Network/Interface/en0/AirPort
> n.watch
#  verify the notification was added
> n.list
  notifier key [0] = State:/Network/Interface/en0/AirPort
```

2. **Disable Airport (or unplug your network cable).** You should see notification messages break through the `scutil` prompt:

```
notification callback (store address = 0x10010a150).
  changed key [0] = State:/Network/Interface/en0/AirPort
notification callback (store address = 0x10010a150).
  changed key [0] = State:/Network/Interface/en0/AirPort
notification callback (store address = 0x10010a150).
  changed key [0] = State:/Network/Interface/en0/AirPort
```

3. **Use the "show" subcommand to see the changed key.** In this case, the power status value has been changed:

```
> show State:/Network/Interface/en0/AirPort
<dictionary> {
  Power Status : 0
  SecureIBSSEnabled : FALSE
  BSSID : <data> 0x0013d37f84d9
  Busy : FALSE
  SSID_STR : AAAA
  SSID : <data> 0x41414141
  CHANNEL : <dictionary> {
    CHANNEL : 11
    CHANNEL_FLAGS : 10
  }
}
```

In order to watch for changes programmatically, you can use the `SCDynamicStore` class. Because obtaining the network connectivity status is a common action, Apple provides the far simpler `SCNetworkReachability` class. Apple Developer also provides sample code demonstrating the usage of the class.[2]

Logging

With the move to a BSD-based platform, OS X also inherited support for the traditional UNIX System log. This support (detailed in Apple Technical Article TA26117[3]) provides the full compatibility with the ages-old mechanism commonly referred to as `syslogd(8)`.

The syslog mechanism is well detailed in many other references (including the aforementioned technical article). In a nutshell, it handles textual messages, which are classified by a message *facility* and *severity*. The facility is the class of the reporting element: essentially, the message source. The various UNIX subsystems (mail, printing, cron, and so on) all have their own facilities, as does the kernel (LOG_KERN, or "kern"). Severities range from LOG_DEBUG and LOG_INFO ("About to open file..."), through LOG_ERR ("Unable to open file"), LOG_CRIT ("Is that a bad sector?"), LOG_ALERT ("Hey, where's the disk?!"), and finally, to LOG_EMERG ("Meltdown imminent!"). By using the configuration file /etc/syslog.conf, the administrator can decide on *actions* to take, corresponding to facility/severity combinations. Actions include the following:

- Message certain usernames specified
- Log to files or devices (specified as a full path, starting with "/" so as to disambiguate files from usernames)
- Pipe to commands (|/path/to/program)
- Send to a network host (*@loghost*)

Programmers interface with syslog using the syslog(3) API, consisting of a call to openlog() (specifying their name, facility, and other options), through syslog(), which logs the messages with a given priority. The syslog daemon intercepts the messages through a UNIX domain socket (traditionally /dev/log, though in OS X this has been changed to /var/run/syslog).

OS X 10.4 (Tiger) introduced a new model for logging called the Apple System Log, or ASL. This new architecture (which is also used in iOS) aims to provide more flexibility than is provided by syslog. ASL is modeled after syslog, with the same levels and severities, but allows more features, such as filtering and searching not offered by syslog.

ASL is modular in that it simultaneously offers four logging interfaces:

- **The backward-compatible syslogd:** Referred to as BSD logging, ASL can be configured to accept syslog messages (using -bsd_in 1), and process them according to /etc/syslog.conf (using -bsd_out 1). In OS X, these are enabled by default, but *not so on iOS*. The messages, as in syslogd, come in through the /var/run/syslog socket.
- **The network protocol syslogd:** On the well-known UDP port 514, this protocol may be enabled by -udp_in 1. It is actually enabled by default, but ASL/syslogd relies on launchd(8) for its socket handling, and therefore the socket is not active by default.
- **The kernel logging interface:** Enabled (the default) by -klog_in 1, this interface accepts kernel messages from /dev/log (a character device, incorrectly specified in the documentation as a UNIX domain socket).
- **The new ASL interface:** By using -asl_in 1, which is naturally enabled by default, ASL messages can be obtained from clients of the asl(3) API using asl_log(3) and friends. These messages come in through the /var/run/asl_input socket, and are of a different format than the syslogd ones (hence the need for two separate sockets).

ASL logs are collected in /var/log/asl. They are managed (rotated/deleted) by the aslmanager(8) command, which is automatically run by launchd (from com.apple.aslmanager.plist). You may also run the command manually.

ASL logs, unlike syslog files, are binary, not text. This makes them somewhat smaller in size, but not as `grep(1)`-friendly as `syslog`'s. Apple includes the `syslog(1)` command in OS X to display and view logs, as well as perform searches and filters.

Experiment: Enabling System Logging on a Jailbroken iOS

Apple has intentionally disabled the legacy BSD `syslog` interface, but re-enabling it is a fairly simple matter for the root user via a few simple steps:

1. **Create an `/etc/syslog.conf` file.** The easiest way to create a valid file is to simply copy a file from an OS X installation. The default `syslog.conf` looks something like Listing 3-4:

LISTING 3-4: A default /etc/syslog.conf, from an OS X system

```
*.notice;authpriv,remoteauth,ftp,install,internal.none      /var/log/system.log
kern.*                                                      /var/log/kernel.log

# Send messages normally sent to the console also to the serial port.
# To stop messages from being sent out the serial port, comment out this line.
#*.err;kern.*;auth.notice;authpriv,remoteauth.none;mail.crit        /dev/tty.serial

# The authpriv log file should be restricted access; these
# messages shouldn't go to terminals or publically-readable
# files.
auth.info;authpriv.*;remoteauth.crit                       /var/log/secure.log

lpr.info                                                   /var/log/lpr.log
mail.*                                                     /var/log/mail.log
ftp.*                                                      /var/log/ftp.log
install.*                                                  /var/log/install.log
install.*                                                  @127.0.0.1:32376
local0.*                                                   /var/log/appfirewall.log
local1.*                                                   /var/log/ipfw.log

*.emerg                                                     *
```

2. **Enable the `–bsd_out` switch for syslogd.** The `syslogd` process is started both in iOS and OS X by `launchd(8)`. To change its startup parameters, you must modify its property list file. This file is aptly named `com.apple.syslogd.plist`, and you can find it in the standard location for all launch daemons: `/System/Library/LaunchDaemons`.

   The file, however, like all plists on iOS, is in binary form. Copy the file to `/tmp` and use `plutil -convert xml1` to change it to the more readable XML form. After it is in XML, just edit it so that the `ProgramArguments` key contains `–bsd_out 1`. Because the key expects an array, the arguments have to be written separately, as follows:

```
<key>ProgramArguments</key>
        <array>
               <string>/usr/sbin/syslogd</string>
               <string>-bsd_out</string>
               <string>1</string>
        </array>
```

   After this is done, convert the file back to the binary format (`plutil –convert binary1` should do the trick), and copy it back to `/System/Library/LaunchDaemons`.

3. **Restart `launchd`, and then `syslogd`.** A `kill -HUP 1` will take care of `launchd`, and — after you find the process ID of `syslogd` — a kill `-TERM` on its PID will cause `launchd` to restart it, this time with the `-bsd_out 1` argument, as desired. A `ps aux` will verify that is indeed the case, as will the log files in `/var/log`.

Apple Events and AppleScript

One of OS X's oft-overlooked, though truly powerful features, lies in its scripting capabilities. AppleScript has its origins traced back to OS 7(!) and a language called HyperCard. It has since evolved considerably, and become the all-powerful mechanism behind the `osascript(1)` command and the friendly (but neglected) Automator.

In a somewhat similar way to how iPhone's SIRI recognizes English patterns, AppleScript allows a semi-natural language interface to scriptable applications. The "semi" is because commands must follow a given grammar. If the grammar is adhered to, however, it allows for a large range of freedom. The OS X built-in applications can be almost fully automated. For those wary of scripts, the Automator provides a feature-oriented drag-and-drop GUI, as shown in Figure 3-1. Note the rich "Library" composed of actions and definitions in `/System/Library/Automator`.

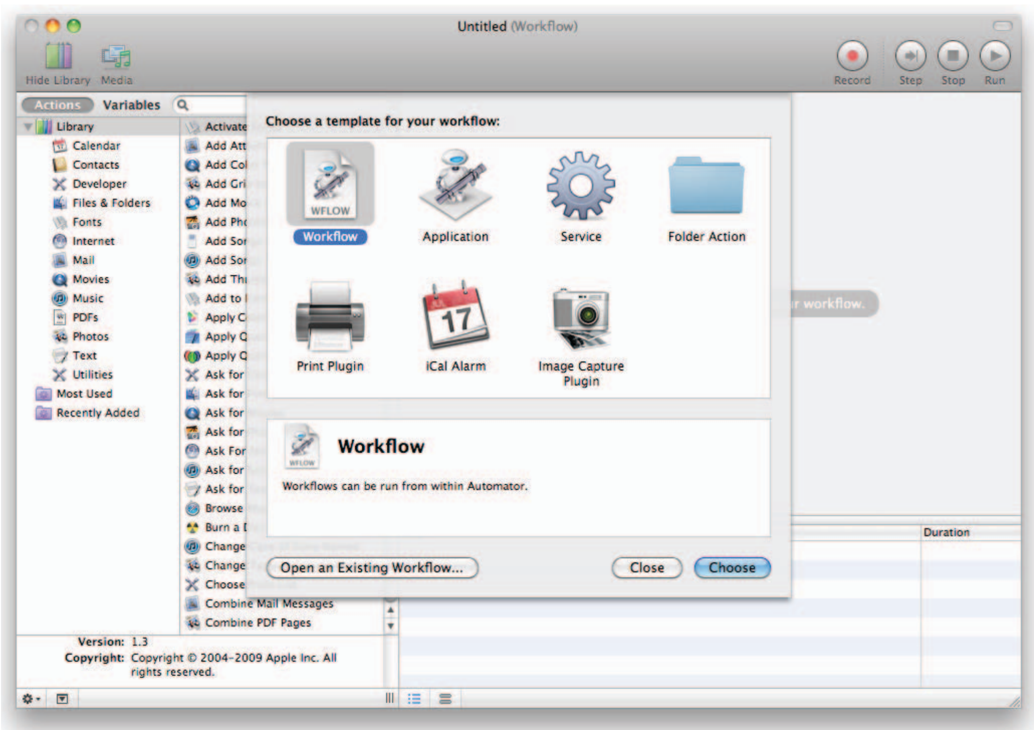


FIGURE 3-1: Automator and its built-in templates.

The mechanism allowing AppleScript's magic is called AppleEvents. AppleScript can be extended to remote hosts, either via the (now obsolete) AppleTalk protocol, or over TCP/IP. In the latter case, the protocol is known as "eppc," and is a proprietary, undocumented protocol that uses TCP port 3031. The remote functionality is only enabled if Remote Apple Events are enabled from the Sharing applet of System Preferences. This tells `launchd(8)` to listen on the eppc port, and — when requests are received — start the AppleEvents server, `AEServer` (found in the `Support/` directory of the `AE.framework`, which is internal to CoreServices). `launchd(8)` is responsible for starting many on-demand services from their respective plist files in `/System/Library/LaunchDaemons`. `AEServer`'s is `com.apple.eppc.plist`.

Though covering it is far beyond the scope of this book, AppleScript is a great mechanism for automating tasks. Outside Apple's own reference, two books devoted to the topic can be found elsewhere.[4,5] The simple experiment described next, however, shows you the flurry of events that occurs behind the scenes when you run AppleScript or Automator.

Experiment: Viewing Apple Events

You can easily see what goes on in the Apple Events plane via two simple environment variables — `AEDebugSends` and `AEDebugReceives`. Then, using `osascript` (or, in some cases, Automator), will generate plenty of output. In Output 3-5, note the debug info only pertains to events sent or received by the shell and its children, not events occurring elsewhere in the system.

OUTPUT 3-5: Output of AppleEvents driving Safari application launch

```
morpheus@ergo(/)$ export AEDebugSends=1 AEDebugReceives=1
morpheus@ergo(/)$ osascript -e 'tell app "Safari" to activate'
{ 1 } 'aevt':  ascr/gdte (i386){
          return id: -16316 (0xffffc044)
     transaction id: 0 (0x0)
  interaction level: 64 (0x40)
     reply required: 1 (0x1)
             remote: 0 (0x0)
      for recording: 0 (0x0)
         reply port: 0 (0x0)
  target:
    { 2 } 'psn ':  8 bytes {
      { 0x0, 0x5af5af } (Safari)
    }
  fEventSourcePSN: { 0x1,0xc044 } ()
  optional attributes:
    < empty record >
  event data:
    { 1 } 'aevt':  - 1 items {
      key '----' -
        { 1 } 'long':  4 bytes {
          0 (0x0)
        }
    }
}
```

continues

OUTPUT 3-5 *(continued)*

```
{ 1 } 'aevt':  aevt/ansr (****){
          return id: -16316 (0xffffc044)
     transaction id: 0 (0x0)
  interaction level: 112 (0x70)
     reply required: 0 (0x0)
             remote: 0 (0x0)
      for recording: 0 (0x0)
         reply port: 0 (0x0)
  target:
    { 1 } 'psn ':  8 bytes {
      { 0x1, 0xc044 } (<process { 1, 49220 } not found>)
      )
    }
  fEventSourcePSN: { 0x0,0x5af5af } (Safari)
  optional attributes:
    < empty record >
  event data:
    { 1 } 'aevt':  - 1 items {
      key '----' -
        { 1 } 'aete':  9952 bytes {
           000: 0100 0000  0000 0500  0a54 7970  6520 4e61     ........-Type Na
           001: 6d65 731a  4f74 6865  7220 636c  6173 7365     mes.Other classe
           ...:  // etc, etc, etc…
```

FSEvents

All modern operating systems offer their developers APIs for file system notification. These enable quick and easy response by user programs for additions, modifications, and deletions of files. Thus, Windows has its `MJ_DIRECTORY_CONTROL`, Linux has `inotify`. Mac OS X and iOS (as of version 5.0) both offer FSEvents.

FSEvents is conceptually somewhat similar to Linux's `inotify` — in both, a process (or thread) obtains a file descriptor, and attempts to `read(2)` from it. The system call blocks until some event occurs — at which time the received buffer contains the event details by which the program can tell what happened, and then act accordingly (for example, display a new icon in the file browser).

FSEvents is, however, a tad more complicated (and, some would say, more elegant) than inotify. In it, the process proceeds as follows:

- The process (or thread) requests to get a handle to the FSEvents mechanism. This is `/dev/fsevents`, a pseudo-device.
- The requestor then issues a special `ioctl(2)`, `FSEVENTS_CLONE`. This `ioctl` enables the specific filtering of events so that only events of interest — specific operations on particular files — are delivered. Table 3-7 lists the types that are currently supported. Supporting these events is possible because FSEvents is plugged into the kernel's file system-handling logic (VFS, the Virtual File system Switch — see Chapter 15 for more on that topic). Each and every supported event will add a pending notification to the cloned file descriptor.

TABLE 3-7: FSEvent Types

FSEVENT CONSTANT	INDICATES
`FSE_CREATE_FILE`	File creation.
`FSE_DELETE`	File/directory has been removed.
`FSE_STAT_CHANGED`	`stat(2)` of file or directory has been changed.
`FSE_RENAME`	File/directory has been renamed.
`FSE_CONTENT_MODIFIED`	File has been modified.
`FSE_EXCHANGE`	The `exchangedata(2)` system call.
`FSE_FINDER_INFO_CHANGED`	File finder information attributes have changed.
`FSE_CREATE_DIR`	A new directory has been created.
`FSE_CHOWN`	File/directory ownership change.
`FSE_XATTR_MODIFIED`	File/directory extended attributes have been modified.
`FSE_XATTR_REMOVED`	File/directory extended attributes have been removed.

- Using `ioctl(2)`, the watcher can modify the exact event details requested in the notification. The control codes defined include `FSEVENTS_WANT_COMPACT_EVENTS` (to get less information), `FSEVENTS_WANT_EXTENDED_INFO` (to get even more information), and `NEW_FSEVENTS_DEVICE_FILTER` (to filter on devices the watcher is not interested in watching).
- The requestor (also called the "watcher") then enters a `read(2)` loop. Each time the system call returns, it populates the user-provided buffer with an array of event records. The read can be tricky, because a single operation might return multiple records of variable size. If events have been dropped (due to kernel buffers being exceeded), a special event (`FSE_EVENTS_DROPPED`) will be added to the event records.

If you check Apple's documentation, the manual pages, or the include files, your search will come out quite empty handed. `<sys/fsevents.h>` did make an early cameo appearance when FSEvents was introduced, but has since been thinned and deprecated (and might disappear in Mountain Lion altogether). This is because, even though the API remains public, it only has some three official users:

- `coreservicesd`: This is an Apple internal daemon supporting aspects of Core Services, such as launch services and others.
- `mds`: The Spotlight server. Spotlight is a "heavy" user of FSEvents, relying on notifications to find and index new files.
- `fseventsd`: A generic user space daemon that is buried inside the CoreServices framework (alongside `coreservicesd`). FSEventsd can be told to not log events by a "`no_log`" file in the `.fseventsd` directory, which is created on the root of every volume.

Both Objective-C and C applications can use the CoreServices Framework (Carbon) APIs of `FSEventStreamCreate` and friends. This framework is a thin layer on top of the actual mechanism, which allows integration of the “real” API with the RunLoop model, events, and callbacks. In essence, this involves converting the blocking, synchronous model to an asynchronous, event-driven one. Apple documents this well.[6] The rest of this section, therefore, concentrates on the lower-level APIs.

Experiment: A File System Event Monitor

Listing 3-5 shows a barebones FSEvents client that will listen on a particular path (given as an argument) and display events occurring on the path. Though functionally similar to `fs_usage(1)`, the latter does not use FSEvents (it uses the little-documented `kdebug` API, described in Chapter 5, “Process Tracing and Debugging”).

LISTING 3-5: A bare bones FSEvents-based file monitor

```
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <sys/ioctl.h>     // for _IOW, a macro required by FSEVENTS_CLONE
#include <sys/types.h>     // for uint32_t and friends, on which fsevents.h relies
#include <sys/fsevents.h>

// The struct definitions are taken from bsd/vfs/vfs_events.c
// since they are no long public in <sys/fsevents.h>

#pragma pack(1)
typedef struct kfs_event_a {
  uint16_t type;
  uint16_t refcount;
  pid_t    pid;
} kfs_event_a;

typedef struct kfs_event_arg {
  uint16_t type;
  uint16_t pathlen;
  char data[0];
} kfs_event_arg;

#pragma pack()

int print_event (void *buf, int off)
{
   // Simple function to print event - currently a simple printf of "event!".
   // The reader is encouraged to improve this, as an exercise.
   // This book's website has a much better (and longer) implementation
   printf("Event!\n");
   return (off);

}

void main (int argc, char **argv)
{

        int fsed, cloned_fsed;
        int i;
```

```
int rc;
fsevent_clone_args  clone_args;
char buf[BUFSIZE];

fsed = open ("/dev/fsevents", O_RDONLY);

int8_t  events[FSE_MAX_EVENTS];


if (fsed < 0)
{
        perror ("open"); exit(1);
}


// Prepare event mask list. In our simple example, we want everything
// (i.e. all events, so we say "FSE_REPORT" all). Otherwise, we
// would have to specifically toggle FSE_IGNORE for each:
//
// e.g.
//       events[FSE_XATTR_MODIFIED] = FSE_IGNORE;
//       events[FSE_XATTR_REMOVED]  = FSE_IGNORE;
// etc..

for (i = 0; i < FSE_MAX_EVENTS; i++)
{
       events[i] = FSE_REPORT;
}

memset(&clone_args, '\0', sizeof(clone_args));
clone_args.fd = &cloned_fsed; // This is the descriptor we get back
clone_args.event_queue_depth = 10;
clone_args.event_list = events;
clone_args.num_events = FSE_MAX_EVENTS;

// Request our own fsevents handle, cloned

rc = ioctl (fsed, FSEVENTS_CLONE, &clone_args);

if (rc < 0) { perror ("ioctl"); exit(2);}
printf ("So far, so good!\n");
close (fsed);

while ((rc = read (cloned_fsed, buf, BUFSIZE)) > 0)
{
        // rc returns the count of bytes for one or more events:
        int offInBuf = 0;

        while (offInBuf < rc) {

           struct kfs_event_a *fse = (struct kfs_event_a *)(buf + offInBuf);
           struct kfs_event_arg *fse_arg;

           struct fse_info *fse_inf;

        if (offInBuf) { printf ("Next event: %d\n", offInBuf);};
```

continues

LISTING 3-5 *(continued)*

```
                offInBuf += print_event(buf,offInBuf); // defined elsewhere


                } // end while offInBuf..
                if (rc != offInBuf)
                   { fprintf (stderr, "***Warning: Some events may be lost\n"); }


        } // end while rc = ..

} // end main
```

If you compile this example on either OS X or iOS 5 and, in another terminal, make some file modifications (for example, by creating a temporary file), you should see printouts of file system event occurrences. In fact, even if you don't do anything, the system periodically creates and deletes files, and you will be able to receive notifications.

Note this fairly rudimentary example can be improved on in many ways, not the least of which is display event details. Singh's book has an "fslogger" application (which no longer compiles on Snow Leopard due to missing dependencies). One nifty GUI-based app is FernLightning's "fseventer," [7] which is conceptually very similar to this example, but whose interface is far richer (yet has not been updated in recent years). The book's companion website offers a tool, filemon, which improves this example and can prove quite useful, especially on iOS 5. Output 3-6 shows a sample output of this tool.

OUTPUT 3-6: Output of an fsevents-based file monitoring tool

```
File /private/tmp/xxxxx has been modified
     PID: 174 (/tmp/a)
     INODE: 7219206 DEV 40007 UID 501 (morpheus) GID 501
File /Users/morpheus/Library/PubSub/Database/Database.sqlite3-journal has been created
     PID: 43397 (mysqld)
     INODE: 7219232 DEV 40007 UID 501 (morpheus) GID 501
File /Users/morpheus/Library/PubSub/Database/Database.sqlite3-journal has been modified
     PID: 43397 (mysqld)
     INODE: 7219232 DEV 40007 UID 501 (morpheus) GID 501
File /Users/morpheus/Library/PubSub/Database/Database.sqlite3-journal has been deleted
Type: 1 (Deleted ) refcount 0  PID: 43397
     PID: 43397 (mysqld)
     INODE: 7219232 DEV 40007 UID 501 (morpheus) GID 501
...
```

Notifications

OS X provides a systemwide notification mechanism. This is a form of distributed IPC, by means of which processes can broadcast or listen on events. The heart of this mechanism is the `notifyd(8)` daemon, which is started at boot time: this is the Darwin notification server. An additional daemon, `distnoted(8)`, functions as the distributed notification server. Applications may use the `notify(3)` API to pass messages to and from the daemons. The messages are for given names, and Apple recommends the use of reverse DNS namespaces here, as well (for example, *com.myCompany.myNotification*) to avoid any collisions.

The API is very versatile and allows requesting notifications by one of several methods. The well-documented `<notify.h>` lists functions to enable the notifications over UNIX signals, Mach ports, and file descriptors. Clients may also manually suspend or resume notifications. The `notifyd(8)` handles most notifications, by default using Mach messages and registering the Mach port of `com.apple.system.notification_center`.

A command line utility, `notifyutil(1)`, is available for debugging. Using this utility, you can wait for (`-w`) and post (`-p`) notifications on arbitrary keys.

An interesting feature of `notifyd(8)` is that it is one of the scant few daemons to use Apple's file-port API. This enables file descriptors to be passed over Mach messages.

Additional APIs of interest

Additional Apple-specific APIs worth noting, but described elsewhere in this book include:

- **Grand Central Dispatch (Chapter 4):** A system framework for parallelization using work queue extensions built on top of pthread APIs.
- **The Launch Daemon (Chapter 7):** Fusing together many of UN*X system daemons (such as init, inetd, at, crond and others), along with the Mach bootstrap server.
- **XPC (Chapter 7):** A framework for advanced IPC, enabling privilege separation between processes
- **kdebug (Chapter 5):** A little-known yet largely-useful facility for kernel-level tracing of system calls and Mach traps.
- **System sockets (Chapter 17):** Sockets in the `PF_SYSTEM` namespace, which allow communication with kernel mode components
- **Mach APIs (Chapters 9, 10, and 11):** Direct interfaces to the Mach core of XNU, which supply functionality matching the higher level BSD/POSIX interfaces, but in some cases well exceeding them.
- **The IOKit APIs (Chapter 19):** APIs to communicate with device drivers, providing a plethora of diagnostics information as well as powerful capabilities for controlling drivers from user mode.

OS X AND IOS SECURITY MECHANISMS

Viruses and malware are rare on OS X, which is something Apple has kept boasting for many years as an advantage for Mac, in their commercials of "Mac versus PC." This, however, is largely due to the Windows monoculture. Put yourself in the role of Malware developer, concocting your scheme for the next devious bot. Would you invest time and effort in attacking over 90% of the world, or under 5%?

Indeed, OS X (and, to an extent, Linux) remain healthy, in part, simply because they do not attract much attention from malware "providers" (another reason is that UN*X has always adhered to the principle of least privilege, in this case not allowing the user root access by default). This, however, is changing, as with OS X's slow but steady increase in market share, so increases its allure for malware. The latest Mac virus, "Flashback" (so called because it is a Trojan masquerading as an Adobe Flash update) infected some 600,000 users in the United States alone. Certain industry experts were quick to pillory Apple for its hubris, chiding their security mechanisms as being woefully inefficient and backdated.

In actuality, however, Apple's application security is light years (if not parsecs) ahead of its peers. Windows' User Account Control (UAC) has been long present in OS X. iOS's hardening makes Android seem riddled in comparison. Nearly all so called "viruses" which do exist in Mac are actually Trojans — which rely on the cooperation (and often utter gullibility) of the unwitting user. Apple is well aware of that, and is determined to combat malware. The arsenal with which to do that has been around since Leopard, and Apple is investing ongoing efforts to upgrade it in OS X and, even more so in iOS.

Code Signing

Before software can be secured, its origin must be *authenticated*. If an app is downloaded from some random site on the Internet, there is a significant risk it is actually malware. The risk is greatly mitigated, however, if the software's origin can be verifiably determined, and it can further be assured that it has not been modified in transit.

Code signing provides the mechanism to do just that. Using the same X.509v3 certificates that SSL uses to establish the identity of websites (by signing their public key with the private key of the issuer), Apple encourages developers to sign their applications and authenticate their identity. Since the crux of a digital signature is that the signer's public key must be a priori known to the verifier, Apple embeds its certificates into both OS X and iOS's keychains (much like Microsoft does in Windows), and is effectively the only root authority. You can easily verify this using the `security(1)` utility, which (among its many other functions) can dump the system keychains, as shown in Output 3-7:

OUTPUT 3-7: Using security(1) to display Apple's built-in certificates on OS X

```
morpheus@Minion (~)$ security -i    # Interactive mode
security> list-keychains
  "/Users/morpheus/Library/Keychains/login.keychain" # User's passwords, etc
  "/Library/Keychains/System.keychain"               # Wi-Fi password,s and certificates

                                    # Non-Interactive mode

morpheus@Minion (~)$ security dump-keychain /Library/Keychains/System.keychain |
                      grep labl                       # Show only labels
    "labl"<blob>="com.apple.systemdefault"
    "labl"<blob>="com.apple.kerberos.kdc"
    "labl"<blob>="Apple Code Signing Certification Authority"
    "labl"<blob>="Software Signing"
    "labl"<blob>="Apple Worldwide Developer Relations Certification Authority"
```

Apple has developed a special language to define code signing requirements, which may be displayed with the `csreq(1)` command. Apple also provides the `codesign(1)` command to allow developers to sign their apps (as well as verify/display existing signatures), but `codesign(1)` won't sign anything without a valid, trusted certificate, which developers can only obtain by registering with Apple's Developer Program. Apple's Code Signing Guide[8] covers the code signing process in depth, with Technical Note 2250[9] discussing iOS.

Whereas in OS X code signing is optional, in iOS it is very much mandatory. If, by some miracle, an unsigned application makes its way to the file system, it will be killed by the kernel upon any attempted execution. This is what makes jailbreakers' life so hard: The system simply refuses to run unsigned code, and so the only way in is by exploiting vulnerabilities in existing, signed applications (and later the kernel itself). Jailbreakers must therefore seek faults in iOS's system apps and libraries (e.g. MobileSafari, Racoon, and others). Alternatively, they may seek faults in the code-signing mechanism itself, as was done by renowned security researcher Charlie Miller in iOS 5.0.[10] Disclosing this to Apple, however, proved a Pyrrhic victory. Apple quickly patched the vulnerability in 5.0.1, and another future jailbreak door slammed shut forever. Mr. Miller himself was controversially banned from the iOS Developer Program.

Code-signed applications may still be malicious. Any applications that violate the terms of service, however, would quickly lead to their developer becoming a persona non grata at Apple, banned from the Mac/iOS App Stores (q.v. Mr. Miller). Since registering with Apple involves disclosing personal details, these malicious developers could also be the target of a lawsuit. This is why you won't find any apps in iOS's App Store attempting to spawn `/bin/bash` or mimic its functionality. Nobody wants to get on Apple's bad side.

Compartmentalization (Sandboxing)

Originally considered a vanguard, nice-to-have feature, *compartmentalization* is becoming an integral part of the Apple landscape. The idea is a simple, yet principal tenet of application security: Untrusted applications must run in a compartment, effectively a quarantined environment wherein all operations are subject to restriction. Formerly known in Leopard as *seatbelt*, the mechanism has since been renamed *sandbox*, and has been greatly improved in Lion, touted as one of its stronger suits. A thorough discussion of the sandbox mechanism (as it was implemented in Snow Leopard) can be found in Dionysus Blazakis's Black Hat DC 2011 presentation[11], though the sandbox has undergone significant improvements since.

iOS — the Sandbox as a jail

In iOS, the sandbox has been integrated tightly since inception, and has been enhanced further to create the "jail" which the "jailbreakers" struggle so hard to break. The limitations in an App's "jail" include, but are not limited to:

- Inability to break out of the app's directory. The app effectively sees its own directory (`/var/mobile/Applications/<app-GUID>`) as the root, similar to the `chroot(2)` system call. As a corollary, the app has no knowledge of any other installed apps, and cannot access system files.
- Inability to access any other process on the system, even if that process is owned by the same UID. The app effectively sees itself as the only process executing on the system.
- Inability to directly use any of the hardware devices (camera, GPS, and others) without going through Apple's Frameworks (which, in turn, can impose limitations, such as the familiar user prompts).
- Inability to dynamically generate code. The low-level implementations of the `mmap(2)` and `mprotect(2)` system calls (Mach's `vm_map_enter` and `vm_map_protect`, respectively, as discussed in Chapter 13) are intentionally modified to circumvent any attempts to make writable memory pages also executable. This is discussed in Chapter 11.
- Inability to perform any operations but a subset of the ones allowed for the user mobile. Root permissions for an app (aside for Apple's own) are unheard of.

Entitlements (discussed later) can release some well-behaving apps from solitary confinement, and some of Apple's own applications do possess root privileges.

Voluntary Imprisonment

Execution in a sandbox is still voluntary (at least, in OS X). A process must willingly call `sandbox_init(3)` to enter a sandbox, with one of the predefined profiles shown in Table 3-8. (This, however, can also be accomplished by a thin wrapper, which is exactly what the command line `sandbox-exec(1)` is used for, along with the `-n` switch and a profile name).

TABLE 3-8: Predefined Sandbox Profiles

KSBXPROFILE CONSTANT	PROFILE NAME (FOR `sandbox-exec -n`)	PROHIBITS
`NoInternet`	`no-internet`	`AF_INET/AF_INET6` sockets
`NoNetwork`	`no-network`	`socket(2)` call
`NoWrite`	`no-write`	File system write operations
`NoWriteExceptTemporary`	`no-write-except-temporary`	File system write operations except temporary directories
`PureComputation`	`pure-computation`	Most system calls

The `sandbox_init(3)` function in turn, calls the `mac_execve` system call (#380), and the profile corresponds to a MAC label, as discussed earlier in this chapter. The profile imposes a set of predefined restrictions on the process, and any attempt to bypass these restrictions results in an error at the system-call level (usually a return code of `-EPERM`). The seatbelt may well have been renamed to "quicksand," instead, because once a sandbox is entered, there is no way out. The benefit of a tight sandbox is that a user can run an untrusted application in a sandbox with no fear of hidden malware succeeding in doing anything insidious (or anything at all, really), outside the confines of the defined profile. The predefined profiles serve only as a point of departure, and profiles can be created on a per-application basis.

Apple has recently announced a requirement for all Mac Store apps to be sandboxed, so the "voluntary" nature of sandboxing will soon become "mandatory," by the time this book goes to print. Because it still requires a library call in the sandboxed program, averting the sandbox remains a trivial manner — by either hooking `sandbox_init(3)` prior to executing the process[12] or not calling it at all. Neither or these are really a weakness, however. From Apple's perspective, the user likely has no incentive to do the former, because the sandbox only serves to enhance his or her security. The developer might very well be tempted to do the latter, yet Apple's review process will likely ensure that all submitted apps willingly accept the shackles in return for a much-coveted spot in the Mac store.

Controlling the Sandbox

In addition to the built-in profiles, it is possible to specify custom profiles in `.sb` files. These files are written in the sandbox's Scheme-like dialect. The files specify which actions to be allowed or denied, and are compiled at load-time by `libSandbox.dylib`, which contains an embedded TinySCHEME library.

You can find plenty of examples in /usr/share/sandbox and /System/Library/Sandbox/Profiles (or by searching for *.sb files). A full explanation of the syntax is beyond the scope of this book Listing 3-6, however, serves to demonstrate the key aspects of the syntax by annotating a sample profile.

LISTING 3-6: A sample custom sandbox profile, annotated

```
(version 1)
(deny default)              ; deny by default – least privilege
(import "system.sb")        ; include another profile as a point of departure


(allow file-read*)          ; Allow all file read operations
(allow network-outbound)    ; Allow outgoing network connections
(allow sysctl-read)
(allow system-fsctl)
(allow distributed-notification-post)

(allow appleevent-send (appleevent-destination "com.apple.systempreferences"))

(allow ipc-posix-shm system-audit system-sched mach-task-name process-fork process-exec)

(allow iokit-open           ; Allow the following I/O Kit calls
       (iokit-connection "IOAccelerator")
       (iokit-user-client-class "RootDomainUserClient")
       (iokit-user-client-class "IOAccelerationUserClient")
       (iokit-user-client-class "IOHIDParamUserClient")
       (iokit-user-client-class "IOFramebufferSharedUserClient")
       (iokit-user-client-class "AppleGraphicsControlClient")
       (iokit-user-client-class "AGPMClient"))
)

allow file-write*           ; Allow write operations, but only to the following path:
       (subpath "/private/tmp")
       (subpath (param "_USER_TEMP"))
)

(allow mach-lookup          ; Allow access to the following Mach services
       (global-name "com.apple.CoreServices.coreservicesd")
)
```

If a trace directive is used, the user-mode daemon sandboxd(8) will generate rules, allowing the operations requested by the sandboxed application. A tool called sandbox-simplify(1) may then be used in order to coalesce rules, and simplify the generated profile.

Entitlements: Making the Sandbox Tighter Still

The sandbox mechanism is undoubtedly a strong one, and far ahead of similar mechanisms in other operating systems. It is not, however, infallible. The "black list" approach of blocking known dangerous operations is only as effective as the list is restrictive. As an example, consider that in November 2011 researchers from Core Labs demonstrated that, while Lion's kSBXProfileNoNetwork indeed restricts network access, it does not restrict AppleEvents.[13] What follows is that a malicious app can trigger AppleScript and connect to the network via a non-sandboxed proxy process.

The sandbox, therefore, has been revamped in Lion, and will likely be improved still in Mountain Lion, where it has been rebranded as "GateKeeper" and is a combination of an already-existing mechanism: HFS+'s quarantine, with a "white list" approach (that is, disallowing all but that which is known to be safe) that aims to deprecate the "black list" of the current sandboxing mechanism. Specifically, applications downloaded will have the "quarantine" extended attribute set, which is responsible for the familiar "...is an application downloaded from the Internet" warning box, as before. This time, though, the application's code signature will be checked for the publisher's identity as well as any potential tampering and known reported malware.

Containers in Lion

Lion introduces a new command line, `asctl(1)`, which enables finer tuning of the sandbox mechanism. This utility enables you to launch applications and trace their sandbox activity, building a profile according to the application requirements. It also enables to establish a "container" for an application, especially those from the Mac Store. The containers are per-application folders stored in the `Library/Containers` directory. This is shown in the next experiment.

It is more than likely that Mac Store applications will, sooner or later, only be allowed to execute according to specific *entitlements*, as is already the case in iOS. Entitlements are very similar in concept to the declarative permission mechanism used in .NET and Java (which also forms the basis for Android's Dalvik security). The entitlements are really nothing more than property lists. In Lion (as the following experiment illustrates) the entitlements are part of the container's plist.

Experiment: Viewing Application Containers in Lion

If you have downloaded an app from the Mac Store, you can see that a container for it has likely been created in your `Library/Containers/` directory. Even if you have not, two apps already thus contained are Apple's own Preview and TextEdit, as shown in Output 3-8:

OUTPUT 3-8: Viewing the container of TextEdit, one of Apple's applications

```
morpheus@Minion (~)$ asctl container path TextEdit
~/Library/Containers/com.apple.TextEdit
morpheus@Minion (~)$ cd Library/Containers
morpheus@Minion (~/Library/Containers)$ ls
com.apple.Preview       com.apple.TextEdit
morpheus@Minion (~/Library/Containers)$ cd com.apple.TextEdit
morpheus@Minion (~/…Edit)$ find .
./Container.plist
./Data
./Data/.CFUserTextEncoding
./Data/Desktop
./Data/Documents
./Data/Downloads
./Data/Library
...
./Data/Library/Preferences
...
./Data/Library/Saved Application State
./Data/Library/Saved Application State
```

```
./Data/Library/Saved Application State/com.apple.TextEdit.savedState
./Data/Library/Saved Application State/com.apple.TextEdit.savedState/data.data
./Data/Library/Saved Application State/com.apple.TextEdit.savedState/window_1.data
./Data/Library/Saved Application State/com.apple.TextEdit.savedState/windows.plist
./Data/Library/Sounds
./Data/Library/Spelling
./Data/Movies
./Data/Music
./Data/Pictures
```

The `Data/` folder of the container forms a jail for the app, in the same way that iOS apps are limited to their own directory. If global files are necessary for the application to function, it is a simple matter to create hard or soft links for them. The various preferences files, for example, are symbolic links, and the files in `Saved Application State/` (which back Lion's Resume feature for apps) are hard links to files in `~/Library/Saved Application State`.

The key file in any container is the `Container.plist`, This is a property list file, though in binary format. Using `plutil(1)` to convert it to XML will reveal its contents, as shown in Output 3-9:

OUTPUT 3-9: Displaying the container.plist of TextEdit

```
morpheus@Minion (~/Library/Containers)$ cp com.apple.TextEdit/Container.plist /tmp
morpheus@Minion (~/Library/Containers)$ cd /tmp
morpheus@Minion (/tmp)$ plutil -convert xml1 Container.plist
morpheus@Minion (/tmp)$ more !$
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>Identity</key>
        <array>
                <data>
                +t4MAAAAADAAAAABAAAABgAAAAIAAAASY29tLmFwcGxlLlRleHRFZGl0AAAA
                AAAD
                </data>
        </array>
        <key>SandboxProfileData</key>
        <data>
        AAD5AAwA9wD2APIA9wD3APcA9wDxAPEA8ADkAPEAjgCMAPgAiwDxAPEAfwB/AHsAfwB/
        AH8AfwB/AH8AfwB/AHoAeQD3AHgA9wD3AGsAaQD3APcA9wD4APcA9wD3APcA9wD3APgA
         ...  Base64 encoded compiled profile data ...
        AAACAAAALwAAAC8=
        </data>
        <key>SandboxProfileDataValidationInfo</key>
        <dict>
                <key>SandboxProfileDataValidationEntitlementsKey</key>
                <dict>
                        <key>com.apple.security.app-protection</key>
                        <true/>
                        <key>com.apple.security.app-sandbox</key>
                        <true/>
```

continues

OUTPUT 3-9 *(continued)*

```
                <key>com.apple.security.documents.user-selected.read-write</key>
                <true/>
                <key>com.apple.security.files.user-selected.read-write</key>
                <true/>
                <key>com.apple.security.print</key>
                <true/>
        </dict>
        <key>SandboxProfileDataValidationParametersKey</key>
        <dict>
                <key>_HOME</key>
                <string>/Users/morpheus</string>
                <key>_USER</key>
                <string>morpheus</string>
                <key>application_bundle</key>
                <string>/Applications/TextEdit.app</string>
                <key>application_bundle_id</key>
                <string>com.apple.TextEdit</string>
                  ...
        </dict>
        <key>SandboxProfileDataValidationSnippetDictionariesKey</key>
        <array>
               <dict>
                       <key>AppSandboxProfileSnippetModificationDateKey</key>
                       <date>2012-02-06T15:50:18Z</date>
                       <key>AppSandboxProfileSnippetPathKey</key>
                <string>/System/Library/Sandbox/Profiles/application.sb</string>
               </dict>
        </array>
        <key>SandboxProfileDataValidationVersionKey</key>
        <integer>1</integer>
</dict>
<key>Version</key>
<integer>24</integer>
</dict>
</plist>
```

The property list shown above has been edited for readability. It contains two key entries:

- **`SandboxProfileData`**: The compiled profile data. Since the output of the compilation is binary, the data is encoded as Base64.
- **`SandboxProfileDataValidationEntitlementsKey`**: Specifying a dictionary of entitlements this application has been granted. Apple currently lists about 30 entitlements, but this list is only likely to grow as the sandbox containers are adopted by more developers.

Mountain Lion's version of the `asctl(1)` command contains a `diagnose` subcommand, which can be used to trace the sandbox mechanism. This functionality wraps other diagnostic commands — `/usr/libexec/AppSandBox/container_check.rb` (a Ruby script), and `codesign(1)` with the `--display` and `--verify` arguments. Although Lion does not contain the subcommand, these commands may be invoked directly, as shown in Output 3-10:

OUTPUT 3-10: Using codesign(1) --display directly on TextEdit:

```
morpheus@Minion (~)$ codesign --display --verbose=99 --entitlements=:-        \
/Applications/TextEdit.app
Executable=/Applications/TextEdit.app/Contents/MacOS/TextEdit
Identifier=com.apple.TextEdit
Format=bundle with Mach-O universal (i386 x86_64)
CodeDirectory v=20100 size=987 flags=0x0(none) hashes=41+5 location=embedded
Hash type=sha1 size=20
CDHash=7b9b2669bddfaf01291478baafd93a72c61eee99
Signature size=4064
Authority=Software Signing
Authority=Apple Code Signing Certification Authority
Authority=Apple Root CA
Info.plist entries=30
Sealed Resources rules=11 files=10

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>com.apple.security.app-sandbox</key>
        <true/>
        <key>com.apple.security.files.user-selected.read-write</key>
        <true/>
        <key>com.apple.security.print</key>
        <true/>
        <key>com.apple.security.app-protection</key>
        <true/>
        <key>com.apple.security.documents.user-selected.read-write</key>
        <true/>
</dict>
</plist>
```

Entitlements in iOS

In iOS, the entitlement plists are embedded directly into the application binaries and digitally signed by Apple. Listing 3-7 shows a sample entitlement from iOS's `debugserver`, which is part of the SDK's Developer Disk Image:

LISTING 3-7: A sample entitlements.plist for iOS's debugserver

```
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
      <key>com.apple.springboard.debugapplications</key>
      <true/>
      <key>get-task-allow</key>
      <true/>
      <key>task_for_pid-allow</key>
      <true/>
```

continues

LISTING 3-7 *(continued)*

```
        <key>run-unsigned-code</key>
        <true/>
</dict>
</plist>
```

The entitlements shown in the listing are among the most powerful in iOS. The task-related ones allow low-level access to the Mach task, which is the low-level kernel primitive underlying the BSD processes. As Chapter 10 shows, obtaining a task port is equivalent to owning the task, from its virtual memory down to its last descriptor. Another important entitlement is `dynamic-codesigning`, which enables code generation on the fly (and creating `rwx` memory pages), currently known to be granted only to MobileSafari.

Apple doesn't document the iOS entitlements (and isn't likely to do so in the near future, at least those which pertain to their own system services), but fortunately the embedded plists remain unencrypted (at least, until the time of this writing). Using `cat(1)` on key iOS binaries and apps (like MobileMail, MobileSafari, MobilePhone, and others) will display, towards the end of the output, the entitlements they use. For example, consider Listing 3-8, which shows the embedded plist in MobileSafari:

LISTING 3-8: using cat(1) to display the embedded entitlement plist in MobileSafari

```
root@podicum (/)# cat -tv /Applications/MobileSafari.app/MobileSafari | tail -31 | more
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
^I<key>com.apple.coreaudio.allow-amr-decode</key>
^I<true/>
^I<key>com.apple.coremedia.allow-protected-content-playback</key>
^I<true/>
^I<key>com.apple.managedconfiguration.profiled-access</key>
^I<true/>
^I<key>com.apple.springboard.opensensitiveurl</key>
^I<true/>
^I<key>dynamic-codesigning</key>       <!-- Required for Safari's Javascript engine !-->
^I<true/>
^I<key>keychain-access-groups</key>
^I<array>
^I^I<string>com.apple.cfnetwork</string>
^I^I<string>com.apple.identities</string>
^I^I<string>com.apple.mobilesafari</string>
^I^I<string>com.apple.certificates</string>
^I</array>
^I<key>platform-application</key>
^I<true/>
^I<key>seatbelt-profiles</key>
^I<array>
^I^I<string>MobileSafari</string> <!-- Safari has its own seatbelt/sandbox profile !-->
^I</array>
^I<key>vm-pressure-level</key>
^I<true/>
</dict>
</plist>
```

iOS developers can only embed entitlements allowed by Apple as part of their developer license. The allowed entitles are themselves, embedded into the developer's own certificate. Applications uploaded to the App Store have the entitlements embedded in them, so verifying application security in this way is a trivial matter for Apple. More than likely, this will be the case going forward for OS X, though at the time of this writing, this remains an educated guess.

Enforcing the Sandbox

Behind the scenes, XNU puts a lot of effort into maintaining the sandboxed environment. Enforcement in user mode is hardly an option due to the many hooking and interposing methods possible. The BSD MAC layer (described earlier) is the mechanism by which both sandbox and entitlements work. If a policy applies for the specific process, it is the responsibility of the MAC layer to callout to any one of the policy modules (i.e. specialized kernel extensions). The main kernel extension responsible for the sandbox is `sandbox.kext`, common to both OS X and iOS. A second kernel extension unique to iOS, `AppleMobileFileIntegrity` (affectionately known as AMFI), enforces entitlements and code signing (and is a cause for ceaseless headaches to jailbreakers everywhere). As noted, the sandbox also has a dedicated daemon, `/usr/libexec/sandboxd`, which runs in user mode to provide tracing and helper services to the kernel extension, and is started on demand (as you can verify if you use `sandbox-exec(1)` to run a process). In iOS, AMFI also has its own helper daemon, `/usr/libexec/amfid`. The OS X architecture is displayed in Figure 3-2.

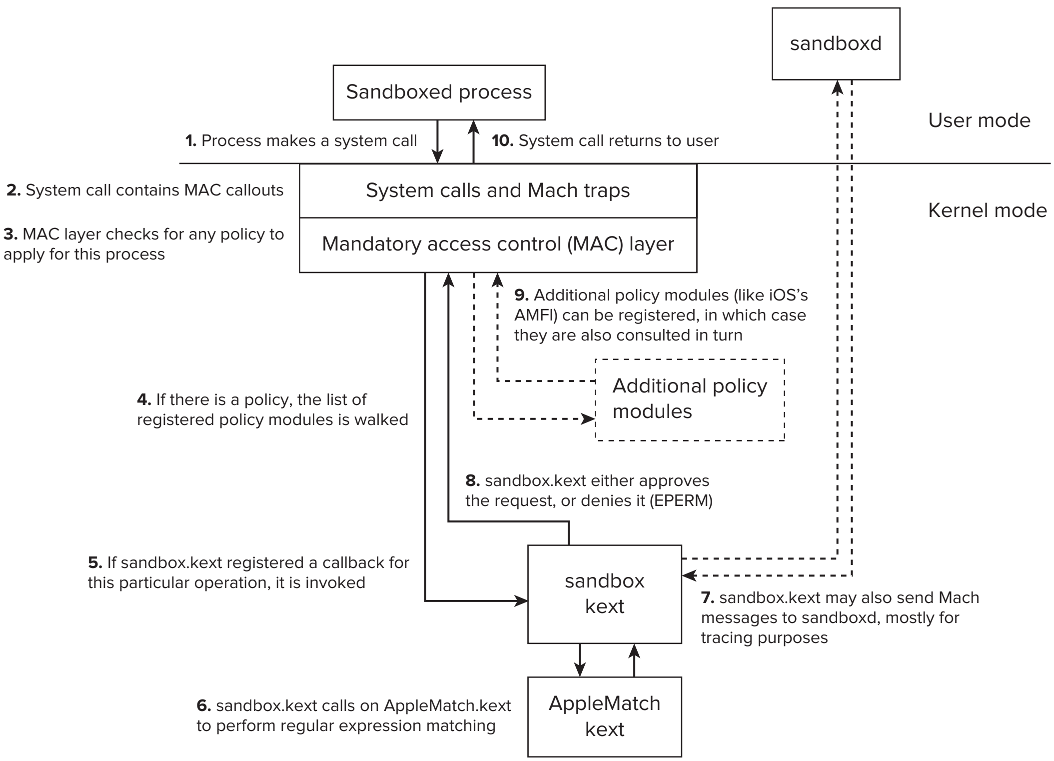


FIGURE 3-2: The sandbox architecture

Chapter 14 discusses the MAC layer in depth from the kernel perspective, and elaborates more on the enforcement of its policies, by both the sandbox and AMFI.

SUMMARY

This chapter gave a programmatic tour of the APIs that are idiosyncratic to Apple. These are specific APIs, either at the library or system-call level, providing the extra edge in OS X and iOS. From the features adopted from BSD, like `sysctl` and `kqueue`, OpenBSM and MAC, through file-system events and notifications, to the powerful and unparalleled automation of AppleEvents. This chapter finally discussed the security architecture of OS X and iOS from the user's perspective, explaining the importance of code signing, and highlighting the use the BSD MAC layer as the foundation for the Apple-proprietary technologies of sandboxing and entitlements.

The next chapters delve deeper into the system calls and libraries, and focus on process internals and using specific APIs for debugging.

REFERENCES

[1] "The TrustedBSD MAC Framework: Extensible Kernel Access Control for FreeBSD 5.0," `http://www.trustedbsd.org/trustedbsd-usenix2003freenix.pdf`

[2] Apple Developer. "Sample Code — Reachability," `http://developer.apple.com/library/ios/#samplecode/Reachability/Introduction/Intro.html`

[3] Apple Technical Note 26117. "Mac OS X Server – The System Log," `http://support.apple.com/kb/TA26117`

[4] Sanderson and Rosenthal. Learn AppleScript: The Comprehensive Guide to Scripting and Automation on Mac OS X (3E), (New York: APress, 2010).

[5] Munro, Mark Conway. *AppleScript* (*Developer Reference*), (New York: Wiley, 2010).

[6] Apple Developer. "File System Events Programming Guide," `http://developer.apple.com/library/mac/#documentation/Darwin/Conceptual/FSEvents_ProgGuide/`

[7] `http://fernlightning.com/doku.php?id=software%3afseventer%3astart`

[8] Apple Developer. "Code Signing Guide," `https://developer.apple.com/library/mac/#documentation/Security/Conceptual/CodeSigningGuide/`

[9] Technical Note 2250. "iOS Code Signing Setup, Process, and Troubleshooting," `http://developer.apple.com/library/ios/#technotes/tn2250/_index.html`

[10] "Charlie Miller Circumvents Code Signing For iOS Apps," `http://apple.slashdot.org/story/11/11/07/2029219/charlie-miller-circumvents-code-signing-for-ios-apps`

[11] Blazakis, Dionysus. "The Apple SandBox," `http://www.semantiscope.com/research/BHDC2011/`

[12] `https://github.com/axelexic/SanboxInterposed`

[13] Core Labs Security. "CORE-2011-09: Apple OS X Sandbox Predefined Profiles Bypass," `http://corelabs.coresecurity.com/index.php?module=Wiki&action=view&type=advisory&name=CORE-2011-0919`

4

Parts of the Process: Mach-O, Process, and Thread Internals

Operating systems are designed as a platform, on top of which applications may execute. Each instance of a running application constitutes a *process*. This chapter discusses the user mode perspective of processes, beginning with their executable format, through the process of loading them into memory, and the memory image which results. The chapter concludes with a discussion of virtual memory from a system-wide perspective, as it pertains to memory utilization and swapping.

A NOMENCLATURE REFRESHER

Before delving into the internals of how processes are implemented, it might be wise to spend a few minutes revising the basic terminology of processes and signals, as interpreted in UNIX. If you are well versed, feel free to skip this section.

Processes and Threads

Much like any other pre-emptive multi-tasking system, UNIX was built around the concept of a process as an instance of an executing program. Such an instance is uniquely defined by a Process ID (which will hence be referred to as a PID). Even though the same executable may be started concurrently in multiple instances, each will have a different PID. Processes may further belong to *process groups*. These are primarily used to allow the user to control more than one process — usually by sending signals (see the following section) to a group, rather than a specific process. A process may join a group by calling `setpgrp(2)`.

A process will also retain its kinship with its parent process — as kept in its Parent Process Identifier, or PPID. This is needed because, in UNIX, it is actually the norm for the parent to outlive its children. A parent can *fork* (or `posix_spawn`) children, and actually expects them to die. UNIX processes, unlike some humans, have a very distinct and clear meaning in life — to run, and then return a single integer value, which is collected by their parent process. This return value is what the process passes to the `exit(2)` system call (or, alternatively, returns from its `main()`).

Modern operating systems no longer treat processes as the basic units of operation, instead work with threads. A thread is merely a distinct register state, and more than one can exist in a given process. All threads share the virtual memory space, descriptors and handles. The process abstraction remains as a container of one or more threads. When we next discuss "processes," it is important to remember that, more often than not, these can be multi-threaded. When a process is single threaded, the terms can be used interchangeably. When multiple threads exist in the same process, however, some things — such as execution state — are applicable separately to the individual threads. Threads are discussed in more detail towards the end of this chapter.

The Process Lifecycle

The full lifecycle of a UNIX process, and therefore that of an OS X one, can be illustrated in the following figure. The `SXXX` constants refer to the ones defined in the kernel, and visible in `<sys/proc.h>` as shown in Figure 4-1:

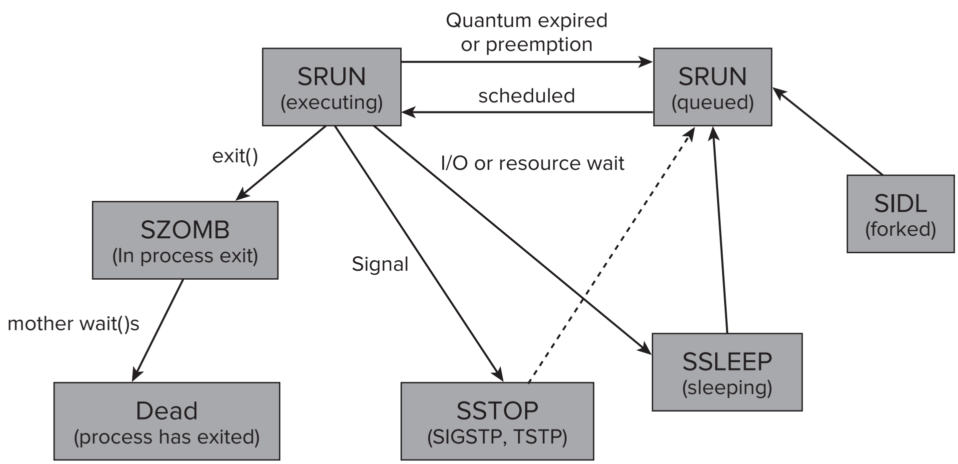


FIGURE 4-1: The process lifecycle

A process begins its life in the `SIDL` state, which represents a momentarily idle process, that has just been created by `fork`ing from its parent. In this state, the process is still defined as "initializing," and does not respond to any signals or perform any action while its memory layout is set up, and its required dependencies load. Once all is ready, the process can start executing, and does not return to `SIDL`. A process in `SIDL` is always single threaded, since threads can only be spawned later.

When a process is executing, it in the `SRUN` state. This state, however, is actually made up of two distinct states: runnable and running. A process is runnable if it is queued to run, but is not actually executing, since the CPU is busy with some other process. Only when the CPU's registers are loaded with those belong to a process (technically, to one of its threads), is a process truly in the running state. Since scheduling is volatile, however, the kernel doesn't bother to differentiate between the two distinct states. A running process may also be "kicked out" of the CPU and back to the queue if its time slice has expired, or if another process of higher priority ousts it.

A process will spend its time in the running/runnable state of `SRUN` for as long as possible, unless it waits on a resource. In this context, a "resource" is usually I/O-related (such as a file or a device). Resources also include synchronization objects (such as mutexes or locks). When a process is waiting, it makes no sense to occupy the CPU, or even consider it in the run queue. It is therefore "put to sleep" (the `SSLEEP` state). A process will sleep until the resource becomes available, at which point it will be queued again for execution — usually immediately after the current process, or sometimes even in place of it. A sleeping process can also be woken up by a signal (discussed next in this chapter).

The main advantage of multithreading is that individual thread states may diverge from one another. Thus, while one thread may be sleeping, another can be scheduled on the CPU. The threads will spend their time between the runnable/running and sleeping (or "blocked") state.

Using a special signal (`TSTOP` or `TOSTOP`), it is possible to stop a process. This "freezes" the process (i.e. simultaneously suspending all of its threads), essentially putting it into a "deep sleep" state. The only way to resume such a process is with another signal (`CONT`), which puts the process back into a runnable state, enabling once more the scheduling of any of its threads.

When a process is done, either by a return from its `main()`, or by calling `exit(2)`, it is cleared from memory, and is effectively terminated. Doing so will terminate all of its threads simultaneously. Before this can be done, however, the process must briefly spend time in the zombie state.

The Zombie State

Of all process states, the one which is least understood is the zombie state. Despite the undead context, it is a perfectly normal state, and every process usually spends an infinitesimal amount of time, just before it can rest in peace.

Recall, that the "meaning of life" for a process is to return a value to its parent. Parent processes bear no responsibility to rear and care for their children. The only thing that is requested of them, however, is to `wait(2)` for them, so their return value is collected. There is an entire family of `wait()` calls, consisting of `wait(2)`, `waitpid(2)`, `wait3(2)`, and `wait4(2)`. All expect an integer pointer amongst their parameters in which the operating system will deliver the dying child's last (double or quad) word.

In cases where the child process does outlive the parent, it is "adopted" by its great ancestor, PID 1 (in UNIX and pre-Tiger OS X, `init`, now reborn as `launchd`), which is the one process that outlives all others, persisting from boot to shutdown. Parents who outlive, yet forsake their children and move on to other things, will damn the children to be stuck in the quasi-dead state of a *zombie*. Zombies are, for all intents and purposes, quite dead. They are the empty shells of processes, which have released all resources but still cling to their PID and show up on the process list as `<defunct>` or with a status of `Z`. Zombies will rest in peace only if their parent eventually remembers to wait for them — and collect their return value — or if the parent dies, granting them rest by allowing them to be adopted, albeit briefly, by PID 1.

The code in Listing 4-1 artificially creates a zombie. After a while, when its parent exits, the zombie disappears.

LISTING 4-1: A program to artificially create a zombie

```
#include <stdio.h>
int main (int argc, char **argv)
{
    int rc = fork(); // This returns twice
    int child = 0;
    switch (rc)
     {
         case -1:
          /**
            * this only happens if the system is severely low on resources,
            * or the user's process limit (ulimit -u) has been exceeded
            */
            fprintf(stderr, "Unable to fork!\n");
           return (1);
          case 0:
            printf("I am the child! I am born\n");
             child++;
             break;
          default:
             printf ("I am the parent! Going to sleep and now wait()ing\n");
            sleep(60);
       }
       printf ("%s exiting\n", (child?"child":"parent"));
        return(0);
}
```

OUTPUT 4-1: Output of the sample program from Listing 4-1

```
Morpheus@Ergo (~)$ cc a.c -o a        # compiling the program
cc a.c -o a
Morpheus@Ergo (~)$ ./a &              # running the program in the background
[2] 3620
I am the parent! *Yawn* Going to sleep..
I am the child! I am born!
child exiting

Morpheus@Ergo (~)$ ps a              # ps "a" shows the STAT column.
  PID   TT  STAT      TIME COMMAND
  264 s000  Ss     0:00.03 login -pf morpheus
  265 s000  S      0:00.10 -bash
 3611 s000  T      0:00.03 vi a.c
 3620 s000  S      0:00.00 ./a
 3621 s000  Z      0:00.00 (a)
 3623 s000  R+     0:00.00 ps a 3601 s000  R+     0:00.00 ps a
```

pid_suspend and pid_resume

OS X (and iOS) added two new system calls in Snow Leopard for process control: `pid_suspend` and `pid_resume`. The former "freezes" a process, and the latter "thaws" it. The effect, while similar to sending the process `STOP`/`CONT` signals, is different. First, the process state remains `SSLEEP`, seemingly a normal "sleep," though in effect a much deeper one. This is because the underlying suspension is performed at a lower level (of the Mach task) rather than that of the process. Second, these calls can be used multiple times, incrementing and decrementing the process suspend count. Thus, for every call to `pid_suspend`, there needs to be a matching call to `pid_resume`. A process with a non-zero suspend count will remain suspended.

The system calls calls are private to Apple, and their prototypes are not published in header files, save for a mention of the system call numbers in `<sys/syscall.h>`. These numbers, however, must not be relied upon, as they have changed between Snow Leopard (wherein they were #430 and #431, respectively) and Lion/iOS (wherein they are #433 and #434). The previous system call numbers are now used by the `fileport` mechanism. The system calls are also largely unused in OS X, but iOS's SpringBoard makes good use of them (as some processes are suspended when the user presses the i-Device's home button).

iOS further adds a private system call, which does not exist in OS X, called `pid_shutdown_sockets` (#435). This system call enables shutting down all of a process's sockets from outside the process. The call is used exclusively by SpringBoard, likely when suspending a process.

UNIX Signals

While alive, processes usually mind their own business and execute in a sequential, sometimes parallelized sequential, manner (the latter, if using threads). They may, however, encounter *signals*, which are software interrupts indicating some exception made on their part, or an external event. OS X, like all UNIX systems, supports the concept of signals — asynchronous notifications to a program, containing no data (or, some would argue, containing a single bit of data). Signals are sent to processes by the operating system, indicating the occurrence of some condition, and this condition usually has its cause in some type of hardware fault or program exception.

There are 31 defined signals in OS X (signal 0 is supported, but unused). They are defined in `<sys/signal.h>`. The numbers are largely the same as one would expect from other UNIX systems. Table 4-1 summarizes the signals and their default behavior.

TABLE 4-1: UNIX signals in OS X, with scope and default behaviors

#	SIG	ORIGIN	MEANING	P/T	DEFAULT
1	HUP	Tty	Terminal hangup (for daemons: `reload conf`).	P	K
2	INT	Tty	Generated by terminal driver on `stty intr`.	P	K
3	QUIT	Tty	Generated by terminal driver on `stty quit`.	P	K,C
4	ILL	HW	Illegal instruction.	T	K,C
5	TRAP	HW	Debugger trap/assembly (`"int 3"`).	T	K,C

(continues)

TABLE 4-1 *(continued)*

#	SIG	ORIGIN	MEANING	P/T	DEFAULT
6	ABRT	OS	`abort()`	P	K,C
7	POLL	OS	If `_POSIX_C_SOURCE` — pollable event.	P	K,C
			Else, emulator trap.	T	K,C
8	FPE	HW	Floating point exception, or zero divide.	T	K,C
9	KILL	User, OS (rare)	The 9mm bullet. Kills, no saving throw. Usually generated by user (`kill -9`).	P	K
10	BUS	HW	Bus error.	T	K,C
11	SEGV	HW	Segmentation violation/fault — NULL dereference, or access protection or other memory.	T	K,C
12	SYS	OS	Interrupted system call.	T	K,C
13	PIPE	OS	Broken pipe (generated when P on read of a pipe is terminated).	T	K
14	ALRM	HW	Alarm.	P	K
15	TERM	OS	Termination.	P	K
16	URG	OS	Urgent condition.	P	I
17	STOP	User	Stop (suspend) process. Send by terminal on `stty stop`.	P	S
18	TSTP	Tty	Terminal stop (`stty tostop`, or full screen in bg).	P	S,T
19	CONT	User	Resume (inverse of `STOP/TSTOP`).	P	I
20	CHLD	OS	Sent to parent on child's demise.	P	I
21	TTIN	Tty	TTY driver signals pending input.	P	S,T
22	TTOU	Tty	TTY driver signals pending output.	P	S,T
23	IO	OS	Input/output.	P	I
24	XCPU	OS	`ulimit -t` exceeded.	P	K
25	XFSZ	OS	`ulimit -f` exceeded.	P	K
26	VTALRM	OS	Virtual time alarm.	P	K
27	PROF	OS	Profiling alarm.	P	K
28	WINCH	Tty	Sent on terminal window resize.	P	I
29	INFO	OS	Information.	P	I
30	USR1	User	User-defined signal 1.	P	K
31	USR2	User	User-defined signal 2.	P	K

Legend:

Origin — Signal originates from:

- **HW:** A hardware exception or fault (for example, MMU trap)
- **OS:** Operating system, somewhere in kernel code
- **Tty:** Terminal driver
- **User:** User, by using `kill(1)` command (user can also use this command to emulate all other signals)

Default — actions to take upon a signal, if no handler is registered:

- **C — SA_CORE**: Process will dump core, unless otherwise stated.
- **I — SA_IGNORE**: Signal ignored, even if no signal handler is set.
- **K — SA_KILL**: Process will be terminated unless caught.
- **S — SA_STOP**: Process will be stopped unless caught
- **T — SA_TTYSTOP**: As `SA_STOP`, but reserved for TTY.

Signals were traditionally sent to processes, although POSIX does allow sending signals to individual threads.

A process can use several system calls to either mask (ignore) or handle any of the signals in Table 4-1, with the exception of `SIGKILL`. `LibC` exposes the legacy `signal(3)` function, which is built over these system calls.

Process Basic Security

UNIX has traditionally been a multi-user system, wherein more than one user can run more than one process concurrently. To provide both security and isolation, each process holds on to two primary credentials: its creator user identifier (UID) and primary group identifier (GID). These are also known as the *real UID* and *real GID* of the process, but are only part of a larger set of credentials, which also includes any additional group memberships and the *effective* UID/GID. The latter two are commonly equal to the real UID, unless invoked by an executable marked `setuid` (`+s`, `chmod 4xxx`) or setgid (`+g`, `2xxx`) on the file system.

Unlike Linux, there is no support for the `setfsuid`/`setfsgid` system calls in XNU, both of which set the above IDs selectively, only for file system checks — but maintain the real and effective IDs otherwise. This call was originally introduced to deal with NFS, wherein UIDs and GIDs needed to be carried across host boundaries, and often mismatched.

Also, unlike Linux, OS X does not support capabilities. Capabilities are a useful mechanism for applying the principle of least privilege, by breaking down and delegating root privileges to non-root processes. This alleviates the need for a web server, for example, to run as root just to be able to get a binding on the privileged port 80. Capabilities made a cameo appearance in POSIX but were removed (and therefore are not mandated to be supported in OS X), although Linux has eagerly adopted them.

In place of capabilities, OS X and iOS support “entitlements,” which are used in the sandbox compartmentalization mechanism. These, along with code signing, provide a powerful mechanism to contain rogue applications and malware (and, on iOS, any jailbreaking apps) from executing on the system.

EXECUTABLES

A process is created as a result of loading a specially crafted file into memory. This file has to be in a format that is understood by the operating system, which in turn can parse the file, set up the required dependencies (such as libraries), initialize the runtime environment, and begin execution.

In UNIX, anything can be marked as executable by a simple `chmod +x` command. This, however, does not ensure the file can actually execute. Rather, it merely tells the kernel to read this file into memory and seek out one of several header signatures by means of which the exact executable format can be determined. This header signature is often referred to as a "magic," as it is some predefined, often arbitrarily chosen constant value. When the file is read, the "magic" can provide a hint as to the binary format, which, if supported, results in an appropriate loader function being invoked. Table 4-2 provides a list of executable formats.

TABLE 4-2: Executable formats, their signatures, and native OSes

EXECUTABLE FORMAT	MAGIC	USED FOR
PE32/PE32+	`MZ`	Portable executables: The native format in Windows and Intel's Extensible Firmware Interface (EFI) binaries. Although OS X does not support this format, its boot loader does and loads `boot.efi`.
ELF	`\x7FELF`	Executable and Library Format: Native in Linux and most UNIX flavors. ELF is not supported on OS X.
Script	`#!`	UNIX interpreters, or script: Used primarily for shell scripts, but also common for other interpreters such as Perl, AWK, PHP, and so on. The kernel looks for the string following the `#!`, and executes it as a command. The rest of the file is passed to that command via standard input (`stdin`).
Universal (fat) binaries	`0xcafebabe` (Little-Endian) `0xbebafeca` (Big-Endian)	Multiple-architecture binaries used exclusively in OS X.
Mach-O	`0xfeedface` (32-bit) `0xfeedfacf` (64-bit)	OS X native binary format.

Of these various executable formats, OS X currently supports the last three: interpreters, universal binaries, and Mach-O. Interpreters are really just a special case of binaries, as they are merely scripts pointing to the "real" binary, which eventually gets executed. This leaves us to discuss two formats, then — Universal binaries, and Mach-O.

UNIVERSAL BINARIES

With OS X, Apple has touted its rather novel concept of "Universal Binaries." The idea is to provide one binary format that would be fully portable and could execute on any architecture. OS X, which was originally built on the PowerPPC architecture, was ported to the Intel architecture (with Tiger, v10.4.7). Universal binaries would allow binaries to execute on both PPC and x86 processors.

In practice, however, "Universal" binaries are nothing more than archives of the respective architectures they support. That is, they contain a fairly simple header, followed by back-to-back copies of the binary for each supported architecture. Most binaries in Snow Leopard contain only Intel images but still use the universal format to support both 32- and 64-bit compiled code. A few, however, still contain a PowerPC image as well. Up to and including Snow Leopard, OS X contained an optional component, called "Rosetta," which allowed PowerPC emulation on Intel-based processors. With Lion, however, support for PowerPC has officially been discontinued, and binaries no longer contain any PPC images.

As the following example in Output 4-2 shows, `/bin/ls` contains two architectures: the 32-bit Intel version (i386), and the 64-bit Intel version (x86_64). A few binaries in Snow Leopard — such as `/usr/bin/perl` — further contain a PowerPC version (`ppc`).

OUTPUT 4-2: Examining universal binaries using the file(1) command

```
morpheus@Ergo (/) % file /bin/ls    # On snow leopard
/bin/ls:                                   Mach-O universal binary with 2 architectures
/bin/ls (for architecture x86_64):         Mach-O 64-bit executable x86_64
/bin/ls (for architecture i386):           Mach-O executable i386
morpheus@Ergo (/) % file /usr/bin/perl
/usr/bin/perl:                             Mach-O universal binary with 3 architectures
/usr/bin/perl (for architecture x86_64):   Mach-O 64-bit executable x86_64
/usr/bin/perl (for architecture i386):     Mach-O executable i386
/usr/bin/perl (for architecture ppc7400):  Mach-O executable ppc

#
# Some fat binaries, like gdb(1) from the iPhone SDK, can contain different
# architectures, e.g. ARM and intel, side by side
#
morpheus@Ergo (/) cd /Developer/Platforms/iPhoneOS.platform/Developer/usr/libexec/gdb
morpheus@Ergo (.../gdb)$ gdb-arm-apple-darwin
gdb-arm-apple-darwin: Mach-O universal binary with 2 architectures
gdb-arm-apple-darwin (for architecture i386):        Mach-O executable i386
gdb-arm-apple-darwin (for architecture armv7):       Mach-O executable arm
```

Containing multiple copies of the same binaries in this way obviously greatly increases the size of the binaries. Indeed, universal binaries are often quite bloated, which has earned them the less marketable, but more catchy, alias of "fat" binaries. The universal binary tool is, thus, aptly named `lipo`. It can be used to "thin down" the binaries by extracting, removing, or replacing specific architectures. It can also be used to display the fat header details (as you will see in an upcoming experiment).

This universal binary format is defined in `<mach-o/fat.h>` as is shown in Figure 4-2.

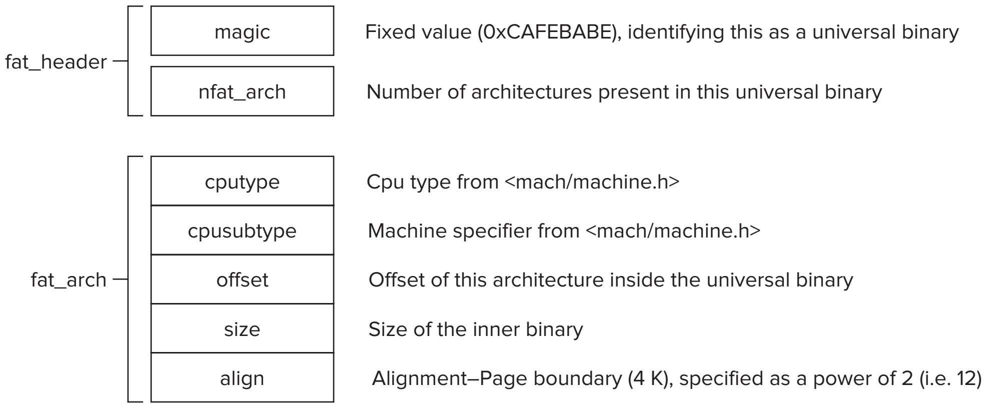


FIGURE 4-2: Fat header format

While universal binaries may take up a lot of space on disk, their structure enables OS X to automatically pick the most suitable binary for the underlying platform. When a binary is invoked, the Mach loader first parses the fat header and determines the available architectures — much as the `lipo` command demonstrates. It then proceeds to load only the most suitable architecture. Architectures not deemed as relevant, thus, do not take up any memory. In fact, the images are all optimized to fit on page boundaries so that the kernel need only load the first page of the binary to read its header, effectively acting as a table of contents, and then proceed to load the appropriate image.

The system picks the image with the `cputype` and `cpusubtype` most closely matching the processor. (This can be overridden with the `arch(1)` command.) Specifically, matching the binary to the architecture is handled by functions in `<mach-o/arch.h>`. Architectures are stored in an `NXArchInfo` struct, which holds the CPU type, `cpusubtype`, and `byteordering` (as well as a textual description). `NXGetLocalArchInfo()` is used to obtain the host's architecture, and `NXFindBestFatArch()` returns the best matching architecture (or NULL, if none match). The code in Listing 4-2 demonstrates some of these APIs.

LISTING 4-2: Handling multiple architectures and universal (fat) binaries

```
#include <stdio.h>
#include <mach-o/arch.h>

const char *ByteOrder(enum NXByteOrder BO)
{
        switch (BO)
        {
                case NX_LittleEndian: return ("Little-Endian");
                case NX_BigEndian:    return ("Big-Endian");
                case NX_UnknownByteOrder: return ("Unknown");
                default: return ("!?!");
        }

}
int main()
{
```

```
   const NXArchInfo *local = NXGetLocalArchInfo();
   const NXArchInfo *known = NXGetAllArchInfos();


while (known && known->description)
{
        printf ("Known: %s\t%x/%x\t%s\n", known->description,
                                         known->cputype, known->cpusubtype,
                                         ByteOrder(known->byteorder));
        known++;

}
if (local) {
printf ("Local - %s\t%x/%x\t%s\n", local->description,
                                     local->cputype, local->cpusubtype,
                                     ByteOrder(local->byteorder));
}

  return(0);

}
```

Experiment: Displaying Universal Binaries with lipo(1) and arch(1)

Using the `lipo(1)` command, you can inspect the fat headers of the various binaries, in this example, Snow Leopard's Perl interpreter:

```
morpheus@Ergo (/) % lipo -detailed_info /usr/bin/perl  # Display specific information.
                                                       # Can also use otool -f
Fat header in: /usr/bin/perl
fat_magic 0xcafebabe
nfat_arch 3
architecture x86_64
    cputype CPU_TYPE_X86_64
    cpusubtype CPU_SUBTYPE_X86_64_ALL
    offset 4096
    size 26144
    align 2^12 (4096)
architecture i386
    cputype CPU_TYPE_I386
    cpusubtype CPU_SUBTYPE_I386_ALL
    offset 32768
    size 25856
    align 2^12 (4096)
architecture ppc7400
    cputype CPU_TYPE_POWERPC
    cpusubtype CPU_SUBTYPE_POWERPC_7400
    offset 61440
    size 24560
    align 2^12 (4096)
```

Using the `arch(1)` command, you can force a particular architecture to be loaded from the binary:

```
morpheus@Ergo (/) % arch -ppc /usr/bin/perl   # Force perl binary to be loaded
You need the Rosetta software to run perl. The Rosetta installer is in Optional Installs
on your Mac OS X installation disc.
```

The Rosetta installer was indeed included in the Optional Installs on the Mac OS X installation disc up to Snow Leopard, but was finally removed in Lion. If you're trying this on Lion, you won't see any PPC binaries — but looking at the iPhone SDK's `gdb` will reveal a mixed platform `gdb`:

```
morpheus@minion (/)$ cd /Developer/Platforms/iPhoneOS.platform/Developer/usr/libexec/gdb
morpheus@minion (.../gdb)$ lipo -detailed_info gdb-arm-apple-darwin
Fat header in: gdb-arm-apple-darwin
fat_magic 0xcafebabe
nfat_arch 2
architecture i386
    cputype CPU_TYPE_I386
    cpusubtype CPU_SUBTYPE_I386_ALL
    offset 4096
    size 2883872
    align 2^12 (4096)
architecture armv7
    cputype (12)
    cpusubtype cpusubtype (9)
    offset 2891776
    size 2537600
    align 2^12 (4096)
```

Mach-O Binaries

UN*X has largely standardized on a common, portable binary format called the Executable and Library Format, or ELF. This format is well documented, has a slew of `binutils` to maintain and debug it, and even allows for binary portability between UN*X of the same CPU architecture (say, Linux and Solaris — and, indeed, Solaris x86 can execute some Linux binaries natively). OS X, however, maintains its own binary format, the Mach-Object (Mach-O), as another legacy of its NeXTSTEP origins.[2]

The Mach-O format (explained in `Mach-O(5)`) and in various Apple documents[3,4] begins with a fixed header. This header, detailed in `<mach-o/loader.h>`, looks like the example in Figure 4-3.

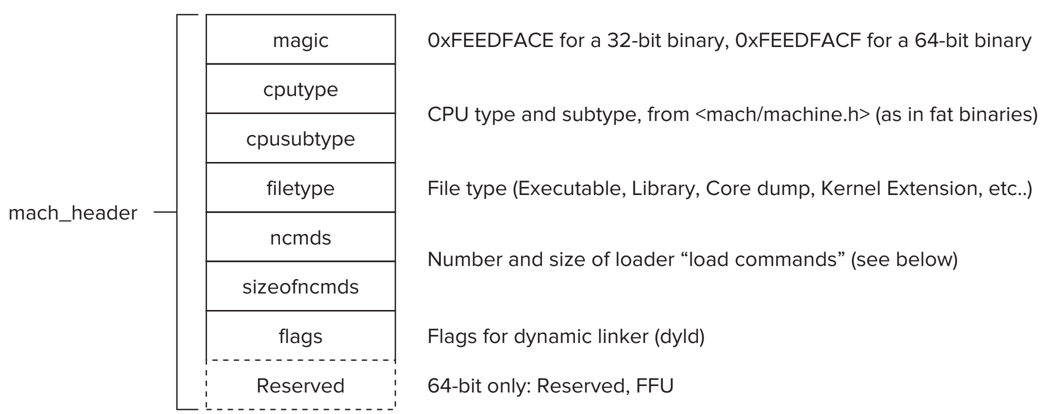


FIGURE 4-3: Mach-O header

The header begins with a magic value that enables the loader to quickly determine if it is intended for a 32-bit (`MH_MAGIC`, `#defined` as `0xFEEDFACE`) or 64-bit architecture (`0xFEEDFACF`, #defined as `MH_MAGIC_64`). Following the magic value are the CPU type and subtype field, which serve the same functionality as in the universal binary header — and ensure that the binary is suitable to be executed on this architecture. Other than that, there are no real differences in the header structure between 32 and 64-bit architectures: while the 64-bit header contains one extra field, it is currently reserved, and is unused.

Because the same binary format is used for multiple object types (executable, library, core file, or kernel extension), the next field, `filetype`, is an `int`, with values defined in `<mach-o/loader.h>` as macros. Common values you'll see in your system include those shown in Table 4-3.

TABLE 4-3: Mach-O file types

FILE TYPE	USED FOR	EXAMPLE
`MH_OBJECT(1)`	Relocatable object files: intermediate compilation results, also 32-bit kernel extensions.	(Generated with `gcc -c`)
`MH_EXECUTABLE(2)`	Executable binaries.	Binaries in `/usr/bin`, and application binary files (in `Contents/MacOS`)
`MH_CORE(4)`	Core dumps.	(Generated in a core dump)
`MH_DYLIB(6)`	Dynamic Libraries.	Libraries in `/usr/lib`, as well as framework binaries
`MH_DYLINKER(7)`	Dynamic Linkers.	`/usr/lib/dyld`
`MH_BUNDLE(8)`	Plug-ins: Binaries that are not standalone but loaded into other binaries. These differ from DYLIB types in that they are explicitly loaded by the executable, usually by NSBundle (Objective-C) or CFBundle (C).	(Generated with `gcc -bundle`) QuickLook plugins at `/System/Library/QuickLook` Spotlight Importers at `/System/Library/Spotlight` Automator actions at `/System/Library/Automator`
`MH_DSYM(10)`	Companion symbol files and debug information.	(Generated with `gcc -g`)
`MH_KEXT_BUNDLE(11)`	Kernel extensions.	64-bit kernel extensions

The header also includes important flags, which are defined in `<mach-o/loader.h>` as well (see Table 4-4).

TABLE 4-4: Mach-O Header Flags

FILE TYPE	USED FOR
`MH_NOUNDEFS`	Objects with no undefined symbols. These are mostly static binaries, which have no further link dependencies
`MH_SPLITSEGS`	Objects whose read-only segments have been separated from read-write ones.
`MH_TWOLEVEL`	Two-level name binding (see "dyld features," discussed later in the chapter).
`MH_FORCEFLAT`	Flat namespace bindings (cannot occur with `MH_TWOLEVEL`).
`MH_WEAK_DEFINES`	Binary uses (exports) weak symbols.
`MH_BINDS_TO_WEAK`	Binary links with weak symbols.
`MH_ALLOW_STACK_EXECUTION`	Allows the stack to be executable. Only valid in executables, but generally a bad idea. Executable stacks are conducive to code injection in case of buffer overflows.
`MH_PIE`	Allow Address Space Layout Randomization for executable types (see later in this chapter).
`MH_NO_HEAP_EXECUTION`	Make the heap non-executable. Useful to prevent the "Heap spray" attack vector, wherein hackers overwrite large portions of the heap blindly with shellcode, and then jump blindly into an address therein, hoping to fall on their code and execute it.

As you can see in the preceding table, there are two flags dealing with "execution": `MH_ALLOW_STACK_EXECUTION` *and* `MH_NO_HEAP_EXECTION`. *Both of these relate to data execution prevention, commonly referred to as NX (Non-eXecutable, referring to the page protection bit of the same name). By making memory pages associated with data non-executable, this (supposedly) thwarts hacker attempts at code injection, as the hacker cannot readily execute code that relies in a data segment. Trying to do so results in a hardware exception, and the process is terminated — crashing it, but avoiding the execution of the injected code.*

Because the common technique of code injection is by stack (or automatic) variables, the stack is marked non-executable by default, and the flag may be (dangerously) used to override that. The heap, by default, remains executable. It is considered harder, although far from impossible, to inject code via the heap.

Both settings can be set on a system-wide basis, by using `sysctl(8)` on the variables `vm.allow_stack_exec` and `vm.allow_heap_exec`. In case of conflict, the more permissive setting (i.e. false before true) applies. In iOS, the `sysctls` are not exposed, and the default is for neither heap nor stack to be executable.

The main functionality of the Mach-O header, however, lies in the load commands. These are specified immediately after the header, and the two fields — `ncmds` and `sizeofncmds` — are used to parse them. I describe those next.

Experiment: Using otool(1) to Investigate Mach-O Files

The `otool(1)` command (part of Darwin's `cctools`) is the native utility to manipulate Mach-O files — and serves as the replacement for the functionality obtained in other UN*X through `ldd` or `readelf`, as well as specific functionality that is only applicable to Mach-O files. The following experiment, using only one of its many switches, `-h`, shows the `mach_header` discussed previously:

```
morpheus@Ergo(/)% otool -hV /bin/ls
/bin/ls:
Mach header
      magic cputype cpusubtype  caps filetype ncmds sizeofcmds    flags
MH_MAGIC_64   X86_6       ALL  LIB64  EXECUTE    13       1928 NOUNDEFS DYLDLINK TWOLEVEL
morpheus@Ergo(/)% otool -arch i386 -hV /bin/ls  # force otool to show the 32-bit header
/bin/ls:
Mach header
   magic cputype cpusubtype  caps    filetype ncmds sizeofcmds    flags
MH_MAGIC    I386        ALL  0x00     EXECUTE    13       1516 NOUNDEFS DYLDLINK TWOLEVEL

morpheus@Ergo(/)% gcc -g a.c -o a   # Compile any file, but use "-g"
morpheus@Ergo(/)% ls -ld a.*
-rw-r--r--  1 morpheus  staff   16 Jan 22 08:29 a.c
drwxr-xr-x  3 morpheus  staff  102 Jan 22 08:29 a.dSYM
```

Note the –g, which usually embeds symbols inside the binary in other UN*X systems, does so on OS X in a companion file

```
morpheus@Ergo(/)% otool -h a.dSYM/Contents/Resources/DWARF/a
a.dSYM/Contents/Resources/DWARF/a:
Mach header
      magic cputype cpusubtype  caps    filetype ncmds sizeofcmds      flags
 0xfeedfacf 16777223          3  0x00          10     7       1768 0x00000000

# Sample using otool on a quick look plugin, which is an MH_BUNDLE:
morpheus@Ergo(/)% otool -h /System/Library/QuickLook/PDF.qlgenerator/Contents/MacOS/PDF
/System/Library/QuickLook/PDF.qlgenerator/Contents/MacOS/PDF:
Mach header
      magic cputype cpusubtype  caps    filetype ncmds sizeofcmds      flags
 0xfeedfacf 16777223          3  0x00           8    13       1824 0x00000085

# Of course, we could have used the verbose mode here..
morpheus@Ergo(/) % otool -hV /System/Library/QuickLook/PDF.qlgenerator/Contents/MacOS/PDF
/System/Library/QuickLook/PDF.qlgenerator/Contents/MacOS/PDF:
Mach header
      magic cputype cpusubtype  caps    filetype ncmds sizeofcmds      flags
MH_MAGIC_64  X86_64        ALL  0x00      BUNDLE    13       1824   NOUNDEFS
DYLDLINK TWOLEVEL
```

> `otool(1)` *is good in analyzing load commands and text segments, but leaves much to be desired in analyzing data segments, and other areas. The book's companion website features an additional binary,* `jtool`, *which aims to improve on otool's functionality. The tool can handle all objects up to and including those of iOS 5.1 and Mountain Lion. It integrates features from* `nm(1)`, `strings(1)`, `segedit(1)`, `size(1)`, *and* `otool(1)` *into one binary, especially suited for scripting, and adds several new features, as well.*

Load Commands

The Mach-O header contains very detailed instructions, which clearly direct how to set up and load the binary, when it is invoked. These instructions, or "load commands," are specified immediately after the basic `mach_header`. Each command is itself a type-length-value: A 32-bit `cmd` value specifies the type, a 32-bit value `cmdsize` (a multiple of 4 for 32-bit, or 8 for 64-bit), and the command (of arbitrary `len`, specified in `cmdsize`) follows. Some of these commands are interpreted directly by the kernel loader (`bsd/kern/mach_loader.c`). Others are handled by the dynamic linker.

There are over 30 such load commands. Table 4-5 describes those the kernel uses. (We discuss the rest, which are used by the link editor, later.)

TABLE 4-5: Mach-O Load Commands Processed by the Kernel

#	COMMAND	KERNEL HANDLER FUNCTION (BSD/KERN/MACH/LOADER.C)	USED FOR
0x01 0x19	LC_SEGMENT LC_SEGMENT_64	load_segment	Maps a (32- or 64-bit) segment of the file into the process address space. These are discussed in more detail in "process memory map."
0x0E	LC_LOAD_DYLINKER	load_dylinker	Invoke `dyld` (`/usr/lib/dyld`).
0x1B	LC_UUID	Kernel copies UUID into internal mach object representation	Unique 128-bit ID. This matches a binary with its symbols
0x04	LC_THREAD	load_thread	Starts a Mach Thread, but does not allocate the stack (rarely used outside core files).
0x05	LC_UNIXTHREAD	load_unixthread	Start a UNIX Thread (initial stack layout and registers). Usually, all registers are zero, save for the instruction pointer/program counter. This is deprecated as of Mountain Lion, replaced by `dyld`'s `LC_MAIN`.
0x1D	LC_CODE_SIGNATURE	load_code_signature	Code Signing. (In OS X — occasionally used. In iOS — mandatory.)
0x21	LC_ENCRYPTION_INFO	set_code_unprotect()	Encrypted binaries. Also largely unused in OS X, but ubiquitous in iOS.

The kernel portion of the loading process is responsible for the basic setup of the new process — allocating virtual memory, creating its main thread, and handling any potential code signing/

encryption. For dynamically linked (read: the vast majority of) executables, however, the actual loading of libraries and resolving of symbols is handled in user mode by the dynamic linker specified in the `LC_LOAD_DYLINKER` command. Control will be transferred to the linker, which in turn further processes other load commands in the header. (Loading of libraries is discussed later in this chapter)

A more detailed discussion of these load commands follows.

LC_SEGMENT and the Process Virtual Memory Setup

The main load command is the `LC_SEGMENT` (or `LC_SEGMENT64`) commands, which instructs the kernel how to set up the memory space of the newly run process. These "segments" are directly loaded from the Mach-O binary into memory.

Each `LC_SEGMENT[_64]` command provides all the necessary details of the segment layout (see Table 4-6).

TABLE 4-6: LCSEGMENT or LC_SEGMENT_64 Parameters

PARAMETER	USE
`segname`	`load_segment`
`vmaddr`	Virtual memory address of segment described
`vmsize`	Virtual memory allocated for this segment
`fileoff`	Marks the segment beginning offset in the file
`filesize`	Specifies how many bytes this segment occupies in the file
`maxprot`	Maximum memory protection for segment pages, in octal (4=r, 2=w, 1=x)
`initprot`	Initial memory protection for segment pages
`nsects`	Number of sections in segment, if any
`flags`	Miscellaneous bit flags

Setting up the process's virtual memory thus becomes a straightforward operation of following the `LC_SEGMENT` commands. For each segment, the memory is loaded from the file: *`filesize`* bytes from offset *`fileoff`*, to *`vmsize`* bytes at address *`vmaddr`*. Each segment's pages are initialized according to *`initprot`*, which specifies the initial page protection in terms of read/write/execute bits. A segment's protection may be dynamically changed, but cannot exceed the values specified in *`maxprot`*. (In iOS, specifying `+x` is mutually exclusive to `+w`.)

`LC_SEGMENT`s are provided for `__PAGEZERO` (`NULL` pointer trap), `_TEXT` (program code), `_DATA` (program data), and `_LINKEDIT` (symbol and other tables used by linker). Segments may optionally be further broken up into sections. Table 4-7 shows some of these sections.

TABLE 4-7: Common segments and sections in Mach-O executables

SECTION	USE
`__text`	Main program code
`__stubs, __stub_helper`	Stubs used in dynamic linking
`__cstring`	C hard-coded strings in the program
`__const`	`const` keyworded variables and hard coded constants
`__TEXT.__objc_methname`	Objective-C method names
`__TEXT.__objc_methtype`	Objective-C method types
`__TEXT.__objc_classname`	Objective-C class names
`__DATA.__objc_classlist`	Objective-C class list
`__DATA.__objc_protolist`	Objective-C prototypes
`__DATA.__objc_imginfo`	Objective-C image information
`__DATA.__objc_const`	Objective-C constants
`__DATA.__objc_selfrefs`	Objective-C Self (this) references
`__DATA.__objc_protorefs`	Objective-C prototype references
`__DATA.__objc_superrefs`	Objective-C superclass references
`__DATA.__cfstring`	Core Foundation strings (`CFStringRefs`) in the program
`__DATA.__bss`	BSS

Segments may also have certain flags set, defined in `<mach/loader.h>`. One such flag used by Apple is `SG_PROTECTED_VERSION_1 (0x08)`, denoting the segment pages are "protected" — i.e., encrypted. Apple encrypts select binaries using this technique — for example, the Finder, as shown in Output 4-3.

OUTPUT 4-3: Using otool(1) on the Finder, displaying the encrypted section

```
morpheus@ergo (/) otool -lV /System/Library/CoreServices/Finder.app/Contents/MacOS
 /Finder
/System/Library/CoreServices/Finder.app/Contents/MacOS/Finder:
Load command 0
      cmd LC_SEGMENT_64
..
  segname __PAGEZERO
 ..
Load command 1
      cmd LC_SEGMENT_64
```

```
  cmdsize 872
  segname __TEXT
   vmaddr 0x0000000100000000
   vmsize 0x00000000003ad000
  fileoff 0
 filesize 3854336
  maxprot rwx
 initprot r-x
   nsects 10
    flags PROTECTED_VERSION_1
```

To enable this code encryption, XNU — the kernel — contains a specific a custom (external) virtual memory manager called "Apple protect," which is discussed in Chapter 12, "Mach Virtual Memory."

XCode's `ld(1)` can be instructed to create segments when constructing Mach-O objects, by using the `-segcreate` switch. XCode likewise, contains a special tool, `segedit(1)`, which can be used to extract or replace segments from a Mach-O file. This can be useful for extracting embedded textual information, like the sections `PRELINK_INFO` of the kernel, as will be demonstrated in chapter 17. Alternatively, the book's companion tool — jtool — offers this functionality as well. The jtool also provides the functionality of a third XCode tool, `size(1)`, which prints the sizes and addresses of the segments.

LC_UNIXTHREAD

Once all the libraries are loaded, `dyld`'s job is done, and the `LC_UNIXTHREAD` command is responsible for starting the binary's main thread (and is thus always present in executables, but not in other binaries, such as libraries). Depending on the architecture, it will list all the initial register states, with a particular flavor that is `i386_THREAD_STATE`, `x86_THREAD_STATE64`, or — in iOS binaries — `ARM_THREAD_STATE`. In any of the flavors, most of the registers will likely be initialized to zero, save for the Instruction Pointer (on Intel) or the Program Counter (r15, on ARM), which hold the address of the program's entry point.

> *Before Apple completely abandoned the PPC platform in Lion, there was also a* `PPC_THREAD_STATE`*. This is still visible on some of the PPC-code containing fat binaries (try* `otool -arch ppc -l /mach_kernel` *on Snow Leopard. Register* `srr0` *is the code entry point in this case.*

LC_THREAD

Similar to `LC_UNIXTHREAD`, `LC_THREAD` is used in core files. The Mach-O core files are, in essence, a collection of `LC_SEGMENT` (or `LC_SEGMENT_64`) commands that set up the memory image of the (now defunct) process, and a final `LC_THREAD`. The `LC_THREAD` contains several "flavors," for each of the machine states (i.e. thread, float, and exception). You can confirm that easily by generating a core dump (which is, alas, all too easy), and then inspecting it with `otool -l`.

LC_MAIN

As of Mountain Lion, a new load command, `LC_MAIN` supersedes the `LC_UNIXTHREAD` command. This command is used to set the entry point address and stack size of the main thread of the program. This makes more sense than using `LC_UNIXTHREAD`, as in any case all the registers save for the program counter are set to zero. With no `LC_UNIXTHREAD`, it is impossible to run Mountain Lion binaries that use `LC_MAIN` on previous OS X versions (causing `dyld(1)` to crash on loading).

LC_CODE_SIGNATURE

An interesting feature of Mach-O binaries is that they can be digitally signed. In OS X this is still largely unused, although it is gaining popularity as code signing ties into the newly improved sandbox mechanism. In iOS, code signing is mandatory, in another attempt by Apple to lock down the system as much as it possibly can: The only signature recognized in iOS is that of Apple. In OS X, the `codesign(1)` utility may be used to manipulate and display code signatures. The man page, as well as Apple's code signing guide and Mac OS X Code Signing In Depth[1] all detail code signing from the administrator's perspective.

The `LC_CODE_SIGNATURE` contains the code signature of the Mach-O binary, and — if it does not match the code (or, in iOS, does not exist) — the process is killed immediately by the kernel with a `SIGKILL`. No questions asked, no saving throw. Prior to iOS 4, it was possible to disable code signature checks with two `sysctl(8)` commands, to overwrite the kernel variables responsible for enforcement, using the kernel's MAC (Mandatory Access Control) component:

```
sysctl -w security.mac.proc_enforce=0  // disable MAC for process
sysctl -w security.mac.vnode_enforce=0 // disable MAC for VNode
```

In later iOSes, however, Apple realized that — upon getting root — jailbreakers would also be able to overwrite the variables. So the variables were made read-only. The "untethered" jailbreaks are able to set the variables anyway due to a kernel-based exploit. The variable default value, however, is enabled, and so the "tethered" jailbreaks result in the non–Apple-signed applications crashing — unless the i-Device is booted in a tethered manner.

Alternatively, a fake code signature can be embedded in the Mach-O, using a tool like Saurik's `ldid`. This tool, an alternative to OS X's `codesign(1)`, enables the generation of fake signatures with self-signed certificates. This is especially important in iOS, as signatures are tied to the sandbox model's application "entitlements," which are mandatory in iOS. Entitles are declarative permissions (in `plist` form), which must be embedded in the Mach-O and sealed by signing, in order to allow runtime privileges for security-sensitive operations.

Both OS X and iOS contain a special system call, `csops` (#169), for code signing operations. Code signatures and MAC are explained in detail from the kernel's perspective in Chapter 12.

Experiment: Observing Load Commands and Dynamic Loading — Stage I

Recall `/bin/ls` in the previous experiment, and that `otool -h` reported 13 load commands. To display them, we use `otool -l` (some commands have been omitted from this sample). As before, we examine a 64-bit binary (see Figure 4-4). You are encouraged to examine a 32-bit binary by specifying `–arch i386` to `otool`.

DYNAMIC LIBRARIES

As discussed in the previous chapter, executables are seldom standalone. With the exception of very few statically linked ones, most executables are dynamically linked, relying on pre-existing libraries, supplied either as part of the operating system, or by third parties. This section turns to discussing the process of library loading: During application launch, or runtime.

Launch-Time Loading of Libraries

The previous section covered the setup performed by the kernel loader (in `bsd/kern/mach_loader.c`) to initialize the process address space according to the segments and other directives. This suffices for very few processes, however, as virtually all programs on OS X are dynamically linked. This means that the Mach-O image is filled with "holes" — references to external libraries and symbols — which are resolved when the program is launched. This is a job for the dynamic linker. This process is also referred to as symbol "binding."

The dynamic linker, you'll recall, is started by the kernel following an `LC_DYLINKER` load command. Typically, it is `/usr/lib/dyld` — although any program can be specified as an argument to this command. The linker assumes control of the fledgling process, as the kernel sets the entry point of the process to that of the linker.

The linker's job is to, literally, "fill the holes" — that is, it must seek out any symbol and library dependencies and resolve them. This must be done recursively, as it is often the case that libraries have dependencies on other libraries still.

> `dyld` *is a user mode process. It is not part of the kernel and is maintained as a separate open source project (though still part of Darwin) by Apple at* `http://www.opensource.apple.com/source/dyld`*. As far as the kernel is concerned,* `dyld` *is a pluggable component and it may be replaced with a third-party linker. Despite (and, actually, because of) being in user mode, the link editor plays an important part in loading processes. Loading libraries from kernel mode would be much harder because files as we see them in user mode do not exist in kernel mode.*

The linker scans the Mach-O header for specific load commands of interest (see Table 4-8).

```
Ergo (/) % otool -l /bin/ls
Load command 0
      cmd LC_SEGMENT_64
  cmdsize 72
  segname __PAGEZERO
   vmaddr 0x0000000000000000
   vmsize 0x0000000100000000
  fileoff 0
 filesize 0
  maxprot 0x00000000
 initprot 0x00000000
   nsects 0
    flags 0x0
Load command 1
      cmd LC_SEGMENT_64
  cmdsize 632
  segname __TEXT
   vmaddr 0x0000000100000000
   vmsize 0x0000000000006000
  fileoff 0
 filesize 24576
  maxprot 0x00000007
 initprot 0x00000005
   nsects 7
    flags 0x0
Section
  sectname __text
   segname __TEXT
      addr 0x0000000100001478
      size 0x00000000000038ef … …
... (other sections omitted) ..
Load command 7
          cmd LC_LOAD_DYLINKER
      cmdsize 32
     name /usr/lib/dyld (offset 12)
......
```

The linker can be instructed to trace `LC_SEGMENT` commands by setting the DYLD_PRINT_SEGMENTS to some non-zero value

```
Ergo% export DYLD_PRINT_SEGMENTS=1
  Ergo ( ) % ls
dyld: Main executable mapped /bin/ls
        __PAGEZERO at 0x00000000->0x100000000
            __TEXT at 0x100000000->0x100006000
            __DATA at 0x100006000->0x100007000
        __LINKEDIT at 0x100007000->0x10000A000
<.. rest of setup performed by dyld for loading libraries, etc ..>
```

Note `PAGEZERO` didn't take up any space on disk `(filesize:0)`. Other segments are loaded `mmap()ed` from their offset in the file directly into memory

maxprot: Maximum protection for this segment `(rwx)`

initprot: Initial protection for this segment `(r-x)`

Seven sections follow in this segment (omitted). Note, though, the `__text` segment, starting at `0x0100001478`.

The reference to `/usr/lib/dyld`, which loads and parses the other headers

```
Load command 9
        cmd LC_UNIXTHREAD
    cmdsize 184
     flavor x86_THREAD_STATE64
      count x86_THREAD_STATE64_COUNT
   rax 0x0000000000000000 rbx 0x0000000000000000 rcx 0x0000000000000000
  rdx 0x0000000000000000 rdi 0x0000000000000000 rsi 0x0000000000000000
  rbp 0x0000000000000000 rsp 0x0000000000000000 r8  0x0000000000000000
   r9 0x0000000000000000 r10 0x0000000000000000 r11 0x0000000000000000
  r12 0x0000000000000000 r13 0x0000000000000000 r14 0x0000000000000000
  r15 0x0000000000000000 rip 0x0000000100001478
rflags 0x0000000000000000 cs 0x0000000000000000 fs 0x0000000000000000
   gs 0x0000000000000000
..
Load command 10
         cmd LC_LOAD_DYLIB
     cmdsize 56
      name /usr/lib/libncurses.5.4.dylib (offset 24)
Load command 11
         cmd LC_LOAD_DYLIB
     cmdsize 56
         name /usr/lib/libSystem.B.dylib (offset 24)
 time stamp 2 Wed Dec 31 19:00:02 1969
     current version 125.2.0
compatibility version 1.0.0
Load command 12
     cmd LC_CODE_SIGNATURE
  cmdsize 16
 dataoff 34160
 datasize 5440
```

RIP will point to the binary's entry. As in this case, it commonly also happens to be the address of the text section

```
Ergo (/) % otool -tV /bin/ls
/bin/ls:
(__TEXT,__text) section
0000000100001478      pushq   $0x00
000000010000147a      movq    %rsp,%rbp
000000010000147d      andq    $0xf0,%rsp
```

These are the libraries this binary depends on — to be loaded by `dyld`

FIGURE 4-4: Load Commands of a simple binary

TABLE 4-8: Load Commands Processed by dyld

	LOAD COMMAND	USED FOR
0x02 0x0B	LC_SYMTAB LC_DSYMTAB	Symbol tables. The symbol tables and string tables are provided separately, at an offset specified in these commands.
0x0C	LC_LOAD_DYLIB	Load additional dynamic libraries. This command supersedes `LC_LOAD_FVMLIB`, used in NeXTSTEP.
0x20	LC_LAZY_LOAD_DYLIB	As `LC_LOAD_DYLIB`, but defer actual loading until use of first symbol from library
0x0D	LC_ID_DYLIB	Found in dylibs only. Specifies the ID, the timestamp, version, and compatibility version of the `dylib`.
0x1F	LC_REEXPORT_DYLIB	Found in dynamic libraries only. Allows a library to re-export another library's symbols as its own. This is how Cocoa and Carbon serve as umbrella frameworks for many others, as well as `libSystem` (which exports libraries in `/usr/lib/system`).
0x24 0x25	LC_VERSION_MIN_IPHONEOS LC_VERSION_MIN_MACOSX	Minimum operating system version expected for this binary. As of Lion, many binaries are set to 10.7 at a minimum.
0x26	LC_FUNCTION_STARTS	Compressed table of function start addresses. New in Mountain Lion
0x2A	LC_SOURCE_VERSION	Version of source code used to build this binary. Informational only and does not affect linking in any known way.
0x2B	?? (Name unknown)	Code Signing sections from `dylibs`

The library dependencies can be displayed by using `otool -L` (the OS X equivalent to the functionality provided in other UN*X by `ldd`). As in other operating systems, however, the `nm` command can be used to display the symbol table of a Mach-O binary, as you will see in the upcoming experiment. The OS X `nm(1)` supports a `-m` switch, which allows to not only display the symbols, but also to follow their resolution. Alternatively, the `dyldinfo(1)` command (part of XCode) may be used for this purpose. Using this command, you can also display the opcodes used by the linker when loading the libraries, as shown in Output 4-4:

OUTPUT 4-4: Displaying dyld's binding opcodes

```
morpheus@ergo (/)$ dyldinfo -opcodes /bin/ls | more
...
lazy binding opcodes:
0x0000 BIND_OPCODE_SET_SEGMENT_AND_OFFSET_ULEB(0x02, 0x00000014)
0x0002 BIND_OPCODE_SET_DYLIB_ORDINAL_IMM(2)
```

```
0x0003 BIND_OPCODE_SET_SYMBOL_TRAILING_FLAGS_IMM(0x00, ___assert_rtn)
0x0012 BIND_OPCODE_DO_BIND()
0x0013 BIND_OPCODE_DONE
0x0014 BIND_OPCODE_SET_SEGMENT_AND_OFFSET_ULEB(0x02, 0x00000018)
0x0016 BIND_OPCODE_SET_DYLIB_ORDINAL_IMM(2)
0x0017 BIND_OPCODE_SET_SYMBOL_TRAILING_FLAGS_IMM(0x00, ___divdi3)
0x0022 BIND_OPCODE_DO_BIND()
0x0023 BIND_OPCODE_DONE
```

Binaries that use functions and symbols defined externally have a section (__stubs) in their text segment, with placeholders for the undefined symbols. The code is generated with a call to the symbol stub section, which is resolved by the linker during runtime. The linker resolves it by placing a JMP instruction at the called address. The JMP transfers control to the real function's body, but without modification of the stack in any way. The real function can thus return normally, as if it had been called directly.

`LC_LOAD_DYLIB` commands instruct the linker where the symbols can be found. Each library specified is loaded and searched for the matching symbols. The library to be linked has a symbol table, which links the symbol names to the addresses. The address can be found in the Mach-O object at the `symoff` specified by the `LC_SYMTAB` load command. The corresponding symbol names are at `stroff`, and there are a total of `nsyms`.

Like all other UN*X, Mach-O libraries can be found in `/usr/lib` (there is no `/lib` in OS X or iOS). There are two main differences, however:

- Libraries are not "shared objects" (`.so`), as OS X is not ELF-compatible, and this concept does not exist in Mach-O. Rather, they are "dynamic library" files, with a `.dylib` extension.
- There is no `libc`. Developers may be familiar with the C Runtime library on other UN*X (or MSVCRT, on Windows). But the corresponding library, `/usr/lib/libc.dylib`, exists only as a symbolic link to `libSystem.B.dylib`. `libSystem` provides `LibC` functionality, as well as additional functions, which in UN*X are provided by separate libraries — for example, mathematical functions (`-lm`), hostname resolution (`-lnsl`), and threads (`-lpthread`).

`libSystem` is the absolute prerequisite of all binaries on the system, C, C++, Objective-C, or otherwise. This is because it serves as the interface to the lower-level system calls and kernel services, without which nothing would get done. It actually serves as an umbrella library for the various libraries in `/usr/lib/system`, which it re-exports (using the `LC_REEXPORT_LIB` load command). In Snow Leopard, only eight or so libraries are re-exported. The number increases dramatically in Lion and iOS to well over 20.

Experiment: Viewing Symbols and Loading

Consider the following simple "hello world" program. It calls on `printf()` twice, then exits:

```
morpheus@Ergo (~) % cat a.c
void main (int argc, char **argv) {
printf ("Salve, Munde!\n");
printf ("Vale!\n");
exit(0);
}
```

Using Xcode's `dyldinfo(1)` `nm(1)` you can resolve the binding and figure out which symbols are exported, and what libraries they are linked against.

```
morpheus@Ergo (~) % dyldinfo -lazy_bind a
lazy binding information (from lazy_bind part of dyld info):
segment section          address    index  dylib            symbol
__DATA  __la_symbol_ptr  0x100001038 0x0000 libSystem        _exit
__DATA  __la_symbol_ptr  0x100001040 0x000C libSystem        _puts
```

Using XCode's `otool(1)`, you can go "under the hood" and actually see things at the assembly level (Output 4-5A and 3-5B):

OUTPUT 4-5A: Demonstrating `otool`'s disassembly of a simple binary

```
morpheus@Ergo (~) %  otool -p _main -tV a # use otool to disassemble, starting at _main:
a:
(__TEXT,__text) section
_main:
0000000100000ed0        pushq   %rbp
0000000100000ed1        movq    %rsp,%rbp
0000000100000ed4        subq    $0x20,%rsp
0000000100000ed8        movl    %edi,%eax
0000000100000eda        movl    $0x00000000,%ecx
0000000100000edf        movl    %eax,0xfc(%rbp)
0000000100000ee2        movq    %rsi,0xf0(%rbp)
0000000100000ee6        leaq    0x00000057(%rip),%rax
0000000100000eed        movq    %rax,%rdi
0000000100000ef0        movl    %ecx,0xec(%rbp)
0000000100000ef3        callq   0x100000f18     ; symbol stub for: _puts
0000000100000ef8        leaq    0x00000053(%rip),%rax
0000000100000eff        movq    %rax,%rdi
0000000100000f02        callq   0x100000f18     ; symbol stub for: _puts
0000000100000f07        movl    0xec(%rbp),%eax
0000000100000f0a        movl    %eax,%edi
0000000100000f0c        callq   0x100000f12     ; symbol stub for: _exit
```

OUTPUT 4-5B: Disassembling the same program, in its iOS form

```
Podicum:~ root# otool -tV -p _main a.arm
a.arm:
(__TEXT,__text) section
_main:
00002f9c            b580        push    {r7, lr}
00002f9e            466f        mov     r7, sp
00002fa0            b084        sub     sp, #16
00002fa2            9003        str     r0, [sp, #12]
00002fa4            9102        str     r1, [sp, #8]
00002fa6        f2400032        movw    r0, 0x32
00002faa        f2c00000        movt    r0, 0x0
00002fae            4478        add     r0, pc
00002fb0        f000e812        blx     0x2fd8  @ symbol stub for: _puts
00002fb4            9001        str     r0, [sp, #4]
00002fb6        f2400030        movw    r0, 0x30
00002fba        f2c00000        movt    r0, 0x0
```

```
00002fbe        4478        add     r0, pc
00002fc0        f000e80a    blx     0x2fd8  @ symbol stub for: _puts
00002fc4        9000        str     r0, [sp, #0]
00002fc6        2000        movs    r0, #0
00002fc8        f000e800    blx     0x2fcc  @ symbol stub for: _exit
```

As the example shows, calls to `exit()` and `printf` (optimized by the compiler to `puts`, because it prints a constant, newline-terminated string rather than a format string) are left unresolved, as a call to specific addresses. These addresses are the symbol-stub table and are left up to the Linker to initialize. You can next use the `otool -l` again to show the load commands, in particular focusing on the `stubs` section. Output 4-6 shows the output of doing so, aligning OS X with iOS:

OUTPUT 4-6: Running otool(1) on OS X and iOS, to display symbol tables

Mac OS X (x86_64)	iOS 5.0 (armv7)
morpheus@Ergo (~) % **otool -l -V a**	morpheus@Ergo (~) % **otool -l -V a.arm**
<pre>Section sectname __stubs segname __TEXT addr 0x0000000100000f12 size 0x000000000000000c offset 3880 align 2^1 (2) reloff 0 nreloc 0 type S_SYMBOL_STUBS attributes PURE_INSTRUCTIONS SOME_INSTRUCTIONS reserved1 0 (index into indirect symbol table) reserved2 6 (size of stubs)</pre>	<pre>Section sectname __symbol_stub4 segname __TEXT addr 0x0000209c size 0x00000018 offset 4252 align 2^2 (4) reloff 0 nreloc 0 type S_SYMBOL_STUBS attributes PURE_INSTRUCTIONS SOME_INSTRUCTIONS reserved1 0 (index into indirect symbol table) reserved2 12 (size of stubs)</pre>
<pre>Section sectname __stub_helper segname __TEXT addr 0x0000000100000f20 size 0x0000000000000024 offset 3872 align 2^2 (4) reloff 0 nreloc 0 type S_REGULAR attributes PURE_INSTRUCTIONS SOME_INSTRUCTIONS reserved1 0 reserved2 0 ...</pre>	No __stub_helper section
<pre>Section sectname __nl_symbol_ptr segname __DATA addr 0x0000000100001028 size 0x0000000000000010 offset 4136 align 2^3 (8)</pre>	<pre>Section sectname __nl_symbol_ptr segname __DATA addr 0x0000301c size 0x00000008 offset 8220 align 2^2 (4)</pre>

OUTPUT 4-6 *(continued)*

```
    reloff 0
    nreloc 0
      type S_NON_LAZY_
SYMBOL_POINTERS
attributes (none)
 reserved1 2 (index into indirect symbol table)
 reserved2 0
```

```
    reloff 0
    nreloc 0
      type S_NON_LAZY_
SYMBOL_POINTERS
attributes (none)
 reserved1 2 (index into indirect symbol table)
 reserved2 0
```

```
Section
  sectname __la_symbol_ptr
   segname __DATA
      addr 0x0000000100001038
      size 0x0000000000000010
    offset 4152
     align 2^3 (8)
    reloff 0
    nreloc 0
      type S_LAZY_SYMBOL_POINTERS
attributes (none)
 reserved1 4 (index into indirect symbol table)
 reserved2 0
...
```

```
Section
  sectname __la_symbol_ptr
   segname __DATA
      addr 0x00003024
      size 0x00000008
    offset 8228
     align 2^2 (4)
    reloff 0
    nreloc 0
      type S_LAZY_SYMBOL_POINTERS
attributes (none)
 reserved1 4 (index into indirect symbol table)
 reserved2 0
```

```
Load command 5
     cmd LC_SYMTAB
 cmdsize 24
  symoff 8360
   nsyms 11
  stroff 8560
 strsize 112
...
Load command 10
          cmd LC_LOAD_DYLIB
      cmdsize 56
         name /usr/lib/libSystem.B.dylib (offset 24)
   time stamp 2 Wed Dec 31 19:00:02 1969
      current version 125.2.11
compatibility version 1.0.0
```

```
Load command 4
     cmd LC_SYMTAB
 cmdsize 24
  symoff 12296
   nsyms 12
  stroff 1246
 strsize 148
```

Finally, you can use `nm` to display the unresolved symbols. These are the same in OS X and iOS.

```
morpheus@Ergo (~) % nm a | grep "U "     # and here are our three unresolved symbols
                 U _exit
                 U _puts
                 U dyld_stub_binder
morpheus@Ergo (~) % nm a | wc -l         # How many symbols in table, overall?
      11                                 # (12 on ARM - also__dyld_func_lookup)
```

And you can use `gdb` to dump the symbol stubs and the `stub_helper`. Note the stub is a `JMP` to a symbol table:

```
morpheus@Ergo (~) % gdb ./a
GNU gdb 6.3.50-20050815 (Apple version gdb-1472) (Wed Jul 21 10:53:12 UTC 2010)
..
done

(gdb) x/2i 0x100000f12 # Dump the address as (2) instructions
0x100000f12 <dyld_stub_exit>:    jmpq   *0x120(%rip)        # 0x100001038
0x100000f18 <dyld_stub_puts>:    jmpq   *0x122(%rip)        # 0x100001040

(gdb) x/2g  0x100001038 # Dump the address as   (2) 64 bit pointers
0x100001038:   0x0000000100000f20   0x0000000100000f2a   // Both in __stub_helper

(gdb) x/2i 0x100000f20      # dump the stub code for exit
0x100000f20:   pushq  $0x0  // pushes "0" on the stack
0x100000f25:     jmpq   0x100000f34

(gdb) x/2i 0x100000f2a                // dump the stub code for puts
0x100000f2a:   pushq  $0xc            // pushes „12" on the stack
0x100000f2f:   jmpq   0x100000f34

# Both jump to 0x100000f34 - so let's inspect that:

(gdb) x/3i 0x100000f34                        // All stubs end up here
0x100000f34:   lea    0xf5(%rip),%r11         # 0x100001030
0x100000f3b:   push   %r11
0x100000f3d:   jmpq   *0xe5(%rip)             # 0x100001028  // dyld_stub_binder


// note the address we jump to is ... empty!
(gdb) x/2g 0x100001028
0x100001028:   0x0000000000000000     0x0000000000000000
```

Setting a breakpoint on `main()` in `gdb`, and then running it, will break the program right after dynamic linkage is complete but before anything gets executed. This will give you a chance to see the address of `dyld_stub_linker` populated:

```
(gdb) b main  # set breakpoint
Breakpoint 1 at 0x100000ef3
(gdb) r       # We don't really want to run - we just dyld(1) to link
Starting program: /Users/morpheus/a
Reading symbols for shared libraries +. done

Breakpoint 1, 0x0000000100000ef3 in main ()

(gdb) x/2g 0x100001028             // revisiting the mystery address:
0x100001028: 0x00007fff89527f94      0x0000000000000000

(gdb) disass 0x00007fff89527f94   // Address now contains dyld_stub_binder
Dump of assembler code for function dyld_stub_binder:
0x00007fff89527f94 <dyld_stub_binder+0>:     push   %rbp
0x00007fff89527f95 <dyld_stub_binder+1>:     mov    %rsp,%rbp
0x00007fff89527f98 <dyld_stub_binder+4>:     sub    $0xc0,%rsp
. . .
```

DISASSEMBLY OF THE SAME SYMBOL, ON IOS:

```
(gdb) x/2i dyld_stub_exit
0x2fcc <dyld_stub_exit>:      ldr     r12, [pc, #0]  ; 0x2fd4 <dyld_stub_exit+8>
0x2fd0 <dyld_stub_exit+4>:    ldr     pc, [r12]

(gdb) x/2i dyld_stub_puts
0x2fd8 <dyld_stub_puts>:      ldr     r12, [pc, #0]  ; 0x2fe0 <dyld_stub_puts+8>
0x2fdc <dyld_stub_puts+4>:    ldr     pc, [r12]

(gdb) x/x 0x2fd4
0x2fd4 <dyld_stub_exit+8>:    0x00003024
(gdb) x/x 0x2fe0
0x2fe0 <dyld_stub_puts+8>:    0x00003028

(gdb) x/2x 0x3024
0x3024: 0x00002f70     0x00002f70

(gdb) disass 0x2f70
Dump of assembler code for function dyld_stub_binding_helper:
0x00002f70 <dyld_stub_binding_helper+0>:   push   {r12}          ; (str r12, [sp, #-4]!)
0x00002f74 <dyld_stub_binding_helper+4>:   ldr    r12, [pc, #12] ; 0x2f88
0x00002f78 <dyld_stub_binding_helper+8>:   ldr    r12, [pc, r12]
0x00002f7c <dyld_stub_binding_helper+12>:  push   {r12}         ; (str r12, [sp, #-4]!)
0x00002f80 <dyld_stub_binding_helper+16>:  ldr    r12, [pc, #4] ; 0x2f8c
0x00002f84 <dyld_stub_binding_helper+20>:  ldr    pc, [pc, r12]
... # Following instructions irrelevant since "ldr pc" effectively jumps
End of assembler dump.
(gdb) x/2x 0x2f88
0x2f88 <dyld_stub_binding_helper+24>: 0x000000ac     0x00000074
```

If you trace through the program, setting a breakpoint on the first and second calls to `dyld_stub_puts` (in their respective offsets in `_main`) will reveal an interesting trick: The first time the stub is called, `dyld_stub_binder` is indeed called, and — through a rather lengthy process — binds all the symbols. The next time, however, `dyld_stub_puts` directly jumps to puts:

```
(gdb) break *0x0000000100000ef3      # as in Listing 4-xyz-a
Breakpoint 1 at 0x100000ef3
(gdb) break *0x0000000100000f02      # as in Listing 4-xyz-a
Breakpoint 2 at 0x100000f02
(gdb) r
Starting program: /Users/morpheus/a
Reading symbols for shared libraries +. done
Breakpoint 1, 0x0000000100000ef3 in main ()
(gdb) disass 0x0000000100000f18      # again, q.v. Listing 4-xyz-a
Dump of assembler code for function dyld_stub_puts:
0x0000000100000f18 <dyld_stub_puts+0>:       jmpq   *0x122(%rip)        # 0x100001040
End of assembler dump.
(gdb) x/g 0x100001040
0x100001040:   0x0000000100000f2a   # the path to dyld_stub_linked ..
(gdb) c
Continuing.
Salve, Munde!
```

```
Breakpoint 2, 0x0000000100000f02 in main ()
(gdb) x/g 0x100001040
0x100001040:   0x00007fff894a5eca   # Now patched to link to puts
```

As the old adage goes, there is no knowledge that is not power. And — if you've followed this long experiment all the way here, the reward is at hand: by patching the stub addresses before the functions are called, it is possible to hook functions. Although `dyld(1)` has a similar mechanism, function interposing, (which is described later in this chapter), patching the table directly is often more powerful.

Shared Library Caches

Another mechanism supported by `dyld` is that of shared library caches. These are libraries that are stored, pre-linked, in one file on the disk. Shared caches are especially important in iOS, wherein most common libraries are cached. The concept is somewhat similar to Android's prelink-map, wherein libraries are pre-linked into fixed offsets in the address space.

If you search on iOS for most libraries, such as `libSystem`, you'll be wasting your time. Although all the binaries have the dependency, the actual file is not present on the file system. To save time on library loading, iOS's `dyld` employs a shared, pre-linked cache, and Apple has moved all the base libraries into it as of iOS 3.0.

In OS X, the `dyld` shared caches are in `/private/var/db/dyld`. On iOS, the shared cache can be found in `/System/Library/Caches/com.apple.dyld`. The cache is a single file, `dyld_shared_cache_armv7`. The OS X shared caches also have an accompanying `.map` file, whereas the iOS one does not.

Figure 4-5 shows the cache header format, which is listed in the `dyld` source files.

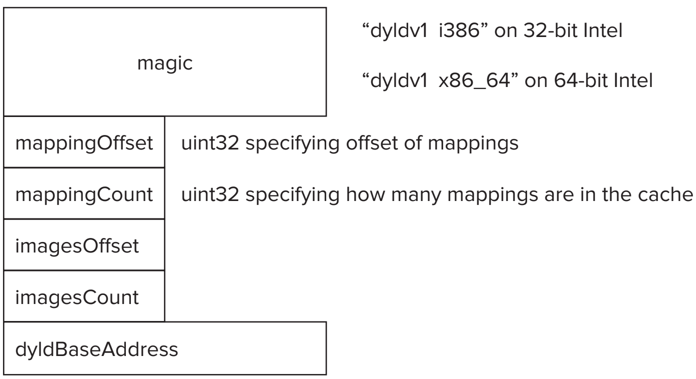


FIGURE 4-5: The dyld cache format

The shared caches, on both OS X on iOS, can grow very large. OS X's contains well over 200 files. iOS's contains over 500(!) and is some 200 MB in size. The jailbreaking community takes special interest in these files and has written various cache "unpackers" to extract the libraries and frameworks inside them. The libraries in their individual form can be found in the `iPhoneOS.platform` directories of the iOS SDK.

Runtime Loading of Libraries

Normally, developers declare the libraries and symbols they will use when they `#include` various headers and, optionally, specify additional libraries to the linker using `-l`. An executable built in this way will not load until all its dependencies are resolved, as you have seen earlier. An alternative, however, is to use the functions supplied in `<dlfcn.h>` to load libraries during runtime. This allows for greater flexibility: The library name needs to be committed to, or known at compile time. In this way, the developer can prepare several libraries and load the most appropriate one based on the features or requirements during runtime. Additionally, if a library load fails, an error code is returned and can be handled by the program.

The API for runtime dynamic library loading in OS X is similar to the one found in POSIX. Its implementation, however, is totally different:

- `dlopen (const char *path)` is used to find and load the library or bundle specified by path.
- `dlopen_preflight(const char *path)` is a Leopard and later extension that simulates the loading process of `dlopen()` but does not actually load anything.
- `dlsym(void *handle, char *sym)` is used to locate a symbol in a handle previously opened by `dlopen()`.
- `dladdr(char *addr, Dl_Info *info)` populates the `DL_Info` structure with the name of the bundle or library residing at address `addr`. This is the same as the GNU extension.
- `dlerror()` is used to provide an error message in case of an error by any of the other functions.

Cocoa and Carbon offer higher-level wrappers for the `dl*` family of functions, as well as a `CFBundle/NSBundle` object, which can be used to load Mach-O bundle files.

One way to check loaded libraries and symbols — from within the program itself — is to use the low-level `dyld` APIs, which are defined in `<mach-o/dyld.h>`. The header also defines a mechanism for callbacks on image load and removal. The `dyld` APIs can also be used alongside the `dl*` APIs (specifically, `dladdr(3)`). This is shown in Listing 4-3:

LISTING 4-3: Listing all Mach-O Images in the process

```
#include <dlfcn.h>        // for dladdr(3)
#include <mach-o/dyld.h>  // for _dyld_ functions

void listImages (void)
{
        // List all mach-o images in a process
        uint32_t i;
        uint32_t ic = _dyld_image_count();

        printf ("Got %d images\n",ic);
        for (i = 0; i < ic; i++)
        {
```

```
            printf ("%d: %p\t%s\t(slide: %p)\n",
                    i,
                    _dyld_get_image_header(i),
                    _dyld_get_image_name(i),
                    _dyld_get_image_slide(i));
        }

}

void add_callback(const struct mach_header* mh, intptr_t vmaddr_slide)
{
  // Using callbacks from dyld, we can get the same functionality
  // of enumerating the images in a binary

  Dl_info info;
  // Should really check return value of dladdr here...
  dladdr(mh, &info);
  printf ("Callback invoked for image: %p %s (slide: %p)\n",
            mh, info.dli_fname, vmaddr_slide);
void main (int argc, char **argv)
{
    // Calling listImages will enumerate all Mach-O objects loaded into
    // our address space, using the _dyld functions from mach-o/dyld.h
        listImages();

    // Alternatively, we can register a callback on add. This callback
    // will also be invoked for existing images at this point.
        _dyld_register_func_for_add_image(add_callback);

}
```

The `listImages()` function is self-contained and can be inserted into any program, given the `dyld.h` file is included (`dyld.h` contains function for checking symbols, as well). If run as is, the program in Listing 4-3 yields the following in Output 4-7:

OUTPUT 4-7: Running the code from Listing 4-3

```
morpheus@Ergo (~) morpheus$ ./lsimg
Got 3 images
0: 0x100000000         /Users/morpheus/./lsimg       (slide: 0x0)
1: 0x7fff87869000      /usr/lib/libSystem.B.dylib    (slide: 0x0)
2: 0x7fff8a2cb000      /usr/lib/system/libmathCommon.A.dylib       (slide: 0x0)

Callback invoked for image: 0x100000000 /Users/morpheus/./lsimg (slide: 0x0)
Callback invoked for image: 0x7fff87869000 /usr/lib/libSystem.B.dylib (slide: 0x0)
Callback invoked for image: 0x7fff8a2cb000 /usr/lib/system/libmathCommon.A.dylib (slide:
0x0)
```

The same, of course, works on iOS, although in this case many more dylibs are preloaded. There is also a non-zero "slide" value, due to Address Space Layout Randomization (ASLR), discussed later in this chapter.

Output 4-8 shows the output of the sample program, on an iOS 5 system. Libraries in bold are new to iOS 5.

OUTPUT 4-8: Running the code from Listing 4-3 on iOS 5

```
root@Podicum (~)# ./lsimg
Got 24 images
0: 0x1000       /private/var/root/./lsimg       (slide: 0x0)
1: 0x304c9000   /usr/lib/libgcc_s.1.dylib       (slide: 0x353000)
2: 0x3660f000   /usr/lib/libSystem.B.dylib      (slide: 0x353000)
3: 0x362c6000   /usr/lib/system/libcache.dylib  (slide: 0x353000)
4: 0x33e60000   /usr/lib/system/libcommonCrypto.dylib   (slide: 0x353000)
5: 0x34a79000   /usr/lib/system/libcompiler_rt.dylib    (slide: 0x353000)
6: 0x30698000   /usr/lib/system/libcopyfile.dylib       (slide: 0x353000)
7: 0x3718d000   /usr/lib/system/libdispatch.dylib       (slide: 0x353000)
8: 0x34132000   /usr/lib/system/libdnsinfo.dylib        (slide: 0x353000)
9: 0x3660d000   /usr/lib/system/libdyld.dylib   (slide: 0x353000)
10: 0x321a3000  /usr/lib/system/libkeymgr.dylib (slide: 0x353000)
11: 0x360b4000  /usr/lib/system/liblaunch.dylib (slide: 0x353000)
12: 0x3473b000  /usr/lib/system/libmacho.dylib  (slide: 0x353000)
13: 0x362f6000  /usr/lib/system/libnotify.dylib (slide: 0x353000)
14: 0x3377a000  /usr/lib/system/libremovefile.dylib     (slide: 0x353000)
15: 0x357c7000  /usr/lib/system/libsystem_blocks.dylib  (slide: 0x353000)
16: 0x36df7000  /usr/lib/system/libsystem_c.dylib       (slide: 0x353000)
17: 0x33ccc000  /usr/lib/system/libsystem_dnssd.dylib   (slide: 0x353000)
18: 0x32aa9000  /usr/lib/system/libsystem_info.dylib    (slide: 0x353000)
19: 0x32ac7000  /usr/lib/system/libsystem_kernel.dylib  (slide: 0x353000)
20: 0x3473f000  /usr/lib/system/libsystem_network.dylib (slide: 0x353000)
21: 0x34433000  /usr/lib/system/libsystem_sandbox.dylib (slide: 0x353000)
22: 0x339d9000  /usr/lib/system/libunwind.dylib (slide: 0x353000)
23: 0x32272000  /usr/lib/system/libxpc.dylib    (slide: 0x353000)

... (callback output is same, and is omitted for brevity) ...
```

Weakly Defined Symbols

An interesting feature in Mac OS is its ability to define symbols as "weak." Typically, symbols are strongly defined, meaning they must all be resolved prior to starting the executable. Failure to resolve symbols in this case would lead to a failure to execute the program (usually in the form of a debugger trap).

By contrast, a weak symbol — which may be defined by specifying `__attribute__((weak_import)` in its declaration — does not cause a failure in program linkage if it cannot be resolved. Rather, the dynamic linker sets it to NULL, allowing the programmer to recover and specify some alternative logic to handle the condition. This is similar to the modus operandi used in dynamic loading (the same effect as `dlopen(3)` or `dlsym(3)` returning `NULL`).

Using `nm` with the `–m` switch will display weak symbols with a "weak" specifier.

dyld Features

Being a proprietary loader, `dyld` offers some unique features, which other loaders can only envy. This section discusses a few of the useful ones.

Two-Level Namespace

Unlike the traditional UN*X `ld`, OS X's `dyld` sports a two-level namespace. This feature, introduced in 10.1, means that symbol names also contain their library information. This approach is better, as it allows for two different libraries to export the same symbol — which would result in link errors in other UN*X. At times, it may be desirable to remove this behavior, restricting a flat namespace (for example, if you want to inject a different library, with the same symbol name, commonly for function hooking). This can be accomplished by setting the `DYLD_FORCE_FLAT_NAMESPACE` environment variable to a non-zero variable. An executable may also force a flat namespace on all its loaded libraries by setting the `MH_FORCE_FLAT` flag in its header.

Function Interposing

Another feature of `dyld` that isn't in the classic `ld` is *function interposing*. The macro `DYLD_INTERPOSE` enables a library to interpose (read: switch) its function implementation for some other function. The snippet in Listing 4-4, from the source of `dyld`, demonstrates this:

LISTING 4-4: DYLD_INTERPOSE macro definition in dyld's include/mach-o/dyld-interposing.h

```
#if !defined(_DYLD_INTERPOSING_H_)
#define _DYLD_INTERPOSING_H_
/*  Example:
 *  static
 *  int
 *  my_open(const char* path, int flags, mode_t mode)
 *  {
 *    int value;
 *    // do stuff before open (including changing the arguments)
 *    value = open(path, flags, mode);
 *    // do stuff after open (including changing the return value(s))
 *    return value;
 *  }
 *  DYLD_INTERPOSE(my_open, open)
 */

#define DYLD_INTERPOSE(_replacment,_replacee) \
   __attribute__((used)) static struct{ const void* replacment; const void* replacee; }
_interpose_##_replacee \
            __attribute__ ((section ("__DATA,__interpose"))) = { (const void*)(unsigned
long)&_replacment, (const void*)(unsigned long)&_replacee };

#endif
```

Interposing simply consists of providing a new `__DATA` section, called `__interpose`, in which the interposing and the interposed are listed, back-to-back. The `dyld` takes care of all the rest.

A good example of a library that uses interposing is OS X's `GuardMalloc` library (a.k.a `/usr/lib/libgmalloc.dylib`). This library replaces `malloc()`-related functionality in `libSystem.B.dylib` with its own implementations, which provide powerful debugging and memory error tracing functionality (try `man libgmalloc`). The library can be forcefully injected into applications, a priori, by setting the `DYLD_INSERT_LIBRARIES` variable. You are encouraged to check the manual page for `libgmalloc(3)` for more details.

Looking at `libgmalloc` with `otool -l`, you will see one of the load commands for the `__DATA` segment sets up a section called `interpose` (Output 4-9).

OUTPUT 4-9: Dumping the interpose section of libgmalloc

```
morpheus@Ergo (/)% otool -lV /usr/lib/libgmalloc.dylib
/usr/lib/libgmalloc:
..
Load command 1
      cmd LC_SEGMENT_64
  cmdsize 632
  segname __DATA
..
Section
  sectname __interpose
   segname __DATA
      addr 0x0000000000005200
      size 0x0000000000000240
    offset 20992
     align 2^4 (16)
    reloff 0
    nreloc 0
      type S_INTERPOSING
attributes (none)
 reserved1 0
 reserved2 0
```

To examine the contents of this section, you can use another Mach-O command, `pagestuff(1)`. This command will show the symbols in the file's logical pages. Output 4-10 is concerned with the interpose-related symbols, which are on logical page 6. (Note that you can also use the `-a` switch for all pages.)

OUTPUT 4-10: Running pagestuff(1) to show interpose symbols in libgmalloc.

```
morpheus@Ergo(/)% pagestuff/usr/lib/libgmalloc.dylib 6
File Page 6 contains contents of section (__DATA,__nl_symbol_ptr) (x86_64)
File Page 6 contains contents of section (__DATA,__la_symbol_ptr) (x86_64)
File Page 6 contains contents of section (__DATA,__const) (x86_64)
File Page 6 contains contents of section (__DATA,__data) (x86_64)
File Page 6 contains contents of section (__DATA,__interpose) (x86_64)
File Page 6 contains contents of section (__DATA,__bss) (x86_64)
File Page 6 contains contents of section (__DATA,__common) (x86_64)
Symbols on file page 6 virtual address 0x5000 to 0x6000
  . . .
  0x0000000000005200 __interpose_malloc_set_zone_name
```

```
0x0000000000005210 __interpose_malloc_zone_batch_free
0x0000000000005220 __interpose_malloc_zone_batch_malloc
0x0000000000005230 __interpose_malloc_zone_unregister
0x0000000000005240 __interpose_malloc_zone_register
0x0000000000005250 __interpose_malloc_zone_realloc
 . . .
0x00000000000053b0 __interpose_free
0x00000000000053c0 __interpose_malloc
```

The interposing mechanism is extremely powerful. Function interposing can easily be used to intercept functions such as `open()` and `close()` — for example, to monitor file system access and even provide a thin layer of virtualization (by redirecting the file during the open operation to some other file, as all other operations that follow use the file descriptor, anyway). Interposing will be used in this book to uncover "behind-the-scenes" operations, as in the following experiment.

Experiment: Using Interposing to Trace malloc()

Listing 4-5 shows a simple application of interposing to provide functionality similar to GLibC's `mtrace (2)` (which OS X does not offer). This function provides a trace of `malloc()` and `free()` operations, printing the pointer value in the operations. In fairness, `libgmalloc` has more powerful features, as do malloc zones (described later in this chapter), but this example demonstrates just how easy implementing those features, as well as others, can be.

LISTING 4-5: GLibC's mcheck-like() functionality, via function interposing

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>
#include <malloc/malloc.h> // for malloc_printf()

// This is the expected interpose structure
typedef struct interpose_s {
    void *new_func;
    void *orig_func;
} interpose_t;

// Our prototypes - requires since we are putting them in
// the interposing_functions, below
void *my_malloc(int size); // matches real malloc()
void  my_free (void *);    // matches real free()

static const interpose_t interposing_functions[] \
    __attribute__ ((section("__DATA, __interpose"))) = {
        { (void *)my_free,  (void *)free  },
        { (void *)my_malloc,  (void *)malloc  },
    };

void *my_malloc (int size)
{
    // In our function we have access to the real malloc() -
    // and since we don't want to mess with the heap ourselves,
```

continues

LISTING 4-5 *(continued)*

```
        // just call it.
        void *returned = malloc(size);

        // call malloc_printf() because the real printf() calls malloc()
        // internally - and would end up calling us, recursing ad infinitum
            malloc_printf ( "+ %p %d\n", returned, size);
            return (returned);
    }

    void my_free (void *freed)
    {
         // Free - just print the address, then call the real free()
         malloc_printf ( "- %p\n", freed);
         free(freed);
    }
```

Note the use of `malloc_printf`, rather than the usual `printf`. This is required because classic `printf()` uses `malloc()` internally, which would lead to a rather messy segmentation fault. In general, when using function interposing on functions provided by `libSystem`, special caution must be taken when relying on `libC` functions, which are in turn provided by `libSystem` itself.

Using this simple library yields clear output, which is easily `grep`-able (matching `+` and `-`, respectively) and enables the quick pinpointing of leaky pointers. To force-load it into an unsuspecting process, we use the `DYLD_INSERT_LIBRARIES` environment variable, as shown in Output 4-11:

OUTPUT 4-11: Running the program from Listing 4-5

```
morpheus@Ergo(~)$ cc -dynamiclib l.c -o libMTrace.dylib -Wall   // compile to dylib
morpheus@Ergo(~)$ DYLD_INSERT_LIBRARIES=libMTrace.dylib ls      // force insert into ls
ls(24346) malloc: + 0x100100020 88
ls(24346) malloc: + 0x100800000 4096
ls(24346) malloc: + 0x100801000 2160
ls(24346) malloc: - 0x100800000
ls(24346) malloc: + 0x100801a00 3312
... // etc.
```

Environment Variables

The OS X `dyld` is highly configurable and can be modified using environment variables. Table 4-9 lists all variables and how they modify the linker's behavior.

TABLE 4-9: DYLD Environment variables and their use

ENVIRONMENT VARIABLE	USE
`DYLD_FORCE_FLAT_NAMESPACE`	Disable two-level namespace of libraries (for `INSERT`). Otherwise, symbol names also include their library name.
`DYLD_IGNORE_PREBINDING`	Disable prebinding for performance testing.

DYLD_IMAGE_SUFFIX	Search for libraries with this suffix. Commonly set to _debug, or _profile so as to load /usr/lib/libSystem.B_debug.dylib or /usr/lib/libSystem.B_profile instead of libSystem.
DYLD_INSERT_LIBRARIES	Force insertion of one or more libraries on program loading — same idea as LD_PRELOAD on UN*X.
DYLD_LIBRARY_PATH	Same as LD_LIBRARY_PATH on UN*X.
DYLD_FALLBACK_LIBRARY_PATH	Used when DYLD_LIBRARY_PATH fails.
DYLD_FRAMEWORK_PATH	As DYLD_LIBRARY_PATH, but for frameworks.
DYLD_FALLBACK_FRAMEWORK_PATH	Used when DYLD_FRAMEWORK_PATH fails.

Additionally, the following control debug printing options in dyld:

- DYLD_PRINT_APIS: Dump dyld API calls (for example dlopen).
- DYLD_PRINT_BINDINGS: Dump symbol bindings.
- DYLD_PRINT_ENV: Dump initial environment variables.
- DYLD_PRINT_INITIALIZERS: Dump library initialization (entry point) calls.
- DYLD_PRINT_LIBRARIES: Show libraries as they are loaded.
- DYLD_PRINT_LIBRARIES_POST_LAUNCH: Show libraries loaded dynamically, after load.
- DYLD_PRINT_SEGMENTS: Dump segment mapping.
- DYLD_PRINT_STATISTICS: Show runtime statistics.

Further detail is well documented in the dyld(1) man page.

Example: DYLD_INSERT_LIBRARIES and Its Resulting Insecurities

Of all the various DYLD options in the last section, none is as powerful as DYLD_INSERT_LIBRARIES. This environment variable is used for the same functionality that LD_PRELOAD offers on UNIX — namely, the forced injection of a library into a newly-created process's address space.

By using DYLD_INSERT_LIBRARIES, it becomes a simple matter to defeat one of Apple's key software protection mechanisms — code encryption. Rather than brute force the decryption, it is trivial to inject the library into the target process and then read the formerly encrypted sections, in clear plaintext. The technique is straightforward and requires only the crafting of such a library. Then, insertion involves only a simple prefixing of the variable to the application to be executed.

Noted researcher Stephan Esser (known more by his handle, i0n1c) has demonstrated this in a very simple library. The library (called dumpdecrypted, part of the Esser's git repository at https://github.com/stefanesser) is force loaded into a Mach-O executable, and then reads the executable, processes its load commands, and simply finds the encrypted section (from the LC_ENCRYPTION_INFO) in its own memory. Because the library is part of process memory, and by that time process memory is decrypted, "decrypting" is a simple matter of copying the address range — which is now plaintext — to disk. The same effect can be achieved from outside the process by using the Mach VM APIs, which this book explores in Chapter 10.

`DYLD_INSERT_LIBRARIES` and the function interposing feature of `dyld` twice played a key feature in the untethered jailbreak ("spirit" and "star") of iOS, up to and including 4.0.*x*, by forcefully injecting a fake `libgmalloc.dylib` into launchd, the very first user mode process. The Trojan library interposes several functions (`unsetenv` and others) used by launchd, injecting a Return-Oriented-Programming (ROP) payload. This means the interposing functions aren't provided by the library (as its code cannot be signed, as is required by iOS), but — rather — by launchd itself. The interposing function of `dyld` was patched in iOS 4.1 to ensure the interposing functions belong to the library, which helps mitigate the attack.

PROCESS ADDRESS SPACE

One of the benefits of user mode is that of isolated virtual memory. Processes enjoy a private address space, ranging from 2-3GB (on iOS), through 4GB (on 32-bit OS X), and up to an unimaginable 16 exabytes on 64-bit OS X. As the previous section has discussed, this address space is populated with segments from the executable and various libraries, using the various `LC_SEGMENT[64]` commands. This section discusses the address space layout, in detail.

The Process Entry Point

As with all standard C programs, executables in OS X have the standard entry point, by default named "main". In addition to the usual three arguments, however — `argc`, `argv` and, `envp` — Mach-O programs can expect a fourth arguments, a char `**` known as "apple."

The "apple" argument, up to and including Snow Leopard, only held a single string – the program's full path, i.e. the first argument of the `execve()` system call used to start it. This argument is used by `dyld(1)` during process loading. The argument is considered to be for internal use only.

Starting with Lion, the "apple" argument has been expanded to a full vector, which now contains two new additional parameters, likewise for internal use only: `stack_guard` and `malloc_entropy`. The former is used by GCC's "stack protector" feature (`-fstack-protector`), and the latter by `malloc`, which uses it to add some randomness to the process address space. These arguments are initialized by the kernel during the Mach-O loading (more on that in Chapter 12) with random values.

The following example (Listing 4-6 and Output 4-12) will display these values, when compiled on Lion, or on iOS 4 and later:

LISTING 4-6: Printing the "apple" argument to Mach-O programs

```
void main (int argc, char **argv, char **envp, char **apple)
{

        int i = 0;
        for (i=0; i < 4; i++)
        printf ("%s\n", apple[i]);

}
```

OUTPUT 4-12: Output of the program from the previous listing

```
Padishah:~ root# ./apple
./apple
stack_guard=0x9e9b3f22f9f1db64
malloc_entropy=0x2b655014ad0fa0c5,0x2f0c9c660cd3fed0
(null)
```

Cocoa applications also start with a standard C main(), although it is common practice to implement the main as a wrapper over NSApplicationMain(), which in turn shifts to the Objective-C programming model.

Address Space Layout Randomization

Processes start up in their own virtual address space. Traditionally, process startup was performed in the same deterministic fashion every time. This meant, however, that the initial process' virtual-memory image was virtually identical for a given program on a given architecture. The problem was further exacerbated by the fact that, even during the process lifetime, most allocations were performed in the same manner, which led to very predictable addresses in memory.

While this offered an advantage for debugging, it provided an even bigger boon for hackers. The primary attack vector hackers use is *code injection*: By overwriting a function pointer in memory, they can subvert program execution to code they provide — as part of their input. Most commonly, the method used to overwrite is a buffer overflow (exceeding the bounds of an array on the stack due to an unchecked memory copy operation), and the overwritten pointer is the function's return address. Hackers have even more creative techniques, however, including subverting `printf()` format strings and heap-based overflows. What's more, any user pointer or even a structured exception handler enables the injection of code. Key here is the ability to determine what to overwrite the pointer with — that is, to reliably determine where the injected code will reside in memory.

The common hacking motto is, to paraphrase java, *exploit once — hack everywhere*. Whatever the vulnerability — buffer overflow, format string attack, or other — a hacker can invest (much) directed effort in dissecting a vulnerable program and finding its address layout, and then craft a method to reliably reproduce the vulnerability and exploit it on similar systems.

Address Space Layout Randomization (ASLR), a technique that is now employed in most operating systems, is a significant protection against hacking. Every time the process starts, the address space is shuffled slightly — shaken, not stirred. The basic layout is still the same, text, data, libraries — as we discuss in the following pages. The exact addresses, however, are different — sufficiently, it is hoped, to thwart the hacker's address guesses. This is done by having the kernel "slide" the Mach-O segments by some random factor.

Leopard was the first version of OS X to introduce address space layout randomization, albeit in a very limited form. The randomization only occurred on system install or update, and randomized only the loading of libraries. Snow Leopard made some improvements, but the heap and stack were both predictable — and the assigned address space persisted across reboots.

Lion is the first version of OS X to support full randomization in user space — including the text segments. Lion provides 16-bit randomization in the text segments and up to 20-bit randomization elsewhere, per invocation of the program. The 64-bit Mach-O binaries are flagged with `MH_PIE`

(`0x00200000`), specifying to the kernel that the binary should be loaded at a random address. 32-bit programs still have no randomization. Likewise, iOS 4.3 is the first version of iOS to introduce ASLR in user space. For Apple, doing so in iOS is even more important, as code injection is the underlying technique behind jailbreaking the various i-Devices. ASLR can be selectively disabled (by setting `_POSIX_SPAWN_DISABLE_ASLR` in call to `posix_spawnattr_setflags()`, if using `posix_spawn()` to create the process), but is otherwise enabled by default.

Mountain Lion further improves on its predecessors and introduces ASLR into the kernel space. A new system call, `kas_info` (#439) is offered to obtain kernel address space information. At the time of this writing, iOS does not offer kernel space randomization. It is more than likely, however, that the next update of iOS will do so as well, in an attempt at thwarting jailbreakers from injecting code into the iOS kernel. The code has also been compiled with aggressive stack-checking logic in many function epilogs, just in case.

It should be noted that ASLR, while a significant improvement, is no panacea. (Neither, for that matter, is the NX protection, discussed earlier.) Hackers still find clever ways to hack. In fact, the now infamous "Star 3.0" exploit, which jailbroke iOS 4.3 on the iPad 2, defeated ASLR. This was done by using a technique called "Return-Oriented Programming," (ROP), in which the buffer overflow corrupts the stack to set up entire stack frames, simulating calls into `libSystem`. The same technique was used in the iOS 5.0.1 "corona" exploit, which has been successfully used to break all Apple devices, including the latest and greatest iPhone 4S.[5]

The only real protection against attacks is to write more secure code and subject it to rigorous code reviews, both automated and manual.

32-Bit (Intel)

While no longer the default, 32-bit address spaces are still possible — in older programs or by specifically forcing 32-bit (compiling with `-arch i386`). The 32-bit address space is capped at 4 GB (232 = 4,294,967,296 bytes). Unlike other operating systems, however, all the 4 GB is accessible from user space — there is no reservation for kernel space.

> *Windows traditionally reserves 2 GB (`0x80000000-`) and Linux 1 GB (`0xC0000000-`) for Kernel space. Even though this memory is technically addressable by the process, trying to access it from user mode generates a general protection fault, and usually leads to a segmentation fault, which kills the process. OS X (in 32-bit mode) uses a different approach, assigning the kernel its own 4 GB address space, thereby freeing the top 1 GB for user space. So instead of Windows' 2/2 and Linux's 3/1, OS X gives a full 4 GB to both kernel and user spaces. This comes at a cost, however, of a full address space switch (CR3 change and TLB flush). This is no longer the case in 64-bit, or on iOS.*

64-Bit

64 bits allow for a huge address space of up to 16 exabytes (that is, 16 giga-gigabytes). While this is never actually needed in practice (and, in fact, most hardware architectures support only 48–52

bits for addressing), it does allow for a sparser address space. The layout is still essentially the same, except that now segments are much farther apart from one another.

It should be noted, that even 64-bit is not true 64-bit. Due to the overhead associated with virtual to physical address translation, the Intel architecture uses only 48 bits of the virtual address. This is a hardware restriction, which is imposed also on Linux and Windows. The highest accessible region of the user memory space, therefore, lies at `0x7FFF-FFFF-FFFF`.

In 64-bit mode, there is such a huge amount of memory available anyway that it makes sense to follow the model used in other operating systems, namely to map the kernel's address space into each and every process. This is a departure from the traditional OS X model, which had the kernel in its own address space, but it makes for much faster user/kernel transition (by sharing CR3, the control register containing the page tables).

32-Bit (iOS)

The iOS address space is even more restricted than its 32-bit Intel counterpart. For starters, unlike 32-bit OS X, the kernel is mapped to `0xC0000000` (iOS 3), or `0x80000000` (iOS 4 and 5), consuming a good 1–2 GB of the space. Further, addresses over `0x30000000` are reserved for the various libraries and frameworks.

A simple program to allocate 1 MB at a time will fail sooner, rather than later. For example, on an iPad, the program croaks at about 80 MB:

```
Root@Padishah:~ root# ./a
a(12236) malloc: *** mmap(size=1048576) failed (error code=12)
*** error: can't allocate region
*** set a breakpoint in malloc_error_break to debug
a(12236) malloc: *** mmap(size=16777216) failed (error code=12)
*** error: can't allocate region
*** set a breakpoint in malloc_error_break to debug
She won't hold, Cap'n! Total allocation was 801112064 MB
```

This low limit makes perfect sense, if one takes into account the fact the there is *no swap space on* i-Devices. Swap and flash storage do not get along very well because of the former's need for many write/delete operations and the latter's limitations in doing so. So, while on a hard drive swap raises no issues (besides the unavoidable hit on performance), on a mobile device swap is not an option.

As a consequence, virtual memory on mobile devices is, by its nature, limited. Tricks such as implicit sharing can give the illusion of more space than exists on a system-wide level, but any single process may not consume more than the available RAM, which is less than the device's physical RAM because of memory used by other processes and by the kernel itself.

General Address Space Layout

Because of ASLR, the address space of processes is very fluid. But while exact addresses may "slide" by some small random offsets, the rough layout remains the same.

The memory segments are as follows:

- `__PAGEZERO`: On 32-bit systems, this is a single page (4 KB) of memory, with all of its access permissions revoked. On 64-bit systems, this corresponds to the entire 32-bit address space — i.e. the first 4 GB. This is useful for trapping NULL pointer references (as NULL is really "0"), or integer-as-pointer references (as all values up to 4,095 in 32-bit, or 4 GB in 64-bit, fall within this page). Because access permissions — read, write, and execute — are all revoked, any attempt to dereference memory addresses that lie within this page will trigger a hardware page fault from the MMU, which in turn leads to a trap, which the kernel can trap. The kernel will convert the trap to a C++ exception or a POSIX signal for a bus error (SIGBUS).

> `PAGEZERO` *is not meant to be used by the process, but it has become somewhat of a cozy breeding ground for malicious code. Attackers wishing to infect a Mach-O with "additional" code often find* `PAGEZERO` *to be convenient for that purpose.* `PAGEZERO` *is normally not part of the file, (its* `LC_SEGMENT` *specified filesize is 0), there is no strict requirement this be the case.*

- `__TEXT`: This is the program code. As in all operating systems, text segments are marked as `r-x`, meaning read-only and executable. This not only helps protect the binary from modification in memory, but optimizes memory usage by making the section shareable. This way, multiple instances of the same program use up only one `__TEXT` copy. The text segment usually contains several sections, with the actual code in `_text`. It can also contain other read-only data, such as constants and hard-coded strings.
- `__LINKEDIT`: For use by `dyld`, this section contains tables of strings, symbols, and other data.
- `__IMPORT`: Used for the import tables on i386 binaries.
- `__DATA`: Used for readable/writable data.
- `__MALLOC_TINY`: For allocations of less than page size.
- `__MALLOC_SMALL`: For allocations of several pages.
- `__MALLOC_LARGE`: For allocations of over 1 MB.

Another segment which doesn't show up in vmmap is the *commpage*. This is a set of pages exported by the kernel to all user mode processes, similar in concept to Linux's `vsyscall` and `vdso`. The pages are shared (read-only) in all processes at a fixed address: `0xffff0000` in i386, `0x7fffffe00000` in x86_64, and `0x40000000` in ARM. They hold various CPU and platform related functions.

The commpage is largely a relic of the days of Mach on the PPC, wherein it was used frequently. Apple is phasing it out, with scant remnants, like libSystem using it to accelerate `gettimeofday()` and (up until Lion and iOS 5) `pthread_mutex_lock()`. Code in the commpage has the unique property that it can be made temporarily non-preemptible, if it resides in the Preemption Free Zone (PFZ). This is discussed further in Chapters 8 and 11.

We discuss the internals of memory management, from the user mode perspective, next. The kernel mode perspective is discussed in Chapter 12. Mach-O segment and section loading is covered in Chapter 13.

Experiment: Using vmmap(1) to Peek Inside a Process's Address Space

Using the `vmmap(1)` command, you can view the memory layout of a process. Carrying the previous experiment further, you use `vmmap -interleaved`, which dumps the address space in a clear way. The `-interleaved` switch sorts the output by address, rather than readable/writable sections.

Consider the following program in Listing 4-7:

LISTING 4-7: A sample program displaying its own address space

```
#include <stdlib.h>
int global_j;
const int ci = 24;
void main (int argc, char **argv)
{
        int local_stack = 0;
        char *const_data = "This data is constant";
        char *tiny = malloc (32);            /* allocate 32 bytes */
        char *small = malloc (2*1024);       /* Allocate 2K */
        char *large = malloc (1*1024*1024);  /* Allocate 1MB */

        printf ("Text is %p\n", main);
        printf ("Global Data is %p\n", &global_j);
        printf ("Local (Stack) is %p\n", &local_stack);
        printf ("Constant data is %p\n",&ci );
        printf ("Hardcoded string (also constant) are at %p\n",const_data );
        printf ("Tiny allocations from %p\n",tiny );
        printf ("Small allocations from %p\n",small );
        printf ("Large allocations from %p\n",large );
        printf ("Malloc (i.e. libSystem) is at %p\n",malloc );
        sleep(100); /* so we can use vmmap on this process before it exits */
}
```

Compiling it on a 32-bit system (or with `-arch i386`) and running it will yield the results shown in Figure 4-6.

The `vmmap(1)` output shows the region names, address ranges, permissions (current and maximum), and the name of the mapping (usually the backing Mach-O object), if any.

For example, `__PAGEZERO` is exactly 4 KB (`0x00000000–0x00001000`) and is empty (`SM=NUL`) and set with no permissions (`current permissions: ---`, `max permissions: ---`).

Other regions are defined as `COW` — meaning copy-on-write. This makes them shareable, as long as they are not modified — that is, up to the point where one of the sharing processes requests to write data to that page. Because that would mean that the two processes would now be seeing different data, the writing process triggers a page fault, which gets the kernel to copy that page.

```
Ergo:~ morpheus$ cc a.c -o a -arch i386
Ergo:~ morpheus$ ./a &
[1] 6331
Ergo:~ morpheus$ Text is 0x1d72
Global Data is 0x2040
Local (Stack) is 0xbffffb1c
Constant data is 0x1e84
Hardcoded string (also constant) are at 0x1e88
Tiny allocations from 0x100130
Small allocations from 0x800000
Large allocations from 0x200000
Malloc (i.e. libSystem) is at 0x946ba246
```

```
==== regions for process 6396 (non-writable and writable regions are interleaved)
__PAGEZERO           00000000-00001000 [    4K] ---/--- SM=NUL /Users/morpheus/a
__TEXT               00001000-00002000 [    4K] r-x/rwx SM=COW /Users/morpheus/a
__DATA               00002000-00003000 [    4K] rw-/rwx SM=PRV /Users/morpheus/a
__LINKEDIT           00003000-00004000 [    4K] r--/rwx SM=COW /Users/morpheus/a
STACK GUARD          00004000-00005000 [    4K] ---/rwx SM=NUL
MALLOC (admin)       00005000-00006000 [    4K] rw-/rwx SM=COW
STACK GUARD          00006000-00008000 [    8K] ---/rwx SM=NUL
MALLOC (admin)       00008000-00013000 [   44K] rw-/rwx SM=COW
STACK GUARD          00013000-00015000 [    8K] ---/rwx SM=NUL
MALLOC (admin)       00015000-00020000 [   44K] rw-/rwx SM=COW
STACK GUARD          00020000-00021000 [    4K] ---/rwx SM=NUL
MALLOC (admin)       00021000-00022000 [    4K] r--/rwx SM=COW
MALLOC_LARGE         00022000-00023000 [    4K] rw-/rwx SM=COW DefaultMallocZone_0x5000
MALLOC_TINY          00100000-00200000 [ 1024K] rw-/rwx SM=COW DefaultMallocZone_0x5000
MALLOC_LARGE         00200000-00300000 [ 1024K] rw-/rwx SM=NUL DefaultMallocZone_0x5000
MALLOC_SMALL         00800000-01000000 [ 8192K] rw-/rwx SM=COW DefaultMallocZone_0x5000
__TEXT               8fe00000-8fe42000 [  264K] r-x/rwx SM=COW /usr/lib/dyld
__DATA               8fe42000-8fe6f000 [  180K] rw-/rwx SM=COW /usr/lib/dyld
__IMPORT             8fe6f000-8fe70000 [    4K] rwx/rwx SM=COW /usr/lib/dyld
__LINKEDIT           8fe70000-8fe84000 [   80K] r--/rwx SM=COW /usr/lib/dyld
__TEXT               946b7000-9485f000 [ 1696K] r-x/r-x SM=COW /usr/lib/libSystem.B.dylib
__TEXT               9496f000-94973000 [   16K] r-x/r-x SM=COW
```

FIGURE 4-6: Virtual address space layout of a 32-bit process

On a 64-bit system, the map is similar:

OUTPUT 4-13: Address space layout of a 64-bit binary

```
Listing …: Address space layout of a 64-bit binary

Virtual Memory Map of process 16565 (a)
Output report format:  2.2  -- 64-bit process

==== regions for process 16565  (non-writable and writable regions are interleaved)
__TEXT                 0000000100000000-0000000100001000 [    4K] r-x/rwx SM=COW
                                                              /Users/morpheus/a
__DATA                 0000000100001000-0000000100002000 [    4K] rw-/rwx SM=PRV
                                                              /Users/morpheus/a
__LINKEDIT             0000000100002000-0000000100003000 [    4K] r--/rwx SM=COW
                                                              /Users/morpheus/a
MALLOC guard page      0000000100003000-0000000100004000 [    4K] ---/rwx SM=NUL
MALLOC metadata        0000000100004000-0000000100005000 [    4K] rw-/rwx SM=COW
MALLOC guard page      0000000100005000-0000000100007000 [    8K] ---/rwx SM=NUL
MALLOC metadata        0000000100007000-000000010001c000 [   84K] rw-/rwx SM=COW
MALLOC guard page      000000010001c000-000000010001e000 [    8K] ---/rwx SM=NUL
MALLOC metadata        000000010001e000-0000000100033000 [   84K] rw-/rwx SM=COW
MALLOC guard page      0000000100033000-0000000100034000 [    4K] ---/rwx SM=NUL
MALLOC metadata        0000000100034000-0000000100035000 [    4K] r--/rwx SM=COW
MALLOC_LARGE metadata  0000000100035000-0000000100036000 [    4K] rw-/rwx SM=COW
                                                 DefaultMallocZone_0x100004000
MALLOC_TINY            0000000100100000-0000000100200000 [ 1024K] rw-/rwx SM=COW
                                                 DefaultMallocZone_0x100004000
MALLOC_LARGE (reserved 0000000100200000-0000000100300000 [ 1024K] rw-/rwx SM=NUL
                                                 DefaultMallocZone_0x100004000
MALLOC_SMALL           0000000100800000-0000000101000000 [ 8192K] rw-/rwx SM=COW
                                                 DefaultMallocZone_0x100004000
STACK GUARD            00007fff5bc00000-00007fff5f400000 [ 56.0M] ---/rwx SM=NUL
                                                      stack guard for thread 0
Stack                  00007fff5f400000-00007fff5fbff000 [ 8188K] rw-/rwx SM=ZER
                                                                      thread 0
Stack                  00007fff5fbff000-00007fff5fc00000 [    4K] rw-/rwx SM=COW
                                                                      thread 0
__TEXT                 00007fff5fc00000-00007fff5fc3c000 [  240K] r-x/rwx SM=COW
                                                                  /usr/lib/dyld
__DATA                 00007fff5fc3c000-00007fff5fc7b000 [  252K] rw-/rwx SM=COW
                                                                  /usr/lib/dyld
__LINKEDIT             00007fff5fc7b000-00007fff5fc8f000 [   80K] r--/rwx SM=COW
                                                                  /usr/lib/dyld
__DATA                 00007fff701b2000-00007fff701d5000 [  140K] rw-/rwx SM=COW
                                                    /usr/lib/libSystem.B.dylib
__TEXT                 00007fff8111b000-00007fff812dd000 [ 1800K] r-x/r-x SM=COW
                                                    /usr/lib/libSystem.B.dylib
__TEXT                 00007fff87d0f000-00007fff87d14000 [   20K] r-x/r-x SM=COW
                                          /usr/lib/system/libmathCommon.A.dylib
__LINKEDIT             00007fff8a886000-00007fff8cc7e000 [ 36.0M] r--/r-- SM=COW
                                          /usr/lib/system/libmathCommon.A.dylib
. . .
```

Cydia packages for iOS do not have `vmmap(1)`, but — as open source — it can be compiled for iOS. Alternatively, the same information can be obtained using `gdb`. By attaching to a process in `gdb`, you can issue one of three commands, which would give you the following information:

- Info mach-regions
- Maintenance info section
- Show files

The same information can be obtained by walking through the load commands (`otool -l`)

> *Later in this book, we discuss Mach virtual memory and regions, and show an actual implementation of* `vmmap(1)` *from the ground up, using the underlying Mach trap,* `mach_vm_region`*. You will also be able to use it on iOS.*

PROCESS MEMORY ALLOCATION (USER MODE)

One of the most important aspects of programming is maintaining memory. All programs rely on memory for their operation, and proper memory management can make the difference between a fast, efficient program, and poor and faulty one.

Like all systems, OS X offers two types of memory allocations — stack-based and heap-based. Stack-based allocations are usually handled by the compiler, as it is the program's automatic variables that normally populate the stack. Dynamic memory is normally allocated on the heap. Note, that these terms apply only in user mode. At the kernel level, neither user heap nor stack exists. Everything is reduced to pages. The following section discusses only the user mode perspective. Kernel virtual memory management is itself deserving of its own chapter. Apple also provides documentation about user mode memory allocation.[6]

The alloca() Alternative

Although the stack is, traditionally, the dwelling of automatic variables, in some cases a programmer may elect to use the stack for dynamic memory allocation, using the surprisingly little known `alloca(3)`. This function has the same prototype as `malloc(3)`, with the one notable exception — that the pointer returned is on the stack, and not the heap.

From an implementation perspective, `alloca(3)` is preferable to `malloc(3)` for two main reasons:

- The stack allocation is usually nothing more than a simple modification of the stack pointer register. This is a much faster method than walking the heap and trying to find a proper zone or free list from which to obtain a chunk. Additionally, the stack memory pages are already resident in memory, mitigating the concern of page faults — which, while unnoticeable in user mode, still have a noticeable effect on performance.
- Stack allocation automatically clears up when the function allocating the space returns. This is assured by the function prolog (which usually sets up the stack frame by saving the stack

pointer on entry), and epilog (which resets the stack pointer to its value from the entry). This makes dreaded memory leaks a non-issue. Given how happily programmers `malloc()` — yet how little they `free()` — addressing memory leaks automatically is a great idea.

All these advantages, however, come at a cost — and that is of stack space. Stack space is generally far more limited than that of the heap. This makes `alloca(3)` suitable for small allocations of relatively short-lived functions, but inadequate for code paths that involve deep nesting (or worse, recursion). Stack space can be controlled by `setrlimit(3)` on `RLIMIT_STACK` (or, from the command line, `ulimit(1) -s`). If the stack overflows, `alloca(3)` will return `NULL` and the process will be sent a `SIGSEGV`.

Heap Allocations

The heap is a user-mode data structure maintained by the C runtime library, which frees the program from having to directly allocate pages. The term "heap" originated from the data structure used — a binary heap — although today's heaps are far more complex. What's more, every operating system has its own preference for heap management, with Windows, Linux, and Darwin taking totally different approaches. The approach taken by Darwin's LibC is especially suited for use by its biggest client, the Objective-C runtime.

Darwin's `LibC` uses a special algorithm for heap allocation, based on allocation *zones*. These are the `tiny, small, large` and `huge` areas shown in the output of vmmap(1) in Figure 4-6 and Output 4-13. Each zone has its own allocator with different semantics, which are optimized for the allocation size. Prior to Snow Leopard, the *scalable allocator* was used, which is now superseded by the *magazine allocator*. The allocation logic of both allocators is fairly similar, but allocation magazines are thread-specific, and therefore less prone to locking or contention. The magazine allocator also does away with the huge zones. The Foundation.Framework encapsulates malloc zones with `NSZones`.

New zones can be added fairly easily (by calling `NSCreateZone/malloc_create_zone,` or directly initializing a `malloc_zone_t` and calling `malloc_zone_register`), and malloc can be redirected to allocated from a specific zone (by calling `malloc_zone_malloc`). Memory management functions in a zone may be hooked. For debugging purposes, however, it suffices to use the `introspect` structure and provide user-defi ned callbacks. As shown in Figure 4-7, introspection allows detailed debugging of the zone, including presenting its usage, statistics, and all pointers. The `<malloc/malloc.h>` header provides many other functions which are useful for debugging and diagnostics, the most powerful of which is `malloc_get_all_zones()`, which (unlike most others) can be called from outside the process for external memory monitoring.

Snow Leopard and later support *purgeable* zones, which underlie `libcache` and Cocoa's `NSPurgeableData`. Lion further adds support for discharged pointers and VM pressure relief. VM pressure is a concept in XNU (more accurately, in Mach), which signals to user mode that the system is low on RAM (i.e. too many pages are resident). The pressure relief mechanism then kicks in and attempts to automatically free a supplied goal of bytes. RAM is especially important in iOS, where the VM pressure mechanism is tied to Jetsam, a mechanism similar to Linux's Out-Of-Memory (OOM) killer. Most objective-C developers interface with the mechanism when they implement a `didReceiveMemoryWarning`, to free as much memory as possible and pray they will not be ruthlessly killed by Jetsam.

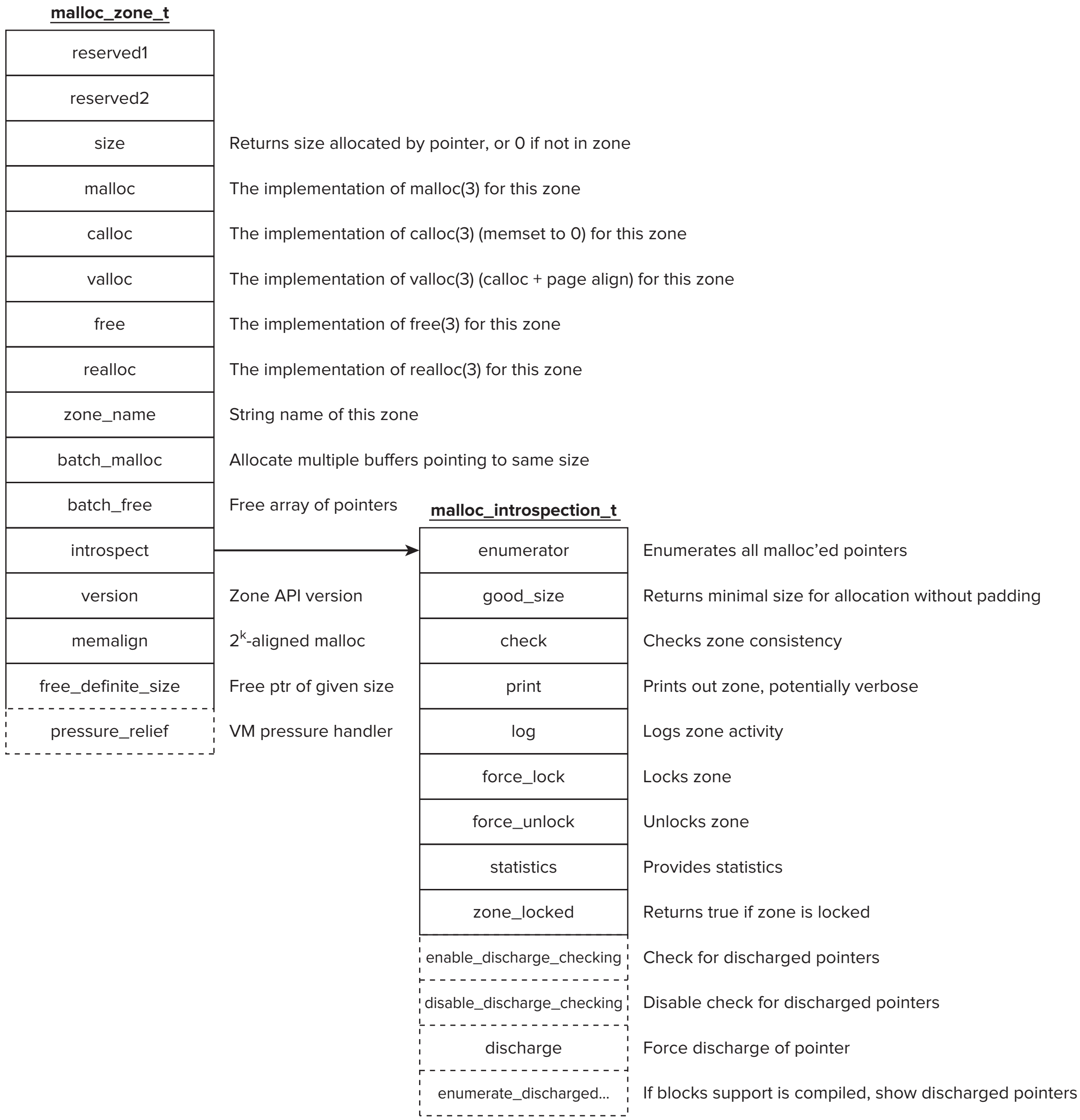


FIGURE 4-7: The structure of malloc zone objects

Virtual Memory — The sysadmin Perspective

It is assumed the reader is no stranger to virtual memory and the page lifecycle. Because the nomenclature used differs slightly with each operating system, however, the following serves both to refresh and adapt the terms to those used in Mach-dom:

Page Lifecycle

Physical memory pages spend their lives in one of several states, as shown in Table 4-10 and Figure 4-8

TABLE 4-10: Physical Page States

PAGE STATE	APPLIES WHEN
Free	Physical page is not used for any virtual memory page. It may be instantly reclaimed, if the need arises.
Active	Physical page is currently used for a virtual memory page and has been recently referenced. It is not likely to be swapped out, unless no more inactive pages exist. If the page is not referenced in the near future, it will be deactivated.
Inactive	Physical page is currently used for a virtual memory page but has not been recently referenced by any process. It is likely to be swapped out, if the need arises. Alternatively, if the page is referenced at any time, it will be reactivated.
Speculative	Pages are speculatively mapped. Usually this is the result of a guessed allocation about possibly needing the memory, but it is not active yet (nor really inactive, as it might be accessed shortly).
Wired down	Physical page is currently used for a virtual memory page but cannot be paged out, regardless of referencing.

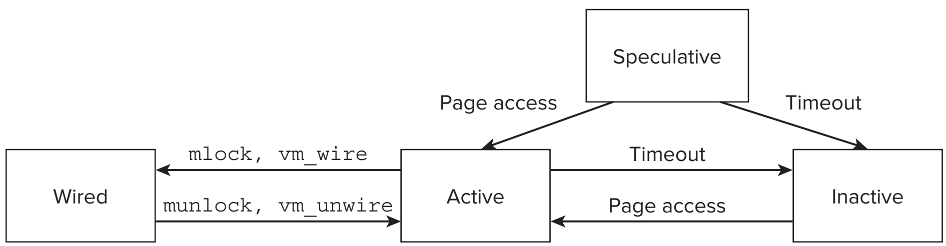


FIGURE 4-8: Physical page state transitions

vm_stat(1)

The `vm_stat(1)` utility (not to be confused with the UNIX `vmstat`, which is different) displays the in-kernel virtual memory counters. The Mach core maintains these statistics (in a `vm_statistics64` struct), and so this utility simply requests them from the kernel and prints them out (how exactly it does so is shown in a more detailed example in Chapter 10). Its output looks something like the following:

```
morpheus@ergo (/)$ vm_stat
Mach Virtual Memory Statistics: (page size of 4096 bytes)
Pages free:                          5366.
Pages active:                      440536.
Pages inactive:                    267339.
Pages speculative:                  19096.
```

```
Pages wired down:                    250407.
"Translation faults":              18696843.
Pages copy-on-write:                 517083.
Pages zero filled:                  9188179.
Pages reactivated:                    98580.
Pageins:                             799179.
Pageouts:                             42569.
```

The vm_stat utility lists the counts of pages in various lifecycle stages, and additionally displays cumulative statistics since boot, which include:

- **Translation faults:** Page fault counts
- **Pages copy-on-write:** Number of pages copied as a result of a COW fault
- **Pages zero filled:** Pages that were allocated and initialized
- **Pageins:** Fetches of pages from
- **Pageouts:** Pushes of pages to swap

sysctl(8)

The sysctl(8) command, which is a UNIX standard command to view and toggle kernel variables, can also be used to manage virtual memory settings. Specifically, the vm namespace holds the following variables shown in Table 4-11:

TABLE 4-11: sysctl variables to control virtual memory settings

VARIABLE	USED FOR
vm.allow_stack_exec	Executable stacks. Default is 0.
vm.allow_data_exec	Executable heaps. Default is 1.
vm.cs_*	Miscellaneous settings related to code signing. These are discussed under "Code Signing" in Chapter 12.
vm.global_no_user_wire_amount vm.global_user_wire_limit vm.user_wire_limit	Global and per user settings for wired (mlocked) memory.
vm.memory_pressure	Is system low on virtual memory?
kern.vm_page_free_target page_free_wanted	Target number of pages that should always be free.
shared_region_*	Miscellaneous settings pertaining to shared memory regions.

dynamic_pager(8)

OS X is unique in that, following Mach, swap is not managed directly at the kernel level. Instead, a dedicated user process, called the dynamic_pager(8) handles all swapping requests. It is started at boot by launchd, from a property list file called com.apple.dynamic_pager.plist (found amidst the other startup programs, in /System/Library/LaunchDaemons, as discussed in Chapter 6). It is possible to disable swapping altogether, by unloading (or removing) the property list from launchd, but this is not recommended.

The dynamic_pager is responsible for managing the swap space on the disk. The launchd starts the pager with the swap set to /private/var/vm/swapfile. This can be changed with the -F switch, to specify another file path and prefix. Other settings the pager responds to are shown in Table 4-12:

TABLE 4-12: Switches used by dynamic_pager(8)

SWITCH	USED FOR
-F	Path and prefix of swap files. Default set by launchd is /private/var/vm/swapfile.
-S	File size, in bytes, for additional swap file.
-H	High water mark: If there are fewer pages free than this, swap files are needed.
-L	Low water mark: If there are more pages free than this, the swap files may be coalesced. For obvious reasons, it must hold that -L >= -S + H, as the coalescing will free a swap file of S bytes.

The dynamic_pager has its own property list file (Library/Preferences/com.apple.virtualMemory.plist). The only key defined, at present, is a Boolean — prior to Lion, useEncryptedSwap (default, no), and as of Lion, disableEncryptedSwap (default, yes). Because the encrypted swap feature follows the hard-coded default (true for laptops, false for desktops/servers), this file should be created if the default is to be changed — which may be accomplished with the defaults(1) command.

The above mentioned sysctl(8) command can be used to view (among other things) the swap utilization, by vm.swapusage.

THREADS

Processes as we know them are a thing of the past. Modern operating systems, OS X and iOS included, see only threads. Apple raises the notch a few levels higher by supporting far richer APIs than other operating systems, to facilitate the work with multiple threads. This section reviews the ideas behind threads, then discusses the OS X/iOS-specific features.

Unraveling Threads

Originally, UNIX was designed as a multi-processed operating system. The process was the fundamental unit of execution, and the container of the various resources needed for execution: virtual memory, file descriptors, and other objects. Developers wrote sequential programs, starting with the entry point — main — and ending when the main function returned (or when exit(2) was called. Execution was thus serialized, and easy to follow.

This, however, soon proved to be too rigid an approach, offering little flexibility to tasks which needed to be executed concurrently. Chief among those was I/O: calls such as read(2) and `write(2)` could block indefinitely — especially when performed on sockets. A blocking read meant that socket code, for example, could not keep on sending data while waiting to read. The `select(2)` and `poll(2)` system calls provided somewhat of workaround, by enabling a process to put all its file descriptors into one array, thereby facilitating I/O multiplexing. Coding in this way is neither scalable nor very efficient, however.

Another consideration was that most processes block on I/O sooner rather than later. This means that a large portion of the process timeslice is effectively lost. This greatly impacts performance, because the cost of process context switching is considered expensive.

Threads were thus introduced, at the time, primarily as a means of maximizing the process timeslice: By enabling multiple threads, execution could be split into seemingly concurrent subtasks. If one subtask would block, the rest of the timeslice could be allocated to another subtask. Additionally, polling would no longer be required: One thread could simply block read and wait for data indefinitely, while another would be free to keep on doing other things, such as `write(2)`, or any other operation.

CPUs at the time were still limited, and even multi-threaded code could only run one thread at a time. The thread preemption of a process was a smaller-scale rendition of the preemptive multitasking the system did for processes. At that point, it started making more sense for most operating systems to switch their scheduling policies to threads, rather than processes. The cost of switching between threads is minimal — merely saving and restoring register state. Processes, by contrast, involve switching the virtual memory space as well, including low-level overhead such as flushing caches, and the Translation Lookaside Buffer (TLB).

With the advent of multi-processor, and — in particular — multi-core architectures, threads took a life of their own. Suddenly, it became possible to actually run two threads in a truly concurrent manner. Multiple cores are especially hospitable to threads because cores share the same caches and RAM – facilitating the sharing of virtual memory between threads. Multiple processors, by contrast, can actually suffer due to non-uniform memory architecture, and cache coherency considerations.

UN*X systems adopted the POSIX thread model. Windows chose its own API. Mac OS X naturally followed in the UN*X footsteps, but has taken a few steps further with its introduction of higher-level APIs — those of Objective-C and (as of Snow Leopard) — the Grand Central Dispatcher.

POSIX Threads

The POSIX thread model is effectively the standard threading API in all systems but Windows (which clings to the Win32 Threading APIs). OS X and iOS actually support more of `pthread` than other operating systems. A simple man `-k pthread` will reveal the extent of functions supported, as will a look at `<pthread.h>`.

The `pthread` APIs, as in other systems, are mapped to native system calls which direct the kernel to create the threads. Table shows this mapping. Unlike other operating systems, XNU also contains specific system calls meant to facilitate `pthread`'s synchronization objects to be managed in kernel mode (collectively known as `psynch`). This makes thread management more efficient, than leaving the objects in user mode. These calls, however, are not necessarily enabled (being conditionally compiled in the kernel). `libSystem` dynamically checks, and — if supported — uses internal new `_pthread_*` functions in place of the "old" `pthread` ones (e.g. `new_pthread_mutex_init`, `new_pthread_rwlock_rdlock`, and the like). Note that the psynch APIs (shown in table 4-13) aren't necessarily supported.

TABLE 4-13: Some pthread APIs and their corresponding system calls in XNU.

PTHREAD API	UNDERLYING SYSTEM CALL
`pthread_create`	`bsdthread_create`
`pthread_sigmask`	`pthread_sigmask`
`pthread_cancel`	`pthread_markcancel`
`pthread_rwlock_rdlock`	`psynch_rw_rdlock`
`pthread_cond_signal`	`psynch_cvsignal`
`pthread_cond_wait`	`psynch_cvwait`
`pthread_cond_broadcast`	`psynch_cvbroad`

Grand Central Dispatch

Snow Leopard introduces a new API for multi-processing called the Grand Central Dispatch (GCD). Apple promotes this API as an alternative to threads. This presents a paradigm shift: Rather than think about threads and thread functions, developers are encouraged to think about functional blocks. GCD maintains an underlying thread pool implementation to support the concurrent and asynchronous execution model, relieving the developer from the need to deal with concurrency issues, and potential pitfalls such as deadlocking. This mechanism can also deal with other asynchronous notifications, such as signals and Mach messages. Lion further extends this to support asynchronous I/O. Another advantage of using GCD is that the system automatically scales to the number of available logical processors.

The developer implements the work units as either functions, or functional block. A functional block, quite like a C block, is enclosed in curly braces, but — like a C function — can be pointed to (albeit with a caret `(^)` rather than an asterisk `(*))`. The dispatch APIs can work well with either.

Work is performed by one of several dispatch queues:

- **The global dispatch queues**: are available to the application by calling `dispatch_get_global_queue()`, and specifying the priority requested: `DISPATCH_QUEUE_PRIORITY_DEFAULT`, `_LOW`, or `_HIGH`.
- **The main dispatch queue**: which integrates with Cocoa applications' run loop. It can be retrieved by a call to `dispatch_get_main_queue()`.

- **Custom queues:** Created manually by a call to `dispatch_queue_create()`, can be used to obtain greater control over dispatching. These can either be serial queues (in which tasks are executed FIFO) or concurrent ones.

The APIs of the Grand Central Dispatch are all declared in `<dispatch/dispatch.h>`, and implemented in `libDispatch.dylib`, which is internal to `libSystem`. The APIs themselves are built over `pthread_workqueue` APIs, which XNU supports with its workq system calls (#367, #368). Chapter 14 discusses these system calls in more detail. A good documentation on the user mode perspective can be found in Apple's own GCD Reference[7] and Concurrency Programming Guide.[8] It should be noted that Objective-C further wraps these APIs by those exposed by the `NSOperation`-related objects.

REFERENCES

1. Apple Technical Note — TN2206: "Mac OS X Code Signing In Depth"
2. NeXTSTEP 3.3 DevTools documentation, Chapter 14, "Mach Object Files" — Documents the original Mach-O format (which remains largely unchanged in OS X).
3. Apple Developer: Mach-O Programming Topics — Basic architecture and loading
4. Apple Developer: Mac OS X ABI Mach-O File Format Reference — Discussion on load commands
5. Dream Team — Absinthe and Corona Jailbreaks for iOS 5.0.1: `http://conference.hitb.org/hitbsecconf2012ams/materials/`
6. Apple Developer: Memory Management — Discusses memory management from the user mode perspective
7. Apple Developer: Grand Central Dispatcher Reference
8. Apple Developer: Concurrency Programming Guide

5

Non Sequitur: Process Tracing and Debugging

Sooner or later, any developer — and often, the system administrator as well — are required to call on debugging skills. Whether it is their own code, an installed application, or sometimes the system itself, and whether they are just performing diagnostics or trying to reverse engineer, debugging techniques prove invaluable.

Debugging can quickly turn into a quagmire, and often requires that you unleash the might of GDB — the GNU Debugger, and go deep into the nether regions of architecture-specific assembly. OS X contains a slew of debugging tools and enhancements, which can come in very handy, and help analyze the problem before GDB is invoked. Apple dedicates two TechNotes for what they call "Debugging Magic"[1,2], but there are even more arcane techniques worth discussing. We examine these next.

DTRACE

First and foremost mention amongst all debugging tools in OS X must be given to DTrace. DTrace is a major debugging platform, which was ported from Sun's (Oracle's) Solaris. Outside Solaris, OS X's adoption of DTrace is the most complete. Detailing the nooks and crannies of DTrace could easily fill up an entire book, and in fact does[3], and therefore merits the following section.

The D Language

The "D" in Dtrace stands for the D language. This is a complete tracing language, which enables the creation of specialized tracers, or *probes*.

D is a rather constrained language, with a rigorous programming model, which follows that of AWK. It lacks even the basic flow control, and loops have been removed from the language altogether. This was done quite intentionally, because the D scripts are compiled and executed by kernel code, and loops run the risk of being too long, and possibly infinite. Despite these constraints, however, DTrace offers spectacular tracing capabilities, which rival — and in some cases greatly exceed — those of `ptrace(2)`. This is especially true in OS X, where the implementation of the latter is (probably intentionally) crippled, and hence deserves little mention in this book.

> *Both the DTrace and* `ptrace(2)` *facilities in OS X are not operating at their full capacity. Quite likely, this is due to Apple's concerns about misuse of the tremendous power these mechanisms provide, which could give amateurs and hackers the keys to reverse engineer functionality. This holds even stronger in iOS, wherein DTrace functionality is practically non-existent.*
>
> *The* `ptrace(2)` *functionality is especially impaired: Unlike its Linux counterpart, which allows the full tracing and debugging of a process (making it the foundation of Linux's* `strace`*,* `ltrace`*, and* `gdb`*), the OS X version is severely crippled, not supporting any of the* `PT_READ_*` *or* `PT_WRITE_*` *requests, leaving only the basic functions of attachment and stopping/continuing the process.*
>
> *Apple's protected processes, such as iTunes, make use of a* `P_LNOATTACH` *flag to completely deny tracing (although this could be easily circumvented by recompiling the kernel).*

DTrace forms the basis of XCode's Instruments tool, which is, at least in this author's opinion, the best debugging and profiling tool to come out of any operating system. Instruments allow the creation of "custom" instruments, which are really just wrappers over the raw D scripts, as shown in Figure 5-1.

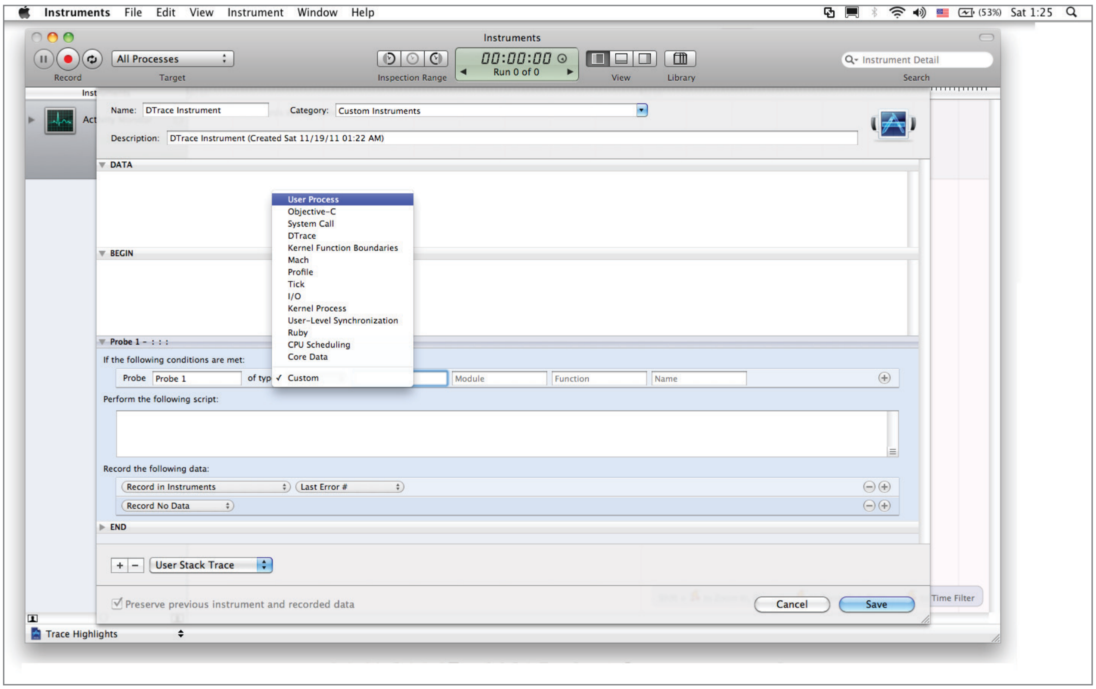


FIGURE 5-1: Instruments' custom instrument dialog box, a front-end to DTrace

Many of Solaris's D scripts have been copied verbatim (including the Solaris-oriented comments) to OS X. They are generally one of two types:

- **Raw D scripts:** These are clearly identifiable by their `.d` extension and are set to run under `/usr/sbin/dtrace -s`, using the `#!` magic that is common to scripts in UNIX. When the kernel is requested to load them, the `#!` redirects to the actual DTrace binary. These scripts accept no arguments, although they may be tweaked by direct editing and changing of some variables.
- **D script wrappers:** These are shell scripts (`#!/bin/sh`), that use the shell functionality to process user arguments and embed them in an internal D script (by simple variable interpolation). The actual functionality is still provided by DTrace (`/usr/sbin/dtrace -n`) but is normally invisible.

Because of the `.d` extension, it is easy to find all raw scripts in a system (try `find / -name "*.d" 2>/dev/null`). The wrapped scripts, however, offer no hint as to their true nature. Fortunately, both types of scripts have corresponding man pages, and a good way to find both types is to search by the `dtrace` keyword: they all have "Uses DTrace" in their description, as shown in Output 5-1:

OUTPUT 5-1: Displaying DTrace related programs on OS X using the man "–k" switch

```
morpheus@ergo (/) man -k dtrace
bitesize.d(1m)          - analyse disk I/O size by process. Uses DTrace
cpuwalk.d(1m)           - Measure which CPUs a process runs on. Uses DTrace
creatbyproc.d(1m)       - snoop creat()s by process name. Uses DTrace
dappprof(1m)            - profile user and lib function usage. Uses DTrace
dapptrace(1m)           - trace user and library function usage. Uses DTrace
diskhits(1m)            - disk access by file offset. Uses DTrace
dispqlen.d(1m)          - dispatcher queue length by CPU. Uses DTrace
dtrace(1)               - generic front-end to the DTrace facility
dtruss(1m)              - process syscall details. Uses DTrace
errinfo(1m)             - print errno for syscall fails. Uses DTrace
execsnoop(1m)           - snoop new process execution. Uses DTrace
fddist(1m)              - file descriptor usage distributions. Uses DTrace
filebyproc.d(1m)         - snoop opens by process name. Uses DTrace
hotspot.d(1m)           - print disk event by location. Uses DTrace
httpdstat.d(1m)         - realtime httpd statistics. Uses DTrace
iofile.d(1m)             - I/O wait time by file and process. Uses DTrace
iofileb.d(1m)            - I/O bytes by file and process. Uses DTrace
iopattern(1m)           - print disk I/O pattern. Uses DTrace
iopending(1m)           - plot number of pending disk events. Uses DTrace
iosnoop(1m)             - snoop I/O events as they occur. Uses DTrace
iotop(1m)               - display top disk I/O events by process. Uses DTrace
kill.d(1m)              - snoop process signals as they occur. Uses DTrace
lastwords(1m)           - print syscalls before exit. Uses DTrace
loads.d(1m)             - print load averages. Uses DTrace
newproc.d(1m)           - snoop new processes. Uses DTrace
opensnoop(1m)           - snoop file opens as they occur. Uses DTrace
pathopens.d(1m)         - full pathnames opened ok count. Uses DTrace
pidpersec.d(1m)         - print new PIDs per sec. Uses DTrace
plockstat(1)            - front-end to DTrace to print statistics about POSIX mutexes
                          and read/write locks
```

continues

OUTPUT 5-1 *(continued)*

```
priclass.d(1m)          - priority distribution by scheduling class. Uses DTrace
pridist.d(1m)           - process priority distribution. Uses DTrace
procsystime(1m)         - analyse system call times. Uses DTrace
runocc.d(1m)            - run queue occupancy by CPU. Uses DTrace
rwbypid.d(1m)           - read/write calls by PID. Uses DTrace
rwbytype.d(1m)          - read/write bytes by vnode type. Uses DTrace
rwsnoop(1m)             - snoop read/write events. Uses DTrace
sampleproc(1m)          - sample processes on the CPUs. Uses DTrace
seeksize.d(1m)          - print disk event seek report. Uses DTrace
setuids.d(1m)           - snoop setuid calls as they occur. Uses DTrace
sigdist.d(1m)           - signal distribution by process. Uses DTrace
syscallbypid.d(1m)      - syscalls by process ID. Uses DTrace
syscallbyproc.d(1m)     - syscalls by process name. Uses DTrace
syscallbysysc.d(1m)     - syscalls by syscall. Uses DTrace
topsyscall(1m)          - top syscalls by syscall name. Uses DTrace
topsysproc(1m)          - top syscalls by process name. Uses DTrace
weblatency.d(1m)        - website latency statistics. Uses DTrace
```

The (hopefully intrigued) reader is encouraged to check out these scripts on his or her own. Although not all work perfectly, those that are functional offer a staggering plethora of information. The potential uses (for tracing/debugging) and misuses (reversing/cracking) are equally vast.

dtruss

Of the many DTrace-enabled tools in OS X, one deserves an honorable mention. The `dtruss(1)` tool is a DTrace-powered equivalent of Solaris's longtime `truss` tool (which is evident by its man page, which still contains references to it). The `truss` tool may be more familiar to Linux users by its counterpart, strace. Both enable the tracing of system calls by printing the calls in C-like form, showing the system call, arguments, and return value. This is invaluable as a means of looking "under the hood" of user mode, right down to the kernel boundary.

Unlike Linux's strace, `dtruss` isn't smart enough to go the extra step and dereference pointers to structures, providing detailed information on fields. It is, however, powerful enough to display character data, which makes it useful for most system calls that accept file names or string data. There are three modes of usage:

- **Run a process under dtruss:** By specifying the command and any arguments after those of `dtruss`
- **Attach to a specific instance of a running process:** By specifying its PID as an argument to `dtruss -p`
- **Attach to named processes:** By specifying the name as an argument to `dtruss -n`

Another useful feature of `dtruss` is its ability to automatically latch onto subprocesses (specify `-f`). This is a good idea when the process traced spawns others.

It is possible to use `dtruss` as both a tracer and a profiler. The default use will trace all system calls, presenting a very verbose output. Output 5-2 shows a sample, truncated for brevity.

OUTPUT 5-2: A sample output of dtruss

```
SYSCALL(args)           = return
getpid(0x7FFF5FBFF970, 0x7FFFFFE00050, 0x0)            = 5138 0

... // Loading the required libraries

bsdthread_register(0x7FFF878A2E7C, 0x7FFF87883A98, 0x2000)           = 0 0
thread_selfid(0x7FFF878A2E7C, 0x7FFF87883A98, 0x0)            = 69841 0
open_nocancel("/dev/urandom\0", 0x0, 0x7FFF70ED5C00)          = 3 0

    // read random data from /dev/urandom

    // various sysctls…

getrlimit(0x1008, 0x7FFF5FBFF520, 0x7FFF8786D2EC)            = 0 0
open_nocancel("/usr/share/locale/en_US.UTF-8/LC_CTYPE\0", 0x0, 0x1B6)             = 3 0
  // read various locale (language) settings
read_nocancel(0x3, "RuneMagAUTF-8\0", 0x1000)                = 4096 0
read_nocancel(0x3, "\0", 0x1000)             = 4096 0
  // …
read_nocancel(0x3, "@\004\211\0", 0xDB70)             = 56176 0
close_nocancel(0x3)            = 0 0

  // open the file in question

open("/etc/passwd\0", 0x0, 0x0)              = 3 0
fstat64(0x1, 0x7FFF5FBFF9D0, 0x0)            = 0 0
mmap(0x0, 0x20000, 0x3, 0x1002, 0x3000000, 0x0)              = 0x6E000 0
mmap(0x0, 0x1000, 0x3, 0x1002, 0x3000000, 0x0)               = 0x8E000 0

  // read the data

read(0x3, „##\n# User Database\n# \n# Note that this file is consulted directly only
when the system is running\n# in single-user mode.  At other times this information
is provided by\n# Open Directory.\n#\n# This file will not be consulted for
authentication unless the BSD", 0x20000)
         = 3662 0
..
```

The various system calls can be quickly looked up in the man (section 2). Even more valuable output can be obtained from adding `-s`, which offers a stack trace of the calls leading up to the system call. This makes it useful to isolate which part of the executable, or a library thereof, was where the call originated. If you have the debugging symbols (that is, compiled with `–g`, and have the companion `.dSym` file), this can quickly pinpoint the line of code, as well.

For profiling, the `–c`, `-d`, `-e`, and `–o` switches come in handy. The first prints the summary of system calls, and the others print various times spent in the system call. Note that sifting through so much information is no mere feat by itself. The primary advantages of using DTrace scripts and `dtruss` are remote execution and textual format, which is relatively easily grep(1)-pable. If a Graphical User Interface (GUI) is preferable, the Instruments application provides a superb GUI, which enables a timeline-based navigation and arbitrary levels of zooming in and out on the data.

How DTrace Works

DTrace achieves its debugging magic by enabling its probes to execute in the kernel. The user mode portion of DTrace is carried out by /usr/lib/dtrace.dylib, which is common to both Instruments and /usr/sbin/dtrace, the script interpreter. This is the runtime system that compiles the D script. For most of the useful scripts, however, the actual execution, is in kernel mode. The DTrace library uses a special character device (/dev/device) to communicate with the kernel component.

Snow Leopard has some 40 DTrace providers and Lion has about 55, although only a small part of them are in the kernel. Using dtrace -l will yield a list of all providers, but those include PID instances, with multiple instances for function names. To get a list of the actual provider names, it makes sense to strip the PID numbers and then filter out only unique matches. A good way to do so is shown in Output 5-3.

OUTPUT 5-3: Displaying unique DTrace providers

```
root@ergo(/)# dtrace -l |       # List all providers
              tr -d '[0-9]'  |  # Remove numbers (pids , etc)
              tr -s ' ' |       # Squeeze spaces (so output can be cut)
              cut -d' ' -f2 |   # isolate second field (provider)
              sort -u           # Sort, and only show unique providersCalAlarmAgentProbe
Cocoa_Autorelease
CoreData
CoreImage
ID
JavaScriptCore
MobileDevice
PrintCore
QLThumbnail
QuickTimeX
RawCamera
..
```

The key registered DTrace providers in the kernel are shown in Table 5-1:

TABLE 5-1: Registered DTrace providers in OS X (partial list)

PROVIDER	PROVIDERS
dtrace	DTrace itself (used for BEGIN, END, and ERROR).
fbt	Function boundary tracing: low-level tracing of function entry/exit.
mach_trap	Mach traps (entry and return).
proc	Process provider: Enables monitoring a process by PID.
profile	Profiling information. Used to provide a tick in scripts that require periodic sampling.
sched	The Mach scheduler.
syscall	BSD system calls (entry and return).
vminfo	Virtual memory information.

Exercise: Demonstrating deep kernel system call tracing

As another great example of just how powerful DTrace is, consider the script in Listing 5-1:

LISTING 5-1: A D script to trace system calls — all the way into kernel space

```
#pragma D option flowindent /* Auto-indent probe calls */

syscall::open:entry
{
        self->tracing = 1;    /* From now on, everything is traced */
        printf("file at: %x  opened with mode %x", arg0, arg1);
}

fbt:::entry
/self->tracing/
{
   printf("%x %x %x", arg0, arg1,arg2);  /* Dump arguments */
}

fbt::open:entry
/self->tracing/
{
  printf ("PID %d (%s) is opening \n"  ,
     ((proc_t)arg0)->p_pid , ((proc_t)arg0)->p_comm);
}


fbt:::return
/self->tracing/
{
   printf ("Returned %x\n", arg1);
}
syscall::open:return
/self->tracing/
{
        self->tracing = 0;   /* Undo tracing */
        exit(0);             /* finish script */
}
```

The script begins with a `syscall` probe, in this case probing `open(2)` — you can modify the script easily by simply replacing the system call name. On entry, the script sets a Boolean flag — `tracing`. The use of the "self" object makes this flag visible in all other probes, effectively serving as a global variable.

From the moment `open(2)` is called, the script activates two `fbt` probes. The first simply dumps up to three arguments of the function. The second is a specialized probe, exploiting the fact we know exactly which arguments `open(2)` expects in kernel mode — in this case, the first argument is a `proc_t` structure. By casting the first argument, we can access its subfields — as is shown by printing out the value of `p_pid` and `p_comm`. This is possible because the argument is in the providing module's address space (in this case, the kernel address space, since the providing module is `mach_kernel`).

Finally, on return from any function, its return value — accessible in `arg1` — is printed. When the open function finally returns, the tracing flag is disabled, and the script exits.

Running this script will produce an output similar to Output 5-4:

OUTPUT 5-4: Running the example from Listing 5-1

```
CPU FUNCTION
  3  => open                                file at: 10f80bdf0  openeed with mode 4
  3    -> open                              PId 69 (mds) is opening

  3     | open:entry                        ffffff801561aa80 ffffff80158ac6d4
                                            ffffff801837a608
  3      -> __pthread_testcancel            1 ffffff80158ac6d4 ffffff801837a608
  3      <- __pthread_testcancel            Returned ffffff801837a5c0

  3      -> vfs_context_current             ffffff8015fe0ec0 ffffff80158ac6d4 0
  3      <- vfs_context_current             Returned ffffff801837a718

  3      -> vfs_context_proc                ffffff801837a718 ffffff80158ac6d4 0
  3        -> get_bsdthreadtask_info        ffffff8015fe0ec0 ffffff80158ac6d4 0
  3        <- get_bsdthreadtask_info        Returned ffffff801561aa80
  3      <- vfs_context_proc                Returned ffffff801561aa80
...
   (output truncated for brevity)
...
  3      -> proc_list_unlock                ffffff8013ed5970 10 ffffff8013ed5970
  3      <- proc_list_unlock                Returned ffffff80008d91b0

  3      -> lck_mtx_unlock                  ffffff8013ed5970 10 ffffff8013ed5970
  3      <- lck_mtx_unlock                  Returned 1f0000

  3    <- open                              Returned 0
```

As an exercise, try adapting the D-Script from Listing 5-1 to intercept Mach traps, rather than BSD system calls.

OTHER PROFILING MECHANISMS

DTrace is fast becoming the tracing mechanism of choice in OS X, but it is not the only one. Other alternatives exist, which is especially important in iOS, wherein DTrace does not exist.

The Decline and Fall of CHUD

OS X and iOS had a framework called CHUD (Computer Hardware Understanding and Development). This framework, made private in Snow Leopard and apparently removed as of Lion, was an exceptionally powerful framework, which could be used to register callbacks at various points in the kernel. The CHUD APIs were used by many of the XCode profiling tools back when OS X was primarily PPC-based, chiefly the now obsolete applications such as Reggie_SE and Shark (made extinct by Instruments). The APIs were utilized by specialized kernel extensions, which still exist in Snow Leopard (`CHUDKernLib`, `CHUDProf`, and `CHUDUtils`). These no longer appear in public as of Lion. CHUD still has a dedicated system call (#185), but it returns EINVAL unless a callback has been registered (usually by the `CHUDProf` kext), and CHUD has been enabled.

Before the move to Intel, XNU had architecture-specific calls for PPC to enable CHUD. It seems that, with the fall from grace of PPC, so too has CHUD lost its charm. The APIs are now reserved for Apple's internal use, mostly in iOS. The `CHUD.Framework`, required to access CHUD functionality from user space, is private in Snow Leopard, and has disappeared completely from OS X in Lion. The framework still exist in in the iOS SDK DiskDeveloperImage (`/Developer/Library/PrivateFrameworks`), and some tools, notably `chudRemoteCtrl`, rely on it. Additionally, both the iOS and OS X kernels contain the CHUD symbols, but the APIs are not made public in any way. It is likely that Apple still uses CHUD privately, especially in iOS.

AppleProfileFamily: The Heir Apparent

CHUD may have gone missing, but its essence remains. Profiling in both OS X and iOS is taken over by the private `AppleProfileFamily.framework` (and the `CoreProfile.framework`, which builds on it). This framework is quite similar to CHUD, in that it makes use of the latter's abandoned kernel callbacks, and communicates with various dedicated profiling kexts. The kexts, shown in Table 5-2, resided with their ilk in `/System/Library/Extensions` in Snow Leopard, but have since been moved (in Lion) into the `AppleProfileFamily.Framework/resources` in OS X. Putting kexts into a framework is a rather curious decision, but likely help keeps them private. In iOS these kexts are pre-linked into the kernel.

TABLE 5-2: AppleProfileFamily kexts common to OS X and iOS

KEXT	DESCRIPTION
AppleProfileFamily	Provides foundation and base class for other extensions. This kext also apparently claims the CHUD callbacks in XNU.
AppleProfileCallstackAction	Traces function call stacks. Registers the `appleprofile.actions.callstack` sysctls.
AppleProfileKEventAction	Traces kevents. Registers `appleprofile.actions.kevent` sysctls.
AppleProfileReadCounterAction	Reads performance Monitor counters. Registers `appleprofile.pmcs` sysctls.
AppleProfileRegisterStateAction	Saves register state during profiling. Registers `appleprofile.actions.register_state` sysctls.
AppleProfileTimestampAction	Handles accurate timestamps during events. Registers `appleprofile.actions.timestamp` sysctls.
AppleProfileThreadInfoAction	Profiles threads. Registers `appleprofile.actions.threadinfo` sysctls.

OS X has an additional kext for Intel (or IntelPenryn) profiling. As shown above, the kexts register several `sysctl` MIBs under the `appleprofile` parent (triggers, actions, and `pmcs`), mostly to control buffer and memory sizes. None are, at present, documented, though `sysctl appleprofile` can display them, and using `strings(1)` on the `AppleProfileFamily` kext provides a rough description for them. Another component, `/usr/libexec/appleprofilepolicyd`, remains in user mode and serves as the arbiter and policy decision maker.

PROCESS INFORMATION

In addition to DTrace, which is powerful enough, OS X provides two key mechanisms to obtain detailed process information, such as open handles, memory utilization, and other statistics, the likes of which are used by `ps(1)`, `lsof(1)`, `netstat(1)`, and friends.

sysctl

The sysctl mechanism, which has already been discussed in the previous chapters, offers variables to display statistics pertaining to processes. This mechanism is crucial in order to obtain the list of the process IDs (and is, in fact, the means by which this list is obtained in `ps(1)` and `top(1)`).

The kern namespace exposes the `KERN_PROCARGS` and `KERN_PROCARGS2` MIBs under `CTL_KERN`. These may be used with the third level MIB value of any PID on the system, in order to retrieve the argument and environment of that process.

proc_info

OS X and iOS both offer the `proc_info` system call. This undocumented system call (#336) is fundamental for many system utilities, such as `lsof(1)` and `fuser(1)`. Though it merits its own include file (`<sys/proc_info.h>`), the system call remains well hidden, and should be accessed via `<libproc.h>`, the header file for `libproc.dylib`, which is part of Darwin's `LibC` (and therefore part of `libSystem`)

Using `proc_info`, it is possible to query many aspects of processes and their threads. Chief among those is their use of file descriptors and sockets (hence the importance for `lsof(1)`-like tools). This is cardinal in systems wherein `/dev/kmem` is not available (which, by default, is all systems), as `sysctl(8)` can show addresses in kernel space, but cannot read them.

The `proc_info` system call accepts a `callnum` argument, and a `flavor`. Each `callnum` results in different functionality, according to one of the unnamed integer values in Table 5-3. These values are wrapped in `<libproc.h>` by functions:

TABLE 5-3: callnum values accepted by proc_info

CALLNUM	USED FOR
1	List all PIDs. Wrapped by `proc_listpids()` and others. In this case, the PID argument is taken to be one of the following: `#define PROC_ALL_PIDS 1` `#define PROC_PGRP_ONLY 2` `#define PROC_TTY_ONLY 3` `#define PROC_UID_ONLY 4` `#define PROC_RUID_ONLY 5`

CALLNUM	USED FOR
2	Return PID information for a specific PID. Wrapped by `proc_pidinfo()`. In this case, the flavor argument is taken to be one of the following: `PROC_PIDLISTFDS`: for file descriptors `PROC_PIDTBSDINFO`: for BSD task information info `PROC_PIDTASKINFO`: for Mach task information info `PROC_PIDTASKALLINFO`: Both Mach and BSD information `PROC_PIDTHREADINFO`: list of task's threads `PROC_PIDWORKQUEUEINFO`: kernel work queues held by task `PROC_PIDREGIONINFO`: list of memory regions (q.v. `vmmap(1)`) Lion further adds: `PROC_BSDSHORTINFO`: summary information of BSD attributes `PROC_PIDVNODEPATHINFO`: list of vnodes held by this PID `PROC_PIDLISTFILEPORTS`: List of fileports
3	Return file descriptor information for a specific PID. Wrapped by `proc_pidfdinfo()`. In this case, flavor is: `PROC_PIDFDVNODEINFO`: VNodes `PROC_PIDFDVNODEPATHINFO`: VNodes, with path `PROC_PIDFDSOCKETINFO`: Socket information `PROC_PIDFDPSHMINFO`: Shared memory descriptors `PROC_PIDFDPIPEINFO`: Pipes `PROC_PIDFDKQUEUEINFO`: Kernel queues `PROC_PIDFDATALKINFO`: AppleTalk descriptors
4	Return the kernel message buffer. Wrapped by `proc_kmsgbuf()`
5	Set process control parameters. Wrapped by `proc_setpcontrol();`
6	New in Lion and iOS 4.3: Return information about fileports for a specific PID. Wrapped by `proc_pidfileportinfo()`.

All of these values, save for the fifth, are informational only. The fifth callnum, however, can be used to set process control parameters.

LibProc wraps `proc_info` with several useful functions, as shown in Table 5-4:

TABLE 5-4: Functions in <libproc.h>

FUNCTION PROTOTYPE	USAGE
`int proc_listpids` `(uint32_t` *`type`*`,` `uint32_t` *`typeinfo`*`,` `void *`*`buffer`*`,` `int` *`buffersize`*`);`	Returns in `buffer` a list of all PIDs in the system. Used as the basis for other functions.

continues

TABLE 5-4 *(continued)*

FUNCTION PROTOTYPE	USAGE
`int proc_listpidspath (uint32_t type, uint32_t typeinfo, const char *path, uint32_t pathflags, void *buffer, int buffersize);`	Returns in `buffer` all PIDs holding a reference to `path` according to `pathflags`. (essentially, `fuser(1)` in a library call version). Return value is amount of bytes used in buffer.
`int proc_pidfdinfo (int pid, int fd, int flavor, void *buffer, int buffersize);`	Return in `buffer` a `proc_xxx_info` structure corresponding to the file descriptor `fd` of process with PID `pid`. The exact type of information is determined by *flavor*, which is as in `callnum` 3 (which this function wraps). Return value is amount of bytes used in buffer.
`proc_name(int pid, void *buffer, uint32_t buffersize);` `proc_path(int pid, void *buffer, uint32_t buffersize);`	Return in `buffer` the name (`proc_name`) or the full path (`proc_path`) of the process matching `pid`. Return value is amount of bytes used in `buffer`.
`int proc_regionfilename (int pid, uint64_t address, void *buffer, uint32_t buffersize);`	Return in `buffer` the name of the file mapping (if any) to which the `address` in the process matching `pid` belongs. Return value is amount of bytes used in `buffer`.
`int proc_kmsgbuf (void *buffer, uint32_t buffersize);`	Return up to buffersize bytes from the kernel ring buffer in buffer. This is the same output as one gets from the `dmesg(8)` command (which, in fact, is built around this function). Wraps callnum 4. Return value is amount of bytes actually returned.

Lion and iOS add several more informational wrappers, such as `proc_listallpids`, `proc_listpgrppids` (list processes according to process group), and `proc_listchildpids` (for process children) — but these are all nothing more than simple filters around the basic `listpids` call.

The book's companion website contains a tool, psleuth, demonstrating the many uses of `proc_info` for diagnostics.

PROCESS AND SYSTEM SNAPSHOTS

In addition to DTrace and Instruments, there are several tools in OS X which enable taking "snapshots" of the system or process state.

system_profiler(8)

The `system_profiler(8)` utility is the command line version of the graphical `System Profiler.app`, which most users know as About This Mac ➤ More Info. Whereas the graphical version is useful (and provides the memorable Speak Serial Number option), it is not as handy as its command-line counterpart, which can be run from a terminal and generate what is, essentially, the same output, albeit with greater filtering options. The report can be saved to either plain text or XML.

sysdiagnose(1)

New in Lion, `sysdiagnose(1)` is a one-stop comprehensive diagnostics utility. It generates a barrage of logiles, which are compressed and archived into a gzipped tar. The tool is meant to provide Apple with a complete diagnostics of the system, and produce a report which can be sent to Apple.

In reality, `sysdiagnose(1)` is really nothing more than a wrapper, which runs several other utilities (of which the important ones are described in this book) one after the other, and collects ASL logs and other files, as shown in Output 5-5:

OUTPUT 5-5: Running sysdiagnose(1):

```
root@simulacrum (/)# sysdiagnose
This diagnostic tool generates files that allow Apple to investigate issues with your
computer and help Apple to improve its products. The generated files may contain some
of your personal information, which may include, but not be limited to, the serial
number or similar unique number for your device, your user name, or your computer name.
The information is used by Apple in accordance with its privacy policy (www.apple.com
/privacy)and is not shared with any third party. By enabling this diagnostic tool and
sending a copy of the generated files to Apple, you are consenting to Apple?s use of
the content of such files.

Please press 'Enter' to continue    # If you want the output, you don't have a choice,
                                    # do you?

Helpful Hint: If a single process appears to be slowing down the system, pass in the
```

continues

OUTPUT 5-5 *(continued)*

```
process ID or name as the argument: sysdiagnose [pid | process_name]
Gathering time sensitive information
====================================
Running fs_usage, spindump and top

Done gathering time sensitive information. Proceeding to gather non time sensitive data
=========================================================================================
Running zprint
Running kextstat
Collecting BootCache Statistics
Running netstat
Running lsof
Running pmset diagnostics
Running allmemory. This will take a couple of minutes
Running system profiler
Copying kernel and system logs
Copying spin and crash reports
Running df
Running ioreg
sysdiagnose results written to /var/tmp/sysdiagnose_Apr.26.2012_03-40-56.tar.gz
```

A handy feature of this tool is that it can be run from Finder, by a key-chord (Control-Option-Command-Shift-Period, for which you'll likely need both hands!). Running from the command line offers the advantages of specifying a PID or process name (to run `vmmap(1)` and other memory tracing tools, discussed later in this chapter under "Memory Leaks"). Additionally, thorough mode may be specified (using the `-t` switch) in which it provides a full kernel trace and unflattered `allmemory(1)` data.

allmemory(1)

The `allmemory(1)` tool is used to capture a snapshot of all memory utilization by user mode processes. When run, the tool iterates over each and every process in the system, and dumps their memory maps into files in `/tmp/allmemoryfiles` (or elsewhere, as may be specified by the `-o` switch). The dumps are in a simple plist format, making them suitable for parsing by third party tools, or by `allmemory(1)` itself, when run in "diff" mode, to compare snapshots. Unlike the process-specific `vmmap(1)`, `allmemory(1)` can display a system wide view of memory utilization, by comparing the utilization of similar memory segments by different processes, and focuses on shared memory.

After all process memory snapshots have been acquired, `allmemory(1)` goes on to display the aggregate statistics for each process, as well as for framework memory utilization, as shown in Output 5-6:

stackshot(1)

A little-known, but very useful feature in OS X and iOS is the ability to take a snapshot of the process execution state. Both systems offer a private and undocumented system call, `stack_snapshot` (#365), which can be used to capture the state of all the threads of a given process.

The main user of this system call is the `stackshot(1)` command, technically an on-demand daemon, which is hidden away in `/usr/libexec`. The command is meant to be run by `launchd(1)` (from `com.apple.stackshot.plist`), but is even more useful when run manually. It is possible to either single out a specific PID (with `-p`), or take on all the processes in the system. The default log file

OUTPUT 5-6: Sample output of the allmemory(1) tool

```
root@Ergo (/)# allmemory
.........................................................................................

                              Process Name [PID]  Architecture PrivateRes/NoSpec Copied    Dirty    Swapped    Shared/NoSpec
                              ================== ============ ================= ========= ======== ========   =============

                              vmware-vmx [28749]: 64-bit       251691  /  251691     244     74091        8  31098 / 31087
                                  firefox [ 133]: 64-bit       132360  /  132360     36604  106111     43493  40220 / 40066
                                      ...
                                      ...
                       ALL PROCESSES PRIVATE TOTAL:            593815  /  591785     54371  300656    131889      0 /     0
                          DYLD SHARED CACHE SHARED:             29226  /   29213         0       0        35  29226 / 29213


                                                             TotalRes / NoSpec   Copied    Dirty    Swapped    Shared/NoSpec
                                                             =================  ======== ========= =========  =============

                    ALL PROCESSES TOTAL:                     665779  /  663274     54371    327084    141720  71964 / 71489
                    --------------------
                           Mapped file:                      250203  /  250119      1220     60957      1318   4456 /  4372
                                No tag:                      125254  /  125254     35704    102538     45486   1765 /  1720
                          MALLOC_SMALL:                       63146  /   63070      6873     37518     26515   1232 /  1232
                                 IOKit:                       39391  /   39391         0     37847      4686  13937 / 13937
                           MALLOC_TINY:                       31015  /   31011       177     24152      8303    315 /   315
                          MALLOC_LARGE:                       29780  /   29780        42     15763     18065      0 /     0
                     DYLD shared cache:                       29250  /   29248      4401      2842      2908  24849 / 24847
                              ...

                     Framework/Image Name        Architecture    Resident/NoSpec  Copied    Dirty      Swapped  Filesize
                                                                                                                (pages)
                     =====================       ============    =============== ========= =========  ======== =========
                                      XUL:          64-bit          4572 / 4572       471        55       136     14792
                                  CoreFP1:          32-bit          2725 / 2725         7        10      1025      8947
                                   AppKit:          64-bit          2580 / 2580       563       347       173     11101
                                  WebCore:          64-bit          1741 / 1741        57        20         5     16937
                                ...
```

is saved to /Library/Logs/stackshot.log, unless overridden with a -f switch. It is also possible to send the log to a remote server by specifying a Trace Server key in the daemon's plist. Any number of snapshots can be taken (with the -n switch), though the common use is to use the -i switch to take an immediate snapshot and exit. Incidentally, the man page erroneously states "-u" as a switch to enable symbolification of the output, even though that switch is not supported from the command line.

The stackshot(1) command has been enhanced in Lion by integrating it with the sysdiagnose(1) command. This command, discussed above, collects the stack snapshots of all processes along with the myriad other data and logs. Stackshot also has its own keychord, to run independently of sysdiagnose(1). iOS used to include stackshot(1), but it has mysteriously disappeared in iOS 5. The system call, however, is still available, and can be used as is shown next.

The stack_snapshot System Call

XNU's stack_snapshot system call only gets an obligatory mention in <sys/syscall.h>, by virtue of its being system call number 365. Otherwise, it remains an undocumented system call. Even the stackshot(1) command invokes it via the syscall wrapper (which you can easily verify using dtruss(1) and/or disassembly). The following exercise demonstrates using the system call, by mimicking the functionality of stackshot(1).

Exercise: Using stack_snapshot

Even though stack_snapshot is undocumented in user mode, not all is lost. XNU remains open source, and looking at XNU's sources, (in particular, bsd/kern/kdebug.c) reveals the system call expects a pid (or -1, for all), a buffer to put the snapshot in, a buffer size, and some options. The actual implementation of the snapshot mechanism is tucked deep within the Mach microkernel. Specifically, osmfk/kern/debug.h reveals the structures and constants used by the logic. The APIs are declared private and unstable, but have been around for quite a while, and are also present in iOS. Because they are part of the kernel sources and not the standard #includes, the following example copies them.

Listing 5-2 should compile cleanly on either OS X or iOS, and bring back to iOS the missing stackshot(1) functionality.

LISTING 5-2: Do-it-yourself stackshot for OS X and iOS

```
#include <stdlib.h> // for malloc
#include <stdio.h>
#include <string.h>

struct frame {
        void *retaddr;
        void *fp;
};


// The following are from osfmk/kern/debug.h
#define STACKSHOT_TASK_SNAPSHOT_MAGIC 0xdecafbad
#define STACKSHOT_THREAD_SNAPSHOT_MAGIC 0xfeedface
#define STACKSHOT_MEM_SNAPSHOT_MAGIC  0xabcddcba

struct thread_snapshot {
        uint32_t                snapshot_magic;
```

```
        uint32_t                nkern_frames;
        uint32_t                nuser_frames;
        uint64_t                wait_event;
        uint64_t                continuation;
        uint64_t                thread_id;
        uint64_t                user_time;
        uint64_t                system_time;
        int32_t                 state;
        char                    ss_flags;
} __attribute__ ((packed));


struct task_snapshot {
        uint32_t                snapshot_magic;
        int32_t                 pid;
        uint32_t                nloadinfos;
        uint64_t                user_time_in_terminated_threads;
        uint64_t                system_time_in_terminated_threads;
        int                     suspend_count;
        int                     task_size;    // pages
        int                     faults;         // number of page faults
        int                     pageins;        // number of actual pageins
        int                     cow_faults;     // number of copy-on-write faults
        char                    ss_flags;
        char                    p_comm[17];
} __attribute__ ((packed));



int stack_snapshot(int pid, char *tracebuf, int bufsize, int options)
{
        return syscall (365, pid, tracebuf, bufsize, options);
}


int dump_thread_snapshot(struct thread_snapshot *ths)
{

   if (ths->snapshot_magic != STACKSHOT_THREAD_SNAPSHOT_MAGIC)
        {
             fprintf(stderr,"Error: Magic %p expected, Found %p\n",
                     STACKSHOT_TASK_SNAPSHOT_MAGIC, ths->snapshot_magic);
              return;
        }

    printf ("\tThread ID: 0x%x ", ths->thread_id) ;
    printf ("State: %x\n" , ths->state);
    if (ths->wait_event) printf ("\tWaiting on: 0x%x ", ths->wait_event) ;
    if (ths->continuation) {
    printf ("\tContinuation: %p\n", ths->continuation);

        }
    if (ths->nkern_frames || ths->nuser_frames)
    printf ("\tFrames:    %d kernel %d user\n", ths->nkern_frames, ths->nuser_frames);
```

continues

LISTING 5-2 *(continued)*

```
        return (ths->nkern_frames + ths->nuser_frames);
}

void dump_task_snapshot(struct task_snapshot *ts)
{
   if (ts->snapshot_magic != STACKSHOT_TASK_SNAPSHOT_MAGIC)        {
            fprintf(stderr,"Error: Magic %p expected, Found %p\n",
            STACKSHOT_TASK_SNAPSHOT_MAGIC, ts->snapshot_magic);
             return;
        }
   fprintf(stdout, "PID: %d (%s)\n", ts->pid, ts->p_comm);

}

#define BUFSIZE 50000 // Sufficiently large..

int main (int argc, char **argv)
{
    char buf[BUFSIZE];
    int rc = stack_snapshot(-1, buf, BUFSIZE,100);
    struct task_snapshot *ts;
    struct thread_snapshot *ths;
    int off = 0;
    int warn = 0;
    int nframes = 0;

    if (rc <0) { perror ("stack_snapshot"); return (-1); }

    while (off< rc) {
     // iterate over buffer, which is a contiguous dump of snapshot structures

     ts = (struct task_snapshot *) (buf + off);
     ths = (struct thread_snapshot *) (buf + off);

     switch (ts->snapshot_magic)
        {
           case STACKSHOT_TASK_SNAPSHOT_MAGIC:
                dump_task_snapshot(ts);
                off+= (sizeof(struct task_snapshot));
                warn = 0;
                break;
           case STACKSHOT_THREAD_SNAPSHOT_MAGIC:
                nframes = dump_thread_snapshot(ths);
                off+= (sizeof(struct thread_snapshot));
                off+=8;
                if (nframes)
                  { printf("\t\tReturn Addr\tFrame Ptr\n");}
                while (nframes)
                  {
                    struct frame *f = (struct frame *) (buf + off);
                    printf ("\t\t%p\t%p\n", f->retaddr, f->fp);
                    off += sizeof(struct frame);
                    nframes--;
                  }
```

```
                warn = 0;
                break;
          case STACKSHOT_MEM_SNAPSHOT_MAGIC:
                printf ("MEM magic - left as an exercise to the reader\n");
                break;
           default:
                if (!warn) {
                warn++;
                fprintf(stdout, "Magic %p at offset %d?"
                                "Seeking to next magic\n",
                                ts->snapshot_magic, off);}
                 off++;;


        } // end switch

    } // end while
}
```

KDEBUG

XNU contains a built-in kernel trace facility called `kdebug`. This very powerful, yet poorly documented facility is present in both OS X and iOS, though it is often disabled by default, unless enabled by a `sysctl(8)` setting. At various points throughout, the kernel is laced with special `KERNEL_DEBUG_CONSTANT` macros. These macros enable the tracing of noteworthy events, such as system calls, Mach traps, file system operations and IOKit traces, albeit in compressed form, described later. This means that very little extra information besides the event occurrence itself can be recorded in this manner.

kdebug-based Utilities

OS X provides three utilities which utilize the `kdebug` facility. The tools — `fs_usage(1)`, `sc_usage(1)`, and `latency(1)`, all require root privileges to operate, but provide valuable debugging and tracing information. Since `kdebug` messages are in compressed, encoded form, these utilities (in particular `sc_usage(1)`) rely on the existence of a "code" file, `/usr/share/misc/trace.codes`. This file does not exist in iOS, but can be copied.

sc_usage

The `sc_usage(1)` tool is used to display system call information on a per-process basis. The command can attach to an existing process (specified as a PID or process name), or can execute a new one (when invoked with `-E`). The tool can run in "watch" style mode, continuously updating the screen, or (if invoked with `-l`) display output continuously.

fs_usage

Much like its sister utility, `fs_usage(1)` can be used to display system calls, but in this case ones relating to files, sockets, and directories. Unlike its sibling, it can display calls performed system-wide (if invoked with a PID or command argument).

latency

The `latency(1)` tool displays latency values of interrupts and scheduling. It shows context switches and interrupt handlers falling within thresholds, which can be set with the `-it` or `-st` switches, respectively.

kdebug codes

kdebug uses kernel buffers for logging, and buffer space is extremely limited. Every debug "message," therefore, uses a 32-bit integer code, into which a class, a subclass, and a code must be squeezed. The format is defined in <sys/kdebug.h> as shown in Listing 5-3:

LISTING 5-3: The kdebug message format

```
/* The debug code consists of the following
*
* ----------------------------------------------------------------------
*|              |               |                               |Func   |
*| Class (8)    | SubClass (8)  |          Code (14)            |Qual(2)|
* ----------------------------------------------------------------------
* The class specifies the higher level
*/
```

The kdebug message classes correspond to kernel subsystems, and have, in turn, subclasses which are specific. These are also defined in <sys/kdebug.h>, though the header file also has some subclasses which are unused in practice. Key classes and subclasses are shown in Table 5-5:

TABLE 5-5: kdebug classes and subclasses. Shaded classes are for user space:

KDEBUG CLASS (DBG_)	SUBCLASSES (.. DENOTES CLASS #DEFINE)	USED FOR
MACH (1)	..._EXCP_*	Kernel hardware exceptions and traps
	..._VM(0x30)	Virtual memory subsystem
	..._MACH_LEAKS(0x31)	Memory allocations
	..._SCHED (0x40)	Scheduler subsystem
NETWORK (2)	DBG_NETIP (1) DBG_NETARP (2) DBG_NETUDP (3) DBG_NETTCP (4) ...	Various networking protocols supported in XNU (IP, TCP, UDP, IPSEC, etc). Calls are wrapped with a NETDBG_CODE macro
FSYSTEM (3)	These messages are filtered by fs_usage(1) DBG_FSRW (1) DBG_DKRW (2) DBG_FSLOOOKUP (4) DBG_JOURNAL (5) DBG_IOCTL (6) ...	Various filesystem operations. Calls are wrapped with an FSDBG_CODE macro. FileSystem drivers can register additional subclasses (e.g. DBG_HFS, DBG_EXFAT, etc).

KDEBUG CLASS (DBG_)	SUBCLASSES (.. DENOTES CLASS #DEFINE)	USED FOR
`BSD (4)`		The BSD Subsystem. Calls wrapped with `BSDDBG_CODE`
	…`_PROC (1)`	BSD Processes. Tracks process exit and forced exit events
	…`_EXCP_SC (0x0C)`	BSD System calls. These are filtered by `sc_usage(1)`
	…`_AIO (0x0D)`	Asynchronous I/O
	…`_SC_EXTENDED_INFO (0x0E)` …`_SC_EXTENDED_INFO2 (0x0F)`	Extended information on system calls such as `mmap(2)`, `pread(2)`, and `pwrite(2)`, encoding sizes and pointers
`IOKIT (5)`		IOKit Drivers. Codes up to 32 are internal to IOKit. Other IOKit classes define 32 and up. IOKit is described in detail in chapter 19.
`DRIVERS (6)`		Used by drivers of various buses. Not used in the kernel proper.
`TRACE (7)`		Various debug trace messages. Subcodes are `_DATA(0)`, `_STRING(1)`, and `_INFO(2)`.
`DLIL (8)`		Used by the Data Link Interface Layer (Layer II support, in bsd/net/dlil.c). Calls wrapped with `DLILDBG_CODE`.
`SECURITY (9)`		Reserved for security modules and subsystems. Calls wrapped with `SECURITYDBG_CODE`, but not used in kernel proper
`CORESTORAGE (10)`		New in Lion, to support CoreStorage logical volume management. Undocumented, not used in kernel proper.
`CG (11)`		New in Mountain Lion. Undocumented. Possibly CoreGraphics
`MISC (20)`		Reserved for miscellaneous uses. Undocumented.
`DYLD(31)`		Reserved for `dyld(1)` use.
`QT(32)`		Reserved for QuickTime. Undocumented.
`DBG_APPS(33)`		Used by Applications.

continues

TABLE 5-5 *(continued)*

KDEBUG CLASS (DBG_)	SUBCLASSES (.. DENOTES CLASS #DEFINE)	USED FOR
`LAUNCHD(34)`		Used exclusively by `launchd(1)`.
`DBG_PERF(37)`		New in Mountain Lion. Undocumented, likely for performance
`DBG_MIG(255)`		Used by the the Mach Interface Generator to trace sending and receiving of messages. MIG is described in chapter 9.

When used for function tracing, the last two bits of the code are defined for a "qualifier," which can specify `DBG_FUNC_START` or `DBG_FUNC_END`.

Writing kdebug messages

The `kdebug` facility is extensively used in XNU, but applications can also use it to log their own messages, as in fact some of Apple's own applications do. The `kdebug_trace` system call (#180), however, is purposely undocumented: Even those open source applications which do use it, do so by invoking `syscall` directly. This can be seen in `launchd(1)`, for example, as in Listing 5-3:

LISTING 5-3: Using kdebug through syscall directly.

```
void
runtime_ktrace1(runtime_ktrace_code_t code)
{
        void *ra = __builtin_extract_return_addr(__builtin_return_address(1));

        /* This syscall returns EINVAL when the trace isn't enabled. */
        if (do_apple_internal_logging) {
                syscall(180, code, 0, 0, 0, (long)ra);
        }
}
```

The `kdebug_trace` system call can actually use up to six arguments (the maximum for a system call). The `KERNEL_DEBUG_CONSTANT` pre-initializes some of these arguments, namely the fifth, with the identity of the current thread. The system call implementation and the `KERNEL_DEBUG_CONSTANT` code paths both eventually end up at `kernel_debug_internal()`, which performs the actual debugging. In both cases, though, the path to actual kdebugging first checks if the global kernel variable `kdebug_enable` is set, which is optimized by a `gcc` "improbable," as this variable is zero, unless manually set). The `kernel_debug_internal()` function takes the six arguments and writes them into a `struct kd_buf`, along with a timestamp, where they await to be read. If CHUD is enabled, a callback can be registered, to be invoked on every `kdebug` event.

Reading kdebug messages

Applications can enable `kdebug` and read messages from user mode using `sysctl(2)` calls. Before `kdebug` can be used, `kdebug_enable` must be set to a non-zero value. This variable is not visible from user mode, but `sysctl(2)` can be used here, as well, as shown in Listing 5-4:

LISTING 5-4: Enabling or disabling kdebug_enable from user mode via sysctl

```
int set_kdebug_enable(int value)
{
    int rc;
    int mib[4];

     mib[0] = CTL_KERN;
     mib[1] = KERN_KDEBUG;
     mib[2] = KERN_KDENABLE;
     mib[3] = value;
     if ((rc = sysctl(mib, 4, NULL, &oldlen, NULL, 0) < 0) {perror("sysctl");}
      return (rc);
}
```

The `KERN_KDENABLE` operation(3) is only one of the control codes which may be passed in the `CTL_KERN.KERN_KDEBUG` sysctl. The currently defined operations are listed in Table 5-6:

TABLE 5-6: Defined operations for KERN_KD*

KERN_KD* OPERATION	USAGE
`EFLAGS(1)`	Enable user flags specified (bitwise `OR`).
`DFLAGS(2)`	Disable user flags specified (bitwise `AND-NOT`).
`ENABLE(3)`	Enable/disable `kdebug`, as per above example.
`SETBUF(4)` `GETBUF(5)`	Set or get the number of `kdebug` buffers. The number of buffers should be called prior to KD_ENABLE.
`SETUP(6)`	Used to reinitialize `kdebug`.
`REMOVE(7)`	Clear kdebug buffers.
`SETREG(8)` `GETREG(9)`	Set values used for checking and filtering kdebug messages. Can `KDBG_CLASSTYPE`, `KDBG_SUBCLSTYPE`, `KDBG_RANGETYPE`, or `KDBG_VALCHECK`. `KD_GETREG` is #ifdef'ed out.
`READTR(10)`	Read trace buffer from kernel.
`PIDTR(11)`	Set only a particular PID for kdebug traces.
`THRMAP(12)`	Read thread map. Thread maps contain thread information, and the executable command (`argv[0]`).

continues

TABLE 5-6 *(continued)*

KERN_KD* OPERATION	USAGE
PIDEX(14)	Exclude a given PID from kdebug traces, but enable system-wide tracing.
SETRTCDEC(15)	Set a decrement value.
KDGETENTROPY(16)	Request system entropy. This is used by security software to generate stronger pseudo-random numbers (independent of /dev/random and /dev/urandom).

APPLICATION CRASHES

An unfortunate fact of life is that, sooner or later, most applications crash. In UNIX, a crash is associated with a signal. The true reason for the crash lies in the kernel code, which generates the signal as a last resort, after determining the process simply cannot continue execution. (Kernel crash reports, or "panics," are somewhat similar in concept, but contain different contents. They are discussed in Chapter 9.)

Core Dumps

When a process crashes, a core dump may optionally be generated. This is dependent on the process's `RLIMIT_CORE` resource limit. Processes may restrict this value using `setrlimit(2)`, although it is more common for the user to do so by means of the `ulimit(1)` command. A value of 0 reported by `ulimit -c` means no core dump will be created. Otherwise, a core file of up to the specified size will be created, usually in the `/cores` directory. The core can then be debugged with `gdb`, as shown in Listing 5-5.

LISTING 5-5: Demonstrating program crashes, with and without core.

```
morpheus@Ergo (~)$ cat test.c
#include <stdio.h>
int main ()
{
        int j = 24;
        printf ("%d\n",j/0);
        return (0); // not that we ever get here..
}
morpheus@Ergo (~)$ cc test.c -o test
test.c: In function 'main':
test.c:5: warning: division by zero        # just in case it's not clearly obvious J

morpheus@Ergo (~)$ ulimit -c
0
morpheus@Ergo (~)$ ./test                  # first run: signal kill, no core
Floating point exception

morpheus@Ergo (~)$ ulimit -c 99999999999   # ulimit increased
morpheus@Ergo (~)$ ./test
Floating point exception (core dumped)     # second run: core generated
```

```
morpheus@Ergo (~)$ ls -l /cores/          # and can be found in /cores
total 591904
-r--------  1 morpheus  admin  303054848 Nov 19 00:30 core.6267

morpheus@Ergo (~)$ file /cores/core.6267   # The file is of type Mach-O core
/cores/core.6267: Mach-O 64-bit core x86_64
morpheus@Ergo (~)$ cd ~/Library/Logs/CrashReporter  # Go to where all logs are located
morpheus@Ergo (~)$ ls -l test*                      # and note both examples generated
                                                    # reports
-rw-------  1 morpheus  staff  1855 Nov 19 00:59 test_2011-11-19-005918_Ergo.crash
-rw-------  1 morpheus  staff  1855 Nov 19 01:09 test_2011-11-19-010917_Ergo.crash
```

Core file creation is usually disabled at the user level by default, that is, `ulimit -c` is set to 0. This is for good reason: As the example in Listing 4-2 shows, even a three-line program produces a core of close to 300 MB! It can be re-enabled on a global basis by setting `launchd`'s limits — as all processes in the system are its eventual descendants.

At the system level, core files may be controlled by `sysctl(8)`. The settings shown in Table 5-7 are applicable:

TABLE 5-7: sysctl settings relating to core files

SYSCTL SETTING	DEFAULT	USED FOR
`kern.corefile`	`/cores/core.%P`	Name of core generated. `%P` is a placeholder for the PID, which allows multiple core files to be collected in `/cores`.
`kern.coredump`	`1`	Enabling/disabling core dumps, system-wide. Note: `RLIMIT_CORE` limit must hold per process.
`kern.sugid_coredump`	`0`	Dump core for setuid and setgid programs. Set to `0` because these programs often contain sensitive information.

Crash Reporter

Rather than deal with huge core files, both iOS and OS X contain a CrashReporter, which is triggered automatically on a process abend (abnormal end, i.e. crash), and generate a detailed crash log. This mechanism performs a quick, rudimentary analysis on the process before its quietus, and records the highlights in a crash log. The crash reporter is key for application developers, especially on iOS, and Apple dedicates several TechNotes to its documentation.[4,5]

In both iOS and OS X, CrashReporter logs are sent to the user's `Library/Logs/CrashReporter`, or the system-wide `/Library/Logs/CrashReporter`. In recent version of OS X, these directories are a symbolic link to `../DiagnosticReports`. In iOS, the logs are made available to the host when the device is connected. The report name follows a convention of *process _name* `_YYYY-MM-DD-HHMMSS_`*hostname*`.crash`.

The crash report provides a basic, but oftentimes sufficient, analysis of what went wrong. Depending on architecture — i386, x86_64, or ARM — the format may be different, but it always follows the same basic structure, shown in Output 5-7. The output is from an iOS process crash, and the fields in *italics* are specific to iOS.

OUTPUT 5-7: A sample crash report.

```
Incident Identifier: C15D9ACD-DD6E-4124-857F-24FBBCC18C10
CrashReporter Key:   0941d515f2e15ef3202751ef6776efc732ce4713
Hardware Model:      iPod4,1
Process:         MobileNotes [9123]      // process name, with [PID]
Path:            /Applications/MobileNotes.app/MobileNotes
Identifier:      MobileNotes
Version:         ??? (???)
Code Type:       ARM (Native)            // or i386 or X86-64
Parent Process:  launchd [1]

Date/Time:       2011-11-19 10:16:00.896 +0800
OS Version:      iPhone OS 5.0 (9A334)   //  Mac OS X 10.6.8 (10K549) , etc..
Report Version:  104

Exception Type:  EXC_CRASH (SIGFPE)      // Mach exception code (UNIX signal)
Exception Codes: 0x00000000, 0x00000000  // Exception code, if any
Crashed Thread:  0                       // Thread number of faulting thread

// Thread call stacks follow. Faulting thread (in this case, 0) is specified:
Thread 0 name:  Dispatch queue: com.apple.main-thread
Thread 0 Crashed:
0   libsystem_kernel.dylib            0x327ea010 0x327e9000 + 4112
1   libsystem_kernel.dylib            0x327ea206 0x327e9000 + 4614
// ..
8   MobileNotes                       0x00016c14 0x15000 + 7188
9   MobileNotes                       0x000163f8 0x15000 + 5112

..

  // faulting thread register state is presented:
  // State is architecture specific. For iOS(ARM), r0-r15 and CPSR are shown:
  // OS X would have x86_64 or i386 thread state, similar to LC_UNIXTHREAD
Thread 0 crashed with ARM Thread State:
    r0: 0x00000000    r1: 0x07000006      r2: 0x00000000      r3: 0x00000c00
    r4: 0x00001203    r5: 0xffffffff      r6: 0x00000000      r7: 0x2fe1306c
    r8: 0x00000000    r9: 0x0011b200     r10: 0x07000006     r11: 0xffffffff
    ip: 0xffffffe1    sp: 0x2fe13030      lr: 0x327ea20d      pc: 0x327ea010
  cpsr: 0x400f0010


Binary Images:
  // Listing of process memory space, with all binaries loaded
   0x15000 -    0x43fff +MobileNotes armv7  <53ff805c06ec3aa785e0c0e98b5900b1>
/Applications/MobileNotes.app/MobileNotes
0x2fe14000 - 0x2fe35fff  dyld armv7  <be7c0b491a943054ad12eb5060f1da06> /usr/lib/dyld
0x300b9000 - 0x300c6fff  libbsm.0.dylib armv7  <a6414b0a5fd53df58c4f0b2f8878f81f>
/usr/lib/libbsm.0.dylib
0x301eb000 - 0x301ebfff  libgcc_s.1.dylib armv7  <69d8dab7388b33d38b30708fd6b6a340>
/usr/lib/libgcc_s.1.dylib
......
```

The stack trace of the faulting thread often pinpoints the problem. Even if there are no debugging symbols to tie directly to the source code, it is possible to use a disassembler such as `otool -tV` to figure out the sequence of events leading up to the call trace.

It's interesting to note that Absinthe, the 5.0.1 jailbreak, makes use of the crash log to deduce the address space layout. Because of ASLR, libraries "slide" on iOS, so calling library functions from shellcode can be difficult. The jailbreak intentionally crashes the iOS BackupAgent, inspects its crash log, and deduces the address of `libcopyfile.dylib`.

Changing Crash Reporter Preferences

If you have Xcode, you will find that `/Developer/Applications/Utilities` contains a small application called CrashReporterPrefs. You will see the dialog box shown in Figure 5-2 when you start it.

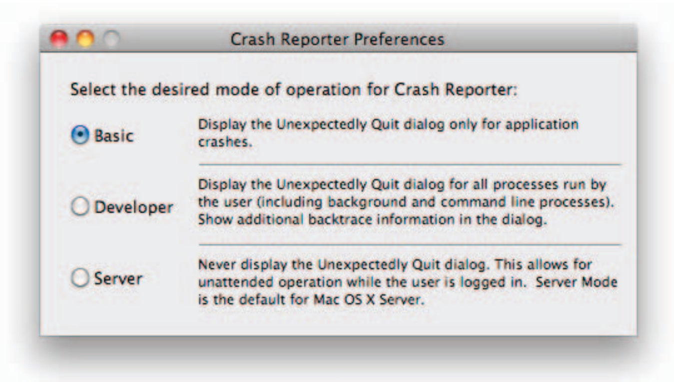


FIGURE 5-2: Crash Reporter preferences

Alternatively, you can use OS X's `defaults(1)` utility to achieve the same purpose, by toggling the `DialogType` property to basic, developer, or server.

> *At this point, you might be asking yourself, "How is it possible to run an application automatically when another crashes?" Doing so in UN*X is hardly trivial, as the parent process would be the only one to receive notification of its child's untimely demise. The mechanism which enables this in OS X and iOS is tied to the exception ports of the Mach task, which underlies the BSD-layer process. This is discussed, along with tasks, in Chapter 11, "Mach Scheduling."*

Application Hangs and Sampling

Sometimes applications don't crash — they merely hang, indefinitely. Oftentimes, this is more frustrating, as the user is left in a state of limbo, gazing at the Spinning Rainbow Wheel of Death (or, more adequately, of paralysis), totally at the mercy of the application, which may or may not choose to become responsive again.

The GUI offers the Force Quit option, which is really just sending a signal to the errant application. Optionally, the user may opt for a "report." The report in question is generated using `spindump(8)`, which probes each and every process on the system and obtains its current call stack (this tool is also part of Lion's `sysdiagnose(1)` tools). The log is then written to the user's (or the system's) `Library/Logs/DiagnosticReports`, similar to CrashReporter logs, but with an extension of `.hang`.

The root user can execute `spindump` manually. Alternatively, it is possible to use `sample(1)` to take a snapshot for a specific process. This tool (which takes the same arguments as `spindump`) can be run by non-root users if the sampling is performed on the user's own processes. The sample log is also in CrashReporter format, providing detailed stack traces and loaded dylib information.

In both cases, the sampling method is similar — the processes are suspended, their stack trace is recorded (`spindump(8)` uses the `stack_snapshot syscall`, described above), and then they are resumed. The sampling interval is usually about 10 milliseconds, and the sampling takes place over a span of 10 seconds. Both settings are configurable.

XCode offers another tool — Spin Control. This small app performs sampling automatically each time the rainbow wheel is displayed (via CoreGraphics). Its only advantage is its call-graph browser, which is somewhat more intuitive than following the textual report. There exists, however, another utility called `filtercalltree(1)`, whose only reason for being is to process call trace logs such as those of `sample(1)` or `malloc_history(1)`, which is a tool we discuss next.

Memory Corruption Bugs

Memory corruption is a common cause for bugs in programs. The main causes of application crashes are buffer overflows (both stack and heap) and heap corruptions. The problem is that, in many cases, the cause and effect are many lines of code apart, and it can sometimes take minutes or more before the bug causes a crash.

Memory Safeguards in LibC

OS X's LibC is highly configurable, and its memory allocation can be controlled by any one of several environment variables, documented in the `malloc(3)` page, as shown in Table 5-8.

TABLE 5-8: LibC's malloc(3) Features

ENVIRONMENT VARIABLE	USED FOR
`MallocLogFile`	Set the `malloc` debugging to write to a file.
`MallocCheckHeapStart` `MallocCheckHeapEach` `MallocCheckHeapSleep/Abort`	Periodically (every ...`Each` allocations) check heap after ...`Start` allocations. If a heap is inconsistent, either sleep (allowing debugging) or `abort(3)` (crashing with `SIGABRT`).

ENVIRONMENT VARIABLE	USED FOR
MallocErrorAbort MallocCorruptionAbort	Call abort(3) (SIGABRT) on any error, or just memory corruption errors
MallocGuardEdges MallocDoNotProtectPrelude MallocDoNotProtectPostlude	Add guard pages before (unless MallocDoNotProtectPrelude is set) and after (unless MallocDoNotProtectPostlude is set) large blocks.
MallocScribble	Fill allocated memory with 0xAA and freed memory with 0x55.
MallocStackLogging MallocStackLoggingNoCompact MallocStackLoggingDirectory	Log all stack traces during malloc operations to /tmp (or to MallocStackLoggingDirectory). Programs such as leaks(1) or malloc_history(1) can then be called. The latter requires NoCompact.

Because the environment variables affect all processes launched when they are set (including the commands that process their output), I recommend that you prefix the traced command with the setting of the variable, rather than export the variable. What's more, exporting variables such as MallocStackLogging can only be countered with "unset," as LibC doesn't really care about its value, so much as it being set.

OS X's memory-leak detection tools, described later, build on these features of LibC to provide extensive capabilities for tracking down memory allocations.

LibGMalloc

If the memory protection features so far do not suffice, OS X offers a special library, libgmalloc.dylib, which can be used to intercept and debug memory allocations. This powerful library works by interposing the allocation functions of LibSystem (as discussed under the "Function Interposing" feature of dyld(1), in Chapter 4). Once the functions are hooked, it becomes easy to replace them with verbose counterparts, which also set more constraints on memory allocation, in the hope of making any slight transgression result in a crash.

Specifically, libgmalloc uses the following techniques:

- **Adding its own custom header to each allocated chunk, which contains debug information recording important allocation details:** The header records the thread ID and backtrace at the time of allocation, along with a constant value ("magic number") of 0xDEADBEEF, which is useful in detecting errors in allocations and reallocations of the same buffer. The header can be seen in Figure 5-3.
- **Allocating chunks on their own pages, making the neighboring page unwritable (if MALLOC_ALLLOW_READS is set), or wholly inaccessible:** The allocated chunk is also pushed to the end of its page (unless MALLOC_PROTECT_BEFORE is set). As a consequence, read/write operations past the end of the buffer automatically become read/write operations past the page boundary, and cause an unhandled page fault, crashing the process on the spot with

a bus error (SIGBUS). Setting the `MALLOC_PROTECT_BEFORE` environment variable flips this behavior to protect against buffer underruns, rather than overruns.

- **Freeing chunks deallocates memory:** The library deallocates its pages on `free()`, once again causing a bus error if a read or write operation is performed on the freed buffer.

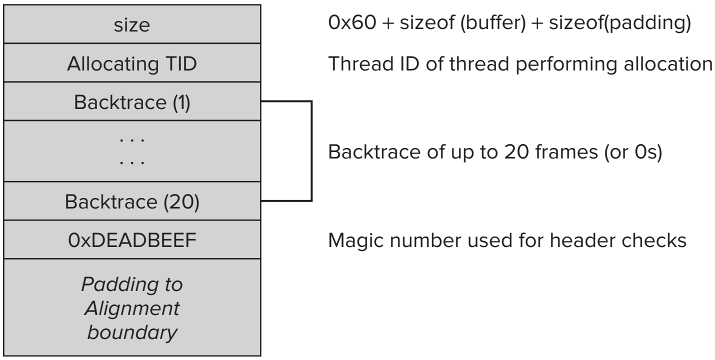


FIGURE 5-3: The GuardMalloc header

The bus faults that occur automatically reveal the presence of a memory handling bug, as it happens, and make debugging relatively simple. By attaching `gdb`, you can pinpoint the crash, and — by inspecting the custom header — work back to the allocation, and either change the buffer allocation parameters or remove the offending operation.

MEMORY LEAKS

Another common application bug is leaking memory. Memory leaks occur when a programmer allocates memory or some object, but neglects to call `free()` or delete. Memory leaks are hard to find because they don't constitute a critical bug. Rather, they slowly weigh on the process' address space, as — once a pointer is lost — there is no way to reclaim the memory.

In 32-bit processes, this can turn into a serious problem because, sooner or later, the leaks can exhaust the available process memory. In 64-bit processes, with their huge address space, it is less of an exigent concern, but can still take a noticeable toll on physical memory (especially in mobile devices) or swap.

> *In addition to the tools described in this section, XCode's Instruments provide an interactive, much more detailed way to sift through the vast amounts of sampling output with a timeline-based GUI. Instruments contain tools for pretty much everything, including specialized tools for tracking memory allocations and leaks (shown in Figure 5-4). The command-line tools, however, do offer the advantage of being lighter and can be run in a terminal.*

Allocations Leaks

FIGURE 5-4: Instruments specifically designed for memory debugging

heap(1)

The `heap(1)` tool lists all the allocated buffers in a given process's heap. The tool is very easy to use — just pass a PID or partial process name. The tool is particularly useful for Objective-C compiled binaries or CoreFoundation-dependent libraries, as it can discern the class names.

leaks(1)

The `leaks(1)` tool walks the process heap to detect suspected memory leaks. It samples the process to produce a report of pointers, which have been allocated but not freed. For example, consider the program in Listing 5-6.

LISTING 5-6: A simple memory leak demonstration

```
#include <stdio.h>
int f()
{
    char *c = malloc(24);
}
void main()
{
    f();
    sleep(100);
}
```

Running leaks on the program produces an output similar to Output 5-8. Note the part in italic, which is displayed if `MallocStackLogging` is set.

OUTPUT 5-8: A leaks(1) generated report for the program from the previous listing

```
morpheus@ergo (/tmp)$ MallocStackLogging=1 ./m &
[1] 8368             # Run process in background to get PID.
m(8368) malloc: recording malloc stacks to disk using standard recorder
m(8368) malloc: stack logs being written into /tmp/stack-logs.8368.m.KaQPVh.index
morpheus@ergo (/tmp) $ leaks 8368
Process:         m [8368]
Path:            /tmp/m
```

continues

OUTPUT 5-8 *(continued)*

```
Load Address:    0x100000000
Identifier:      m
Version:         ??? (???)
Code Type:       X86-64 (Native)
Parent Process:  bash [6519]

Date/Time:       2011-11-22 07:27:49.322 -0500
OS Version:      Mac OS X 10.6.8 (10K549)
Report Version:  7

leaks Report Version:  2.0
Process 8311: 3 nodes malloced for 1 KB
Process 8311: 1 leak for 32 total leaked bytes.
Leak: 0x100100080  size=32  zone: DefaultMallocZone_0x100004000
       0x00000000 0x00000000 0x00000000 0x00000000 ................
       0x00000000 0x00000000 0x00000000 0x00000000 ................
Call stack: [thread 0x7fff70ed8cc0]: | 0x1 | start | main | f | malloc |
malloc_zone_malloc

Binary Images:
    0x100000000 -      0x100000ff7 +m (??? - ???)
<18B7E067-D1EB-30CB-8097-04ED600B3628>
/Users/morpheus/m
    0x7fff5fc00000 -   0x7fff5fc3bdef  dyld (132.1 - ???) <DB8B8AB0-0C97-B51C-BE8B-
B79895735A33> /usr/lib/dyld
...
```

malloc_history(1)

The `malloc_history(1)` tool, which requires `MallocStackLogging` or `MallocStackLoggingNoCompact` to be set, provides a detailed account of every memory allocation that occurred in the process, including the initial ones made by `dyld(1)`. Its report format is very similar to those discussed in `sample(1)` and `leaks(1)`, previously. In fact, using the `-callTree` arguments generates a report that is exactly like `sample(1)`'s, and can be further processed with `filtercalltree(1)`. Additional arguments when displaying the call tree include `-showContent`, which can even peek inside the memory allocated, similar to the `leaks(1)` output shown previously.

This tool can be used to show all allocations in the process (using `-allBySize` or `-allByCount`) and even deallocations (`-allEvents`), demonstrating that there really can be too much of a good thing. A more useful form for tracking memory leaks, however, is to specify just the addresses in question as an argument.

STANDARD UNIX TOOLS

In addition to its proprietary tools, OS X provides the standard UNIX utilities found on other systems, albeit sometimes "tweaked" to deal with OS X idiosyncrasies. This section briefly describes these tools.

Process listing with ps(1)

The standard UNIX command `ps(1)`, used to display the process listing, is naturally available in OS X (and in iOS, when installed as part of the `adv-cmds` package). The term "standard," when applied to `ps(1)`, is somewhat fluid, since the command actually has three versions (BSD, System V, and GNU's). Darwin's `ps(1)`, unsurprisingly enough, closely follows that of BSD, though offers some compatibility with System V's. As in just about any UNIX, `ps(1)` uses most letters of the alphabet (in mixed case) as switches. The useful ones are described in Table 5-9:

TABLE 5-9: Useful switches for ps(1)

SWITCH	USAGE
`-A/-e`	All/every process
`-f`	"full" information, including start time, CPU time, and TTY.
`-M`	Shows threads
`-l`	Long information – including priority/nice, user mode address (paddr) and kernel mode wait address (wchan)
`u`	Classic "top" like display, including CPU and MEM %, virtual size, and resident set size.
`-v`	Similar to "u", but also includes text size and memory limit, among other things.
`-j`	Job information — including session leader

System-Wide View with top(1)

The UNIX `top(1)` command, a key tool for obtaining an ongoing system-wide view, is present in OS X (and iOS), with some modifications. The changes all stem from the adaptation of the tool to the underlying Mach architecture, as it is able to present both the UNIX terms (from XNU's BSD layer) and those of Mach. As `top(1)` is part of Darwin's open source, it can be compiled for iOS as well (and a binary version can be found on Cydia).

`top` dynamically adapts to the terminal window size (via a `SIGWINCH` signal handler) and requires about 210 column terminals for its full splendor. On a standard terminal, you are likely to see something like Output 5-9.

OUTPUT 5-9: top(1) on a standard terminal (82x25)

```
Processes: ## total, # running, ## sleeping, ## threads              HH:MM:SS
Load Avg: 0.72, 0.60, 0.53  CPU usage: 15.56% user, 8.49% sys, 75.94% idle
SharedLibs: 6404K resident, 4900K data, 0B linkedit.
MemRegions: 11835 total, 761M resident, 18M private, 1238M shared.
```

continues

OUTPUT 5-9 *(continued)*

```
PhysMem: 1224M wired, 1709M active, 1034M inactive, 3968M used, 128M free.
VM: 171G vsize, 1043M framework vsize, 796984(0) pageins, 42562(0) pageouts.
Networks: packets: 3041149/3182M in, 2416182/525M out.
Disks: 423708/12G read, 233719/12G written.

PID   COMMAND      %CPU TIME     #TH  #WQ  #POR #MREG RPRVT  RSHRD  RSIZE  VPRVT
5558  top          5.4  00:01.39 1/1  0    24   33    1432K  244K   2012K  17M
5348  Xcode        0.0  00:27.79 9    2    233  873   61M    88M    155M   356M
5346  Image Captur 0.0  00:00.24 2    1    81   74    2184K  10M    7104K  31M
5328  ssh          0.0  00:00.18 1    0    22   24    576K   244K   1844K  17M
5263  vim          0.0  00:00.01 1    0    17   36    520K   244K   1704K  19M
5131  bash         0.0  00:00.11 1    0    17   24    408K   764K   1064K  17M
5128  bash         0.0  00:00.00 1    0    17   25    368K   764K   1064K  9656K
5127  login        0.0  00:00.06 1    0    22   53    536K   312K   1644K  19M
5111  bash         0.0  00:00.04 1    0    17   24    392K   764K   1020K  17M
3206  AppleSpell   0.0  00:00.24 2    1    36   49    608K   5728K  4204K  21M
3194- soffice      0.1  01:27.29 5    1    111  767   38M    19M    88M    83M
2348  iTunesHelper 0.0  00:00.30 3    1    52   74    1068K  4268K  3320K  30M
2077  bash         0.0  00:00.49 1    0    17   24    328K   764K   848K   17M
1167  vmware-vmx   6.0  75:11.73 10   1    142  562   17M    57M    894M   46M
507   Preview      0.0  00:13.68 3    2    112+ 154+  13M+   25M    28M+   38M+
425   bash         0.0  00:00.08 1    0    17   25    280K   764K   624K   9648K
424   login        0.0  00:00.01 1    0    22   53    536K   312K   1548K  19M
```

The OS X `top(1)` is slightly different from the standard GNU `top`, in that it is adapted not only to the BSD nomenclature — `PID`, `UID`, `PGRP`, `SYSBSD`, and so on — but also the Mach one; specifically, Mach regions (`MREG`), messages sent (`MSGSENT`) and received (`MSGRECV`), and Mach traps (SYSMACH) are also viewable. Additionally, because `top(1)` feeds on kernel-provided statistics, it also allows viewing page faults and copy-on-write faults, which the kernel maintains per task.

File Diagnostics with lsof(1) and fuser(1)

Sooner or later, it becomes interesting to see which files are used by a certain processes, or which processes use a certain file. The now ubiquitous utilities of `lsof(1)` and `fuser(1)` can accomplish these, respectively.

`lsof(1)` provides a complementary service to `fs_usage`, described earlier because the latter will see only new file operations and not any existing open files. `lsof(1)` displays a mapping of all file descriptors (including sockets!) owned by a process (or processes). On the other hand, `fs_usage(1)` can run continuously, whereas `lsof` usually generates a single snapshot.

`fuser(1)` provides a reverse mapping — from the file to the process owning it. Its main use is to diagnose file locks or "in use" problems, which most often manifest themselves as a "file system busy" message, which fails a `umount(8)` operation. Using fuser (`-c` on mount points) enables you to see exactly which processes are holding files in the file system and must be dealt with prior to unmounting.

The `lsof` package provided on Cydia for iOS at the time of this writing (33-4) does not work properly, due to incorrect invocation of the underlying `proc_info` system call. The tool accompanying this book, however, works properly.

USING GDB

The GNU Debugger's rich syntax and powerful capabilities have made it the de facto standard debugging tool on all UN*X platforms. Apple has officially ported GDB to Darwin, and it is available for both OS X and iOS, as part of XCode or (in source form) as a tarball from Apple's open source site.

Apple's GDB port, however, is derived from a rather outdated version of GDB — 6.3.50, in 2005. GDB has since long progressed, with the latest version at the time of this writing being 7.4. Apple's GDB fork is also regularly updated with new releases of XCode, resulting in two concurrent branches of GDB: The GNU version, and the Apple official one. The GNU version is, by many reports, "broken," in a sense that many of the Mach-O features, such as fat binaries and PIE, are improperly handled. This section, therefore, focuses on the official Apple port. We assume the reader is familiar with GDB, and discusses the Darwin specific extensions.

GDB Darwin extensions

As discussed throughout this book, while XNU presents a UNIX-compatible persona with full POSIX APIs to user mode, the underlying implementation of the most basic primitives is that of Mach. GDB is aware of the underlying Mach structures, and contains commands suited specifically to display them. The info command contains the options shown in Table 5-10:

TABLE 5-10: Options for the info Command

COMMAND	USAGE
`info mach-tasks` `info mach-task <task>`	Displays a list of all Mach tasks on the system. Roughly speaking, each task corresponds to a PID. Further information can be obtained per task, though this information (`TASK_BASIC_INFO`) is largely useless.
`info mach-threads <task>` `info mach-thread <thread>`	Obtain a list of all Mach threads in a given task. Likewise, further information can be obtained per thread (`THREAD_BASIC_INFO`), which is a little bit more useful than the corresponding `TASK_BASIC_INFO`.
`info mach-regions` `info mach-region <address>`	A `vmmap(1)` like display of all the memory regions in the current debuggee. Alternatively, an address may be specified to seek a particular region.
`info mach-ports <task>` `Info mach-port <task> <port>`	Obtain a list of all Mach ports in a given task. Likewise, further obtain information on a specific port. This command prints out the raw hex values, however, and is therefore less usable.
`get/set inferior-auto-start-dyld`	Controls debugging of `dyld(1)` shared libraries.
`get/set inferior-bind-exception-port`	Controls whether or not GDB takes over the task's exception port. Doing so enables controlling Mach exceptions, even before they are converted to UNIX signals.
`get/set inferior-ptrace[-on-attach]`	Controls the use of the ptrace(2) API to attach to the debuggee.

Ports are explained in Chapter 9. Tasks and Threads are discussed in Chapter 10.

GDB on iOS

The Cydia supplied port of GDB for ARM and iOS is an extremely unstable one, and often crashes. Apple's own GDB works well, and is actually a fat binary, containing an ARM Mach-O side-by-side the i386 one. If you try it on iOS, however, it will fail, complaining, "Unable to access task for process-id xxx," even if used on non-privileged processes. This is because debugging requires access to the low level Mach task structure, underlying the BSD process.

On a jail broken device, however, just about anything is possible, including working around this annoyance. The call required, `task_for_pid`, can be enabled if the executable requesting it is digitally signed with entitlements (as discussed in Chapter 3), or if The AppleMobileFileSecurity kext is disabled. When debugging through XCode, an intermediary process, `debugserver` (found on the Developer Disk Image), is signed and contains the necessary entitlements (which were demonstrated in Listing 3-7, in that chapter). If the same entitlements are copied onto `gdb`, and it is signed (using a pseudo-signing tool such as Saurik's `ldid`), the result is a fully functional GDB on iOS.

LLDB

With Apple's shift to LLVM-gcc, it has also introduced LLDB as an alternative to GDB. LLDB is, for the most part, similar in syntax to GDB, but is considered more advanced in its debugging capabilities. As GDB is still the more widely known and used of the two, the book relies on it, rather than LLDB, for examples and illustrations.

SUMMARY

This chapter provided an overview of debugging techniques in OS X and iOS, which can be employed to deal with the common issues and troubles plaguing developers: system call and function tracing, memory bugs, sampling the call stack, application hangs, and crashes. The poorly documented system calls of `proc_info` and `stack_snapshot` have been detailed, as have their applications in the OS X debugging tools. The chapter also served as a refresher to the common UNIX tools that are included in Darwin.

REFERENCES AND FURTHER READING

1. Apple TN2124 — Mac OS Debugging Magic
2. Apple TN2239 — iOS Debugging Magic
3. Gregg and Mauro, *DTrace*: *Dynamic Tracing in Oracle Solaris*, *Mac OS X and FreeBSD*. (New Jersey: Prentice Hall, 2011)
4. Apple TN2123 — Crash Reporter
5. Apple TN2151 — iOS Crash Reports

6

Alone in the Dark: The Boot Process: EFI and iBoot

The previous chapters have covered the basic aspects of system operation. We now turn our attention to the boot process. Booting is that often overlooked aspect of system startup, which occurs from the moment the machine is powered on, until the CPU starts executing the operating system code. At this most nascent stage, the CPU executes standard startup code. The code is meant to probe the devices around it, find the most likely operating system, and start it up, with any user-defined arguments.

Whereas other operating systems rely on default, or generic boot loaders, both OS X and iOS use custom boot loaders of their own. In this chapter, we describe in detail the operation of the OS X boot loader, which operates in the pre-boot firmware environment.

Another aspect, closely tied to boot is installation and upgrade. This chapter therefore devotes a section to explaining the installation images of both OS X and iOS.

TRADITIONAL FORMS OF BOOT

Prior to its Intel days, the architecture of choice for Mac OS computers was PowerPC. The PowerPC architecture differs in many ways from Intel, not the least of which being the boot process. Intel-based machines traditionally relied on a Basic Input Output System — a BIOS, whereas PowerPC, like many other systems, employed firmware.

Most PCs, at the time of this writing, still use BIOS, as is evident when a special startup key — usually DEL or F2 — is pressed. The BIOS provides a set of simple menus by means of which the user can toggle board parameters, boot device order, and other settings. This is the BIOS *User Interface*. From its other end, a BIOS has a *processor interface*, which is usually accessible by means of a specialized machine instruction (commonly `Int 13h`). Using this instruction, the CPU can invoke specific BIOS-provided functions for device I/O.

Firmware can be thought of as software, which has been put into a chip, hence it is "firm." The firmware code itself can reside in Read-Only Memory (ROM), or — as is more commonly the case — Programmable Read Only Memory (PROM), or Electronically-Erasable (EEPROM). The latter form makes the firmware read-only, but allows its updating by a process known as *flashing*, in which the ROM as a whole is reinitialized and updated with newer versions.

Firmware and BIOS exist to serve the same underlying task: to load the CPU with some basic bootstrap code. This code is responsible for the Power On Self Test phase, in which the CPU "reaches out" to the various hardware buses, and probes them for whatever devices are present. When a computer is first turned on the CPU is, quite literally, in the dark and needs to "prod" its buses to see what devices are reported there. It is the bootstrap code — BIOS or Firmware — which is responsible for locating the boot device, and execute a boot loader program, which in turn finds the operating system of choice, and passes its kernel any necessary command-line arguments.

Technically, BIOS is a type of firmware, but a distinction is drawn between the two, as firmware is generally perceived to be more advanced and more feature-capable than BIOS. Firmware interfaces — both user and processor — are generally richer than those of a BIOS. The standard PC BIOS is wracked with legacy pains. Its origins are in the old days of XTs and ATs, and thus BIOS is still 16-bit compatible.

BIOS — true to its name — is very basic. Most BIOS supports a very simple partitioning scheme — called Master Boot Record partitioning. The name reflects the fact that virtually all partitioning and boot logic resides in one record — the first 512 bytes of the boot disk. When the system is started, BIOS finds the boot disk — as preconfigured by the user — and starts executing code directly from logical block 0, or cylinder 0, head 0, sector 0. It expects to find exactly 440 bytes of loader code there. Usually, these 440 bytes are very simple and directed. They are:

- Read the partition table (at offset `446` of the very same sector, i.e. 6 bytes later).
- The partition table contains exactly four records, each 16 bytes. One of them should be designated as bootable, or *active* (marked by the most significant bit of the first byte in the record).
- The loader then reads the first sector of the active partition, called the partition boot record (PBR), wherein it expects to find the operating system loader code. In Windows' case, this is where the familiar NTLDR (or, post-Vista, `BootMGR`) can be found.

This type of scheme is hardly scalable. If you've ever tried to install more than one operating system side by side on a BIOS based system, you have no doubt run into problems which affect the bootability of one, or both of the systems. Only one system can be marked as active, which leads to the need of a *boot loader*, which is often third party software. Probably the most famous example of a boot loader is GNU's Grand and Unified Bootloader, affectionately referred to as GRUB, which is the de facto standard in UNIX and BSD. GRUB itself is a BIOS-based program (i.e. running before the operating system has been loaded), that takes over, to offer a boot menu. Boot loaders offer some reprieve, but still cannot get past highly restrictive BIOS limitations.

Traditional BIOS can only access about 1 MB of memory. Even this 1 MB is segmented, as 16-bit can only access 64 K of memory. By using the CPU's segment registers, 64 K can be expanded — but the 1 MB serves as a hard limit, and places severe restrictions on code execution. In fact, of the 1 MB, only the lower 640 K (10 segments) were for general purpose RAM, with the top 384 K usually used for shared video memory.

Additionally, traditional BIOS can't interface with today's advanced graphics. If you've ever paid close attention to the way Windows or Linux boot, you see that they start in text mode, then go into graphics mode — but a limited, VGA mode, wherein the screen resolution is usually 640×480, before the screen resets to a higher resolution. This is because, at first, these operating systems draw on the BIOS to access the graphics card. Only when the processor switches to protected mode, and specific device drivers are loaded, is BIOS no longer necessary.

BIOS is also far from extensible, as is probably evident to PC users who add improved bus controllers, like FireWire and USB 3.0 to their systems. The manufacturer BIOS is very rigid, and — while it is possible to "flash" BIOS, much in the same manner as firmware — this is generally a potentially risky operation, and requires specific updates for various BIOS versions. BIOS has no concept of a driver which could be plugged in, much like a kernel driver is to a running operating system.

If all those limitations are not enough, throw in that BIOS is tightly coupled with the MBR partitioning scheme, which allows for only four bootable, or primary partitions in a disk. Due to the fixed format of the boot sector, BIOS cannot split a disk into more than four partitions. A workaround exists in the form of extended partitions (A trick which enables repartitioning of a primary partition), but extended partitions are unbootable. Another restriction, which is becoming more serious at the time of writing, is BIOS's limitations for disks of up to 2 TB. While, back in the day, 2 TB might have seemed an unimaginably large number, let's also not forget the paradigm at the time was "640 K ought to be enough for everybody." With today's hard drives already offering 2 TB, the partitioning scheme itself is becoming a backward-compatibility induced limitation, which does not scale well to today's, much less tomorrow's standards.

It is these limitations of BIOS, and others, which led Apple to adopt a newer 32- or 64-bit compatible standard of the Extensible Firmware Interface — or EFI. Contrary to BIOS, EFI is a full fledged runtime environment, which offers a far more capable interface during boot, and even later during runtime. XNU, the OS X kernel, relies on many of EFI's features, as is discussed next.

EFI DEMYSTIFIED

With the transition to Intel-based architectures, Mac OS X opted to deviate away from the mainstream BIOS architecture, and be the first major OS to adopt EFI. EFI is more complicated, and was initially more costly than BIOS. Apple's tight control and integration with its hardware, however, allowed it to adopt EFI. Given that OS X on PPC relied on OpenFirmware and its rich feature-set, it was only natural for Apple to seek similar capabilities for use with Intel processors; it found those capabilities in EFI.

EFI started as an initiative by Intel, which carried it forward to version 1.10[1], but later merged it with an open standard called Universal EFI — UEFI. The current version of UEFI (at the time of writing) is 2.3.1[2]. Apple's EFI implementation, however, differs somewhat from both standards, and Apple — as Apple — makes little effort to document its changes. Apple's EFI is mostly compliant with EFI 1.10, but also implements some features from UEFI.

Much of the detail this book leaves off can be found in either of the standards. The reader is encouraged to peruse the standards, though the following sections will cover the basics required for understanding EFI as implemented on Macs.

UEFI is processor-agnostic, and has implementations on Intel platforms (naturally), but also on ARM, as well. In iOS, however, Apple employs a custom boot-loader, called iBoot, which is not EFI-based.

Basic Concepts of EFI

Whereas BIOS is a set, usually closed program, EFI is an *interface*. It can be thought more of as a runtime environment, specifying a set of application programming interfaces which EFI-aware programs can draw on and use. EFI programs are generally boot loaders (like Linux's GRUB, or Apple's `boot.efi`, and Boot Camp, both discussed next), but can be diagnostics routines (like Apple's Hardware Test), or even user programs which were compiled to link with EFI APIs, as you will see later in this chapter. Figure 6-1 shows a view of the EFI architecture:

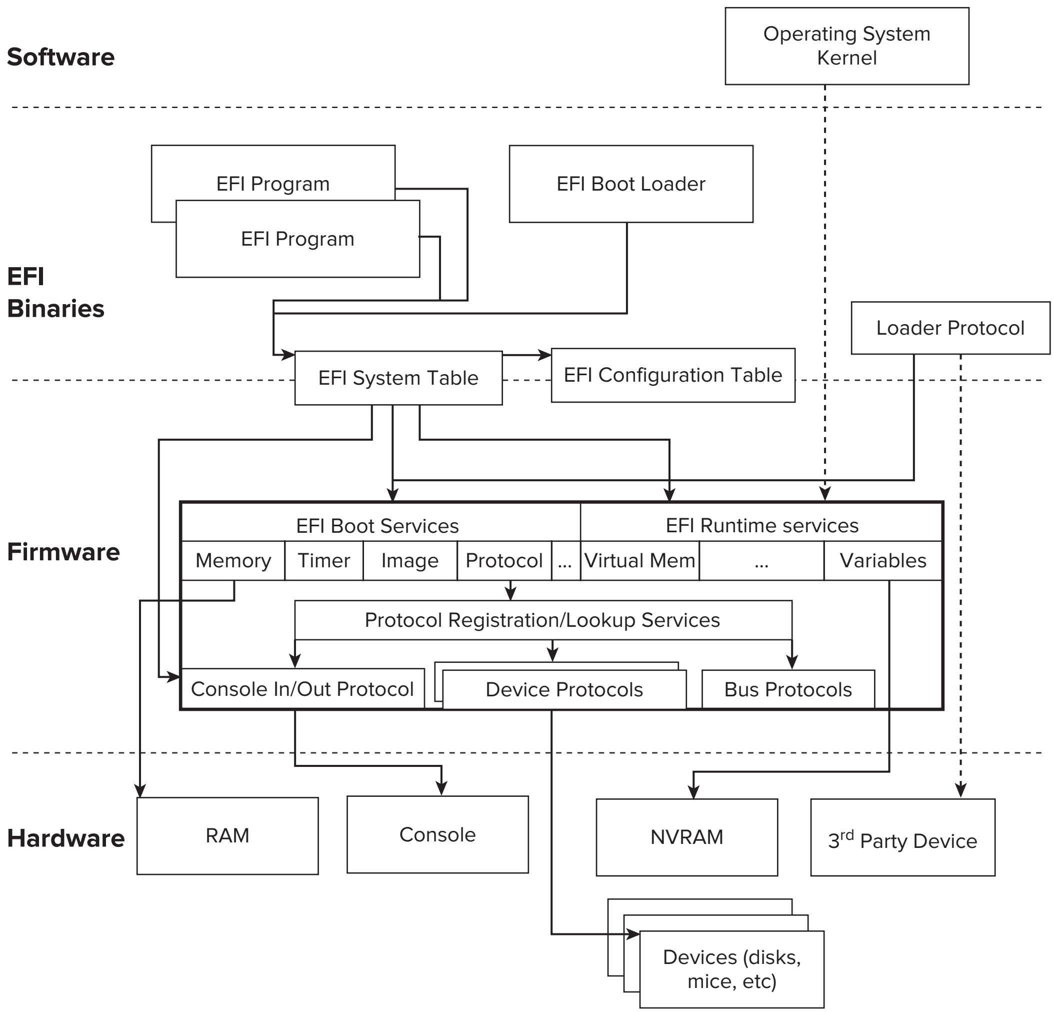


FIGURE 6-1: The EFI Architecture

From the developer's perspective, an EFI program — be it application, boot loader, or driver — is a binary, much like any other binary program. Unlike OS X's Mach-O or Linux's ELF, however, EFI binaries are all PEs — Portable Executables, adhering to the Microsoft adopted executable format, which is native to Windows.

Apple is slightly different in their EFI implementation. For one, Apple wraps their EFI binary with a custom header, not unlike the fat header discussed in the previous chapters. This way, the same binary can be used for 32-bit and 64-bit architectures.

Additionally, Most EFI implementations provide a shell — i.e. a command line interface. Apple's implementation, however, does not. It only responds to specific key presses, which the user should input after the system startup sound (the chime heard when Macs of all kinds boot). Apple, instead, provides their own custom EFI loader, called `boot.efi`, which is a closed-source program.

An EFI binary has a `main()` — just like any old C program, but instead of the familiar command line arguments, EFI binaries all implement the same prototype:

```
typedef EFI_STATUS     (EFIAPI *EFI_IMAGE_ENTRY_POINT)
     (IN EFI_HANDLE ImageHandle,
      IN EFI_SYSTEM_TABLE SystemTable);
```

This is really just to say that EFI binaries accept two parameters from the EFI environment:

- **The EFI Handle —** To the image itself, by means of which it can query the runtime for various details.
- **The EFI System Table —** which is a pointer to a master table, from which all EFI standard handles and runtime API pointers can be obtained.

EFI binaries, like normal C programs, return a status code — an integer, cast as an `EFI_STATUS`. The meaning of this status code, however, is different than in C. Returning `EFI_SUCCESS` clears the program from memory upon exit, whereas returning a non success value leaves it resident in memory.

The handle to the image itself is generally of little use to a program, but the important parameter lies in the `EFI_SYSTEM_TABLE` pointer, which is a structure defined as shown in Listing 6-1:

LISTING 6-1: The EFI system table

```
typedef struct {

 EFI_TABLE_HEADER
 { UINT64 Signature; // Constant
   UINT32 Revision;
   UINT32 HeaderSize; // Sizeof the entire table;
   UINT32 CRC32;      // CRC-32 of table
   UINT32 Reserved;   // set to 0
  } Hdr;
 CHAR16 *FirmwareVendor;                    // For Apple EFI, "Apple"
```

continues

LISTING 6-1 *(continued)*

```
  UINT32 FirmwareRevision;                       // Model dependent
  EFI_HANDLE ConsoleInHandle;                    // stdin handle for binary
  EFI_SIMPLE_TEXT_INPUT_PROTOCOL *ConIn;         // output operations
  EFI_HANDLE ConsoleOutHandle;                   // stdout handle for binary
  EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL*ConOut;        // output operations
  EFI_HANDLE StandardErrorHandle;                // stderr handle for binary
  EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *StdErr;       // output operations (q.v ConOut)
  EFI_RUNTIME_SERVICES *RuntimeServices          // Pointer to Runtime servers
  EFI_BOOT_SERVICES*BootServices                 // Pointer to boot time services
  UINTN NumberOfTableEntries;                    // entries in configuration table
  EFI_CONFIGURATION_TABLE*ConfigurationTable     // system configuration table
} EFI_SYSTEM_TABLE;
```

The `EFI_SYSTEM_TABLE` allows a binary to obtain handles for what every C program takes for granted — standard input, standard output, and standard error. Unlike C, however, there is no `<stdio.h>`, or even `<unistd.h>`, with which to process input and output operations. For this, EFI defines various *protocols*. A protocol is nothing more than a `struct` of function pointers, each defining an operation. EFI uses such protocols for input and output on the console, as well as on more complicated devices.

In addition to the handles and their respective protocols, the system table defines a configuration table, which points to vendor specific data, and two other important tables for the various services. These are discussed next.

The EFI Services

As an *interface*, EFI provides just that — APIs for EFI binaries to use, in order to access basic hardware primitives. These services are classified into two groups — *Boot Services*, and *Runtime Services*.

EFI Boot Services

Boot Services are available while the system is still within the environment of EFI, and up to the point where a special function, aptly called `ExitBootServices()` is called. Boot Services provide access to memory and various hardware, as well as launching EFI programs, when these resources are considered to be "owned" by the firmware. Once `ExitBootServices()` is called, however, Boot services cease to be accessible. Usually, this function is called right before control — and ownership of these resources — is transferred to an operating system kernel.

The boot environment is surprisingly rich — well above and beyond what one would have expected of BIOS. The environment is rich, supporting multi-tasking with preemption, event notification, memory management, and hardware access.

The Boot Services are stored in a `BOOT_SERVICES_TABLE`, a pointer of which is obtained from the `EFI_SYSTEM_TABLE`. The services in this table can generally be classified into several categories, as shown in Table 6-1:

TABLE 6-1: Boot services provided by EFI

CATEGORY	SERVICE CALLS	USED FOR
Memory management	AllocatePages FreePages GetMemoryMap AllocatePool FreePool	Allocate/free physical memory, either directly as physical pages or as a more generic allocation from a pool.
Timer/Event functions	CreateEvent SetTimer WaitForEvent CloseEvent CheckEvent SignalEvent CreateEventEx	Event handling functions which allow to create, wait-on or destroy an event. A "Timer," in this context, is an event which fires automatically after a certain timeout. Events can also be set with specific priorities.
Task priorities	RaiseTPL RestoreTPL	Tasks execute at several levels, and using Raise/Restore can modify task priorities dynamically. Events will get masked or delivered, based on task priority.
Hardware access	InstallProtocolInterface ReinstallProtocolInterface UninstallProtocolInterface HandleProtocol RegisterProtocolNotify LocateHandle OpenProtocol CloseProtocol	Access devices by means of specific protocols. (Protocols are a key mechanism for hardware access, and are covered in the following section.)

Of particular importance in the Boot Services is access to hardware. Just like the simple input and output from the `EFI_SYSTEM_TABLE`, EFI further defines the notion of a *protocol*, to encompass the API associated with a particular device, or device class. Protocols are uniquely defined by 128-bit GUIDs, and may be obtained during runtime. The following tables illustrate some of these protocols. Here, too, there are several classes, including:

Console Protocols

These protocols deal with the console device i.e., the peripheral user input/output devices directly connected to the machine: keyboard, mouse, serial port, and screen, but also more sophisticated devices such as touchscreens and graphics adapters. Table 6-2 lists protocols known to be used by Apple in Lion's EFI loader:

TABLE 6-2: Console protocols supported by Apple's EFI loader

EFI_ PROTOCOL	NOTES
`SIMPLE_TEXT_INPUT_PROTOCOL`	Console-based input. Contains the methods `Reset()` — to reset console, `ReadKeyStroke()`, and a `WaitForKey` event to delay execution until user presses a key
`SIMPLE_TEXT_OUTPUT_PROTOCOL`	Console-based output. Contains various methods to output strings, EGA (4-bit) colors, rudimentary cursor control and textual screen setting capabilities
`SIMPLE_POINTER_PROTOCOL`	Basic interface to a mouse. Somewhat akin to `TEXT_INPUT`, provides a `Reset()`, `GetState()` — for mouse x/y/z and button state — and a `WaitForInput` event to delay execution until the user moves the mouse
`GRAPHICS_OUTPUT_PROTOCOL`	Basic graphics display, backward and forward compatible with any display adapter, effectively replacing the VGA standard
`UGA_DRAW_PROTOCOL`	An older version of the `GRAPHICS_OUTPUT_PROTOCOL`

Media Access

These protocols deal with files and file systems, as well as various devices upon which the file systems may be overlaid including tape devices(!). The ones used in Apple's EFI are listed in Table 6-3:

TABLE 6-3: Media access protocols supported by Apple's EFI loader

EFI_ PROTOCOL	NOTES
`LOAD_FILE_PROTOCOL`	Contain only one method (`LoadFile`), to load a file from a device path into a buffer.
`SIMPLE_FILE_SYSTEM_PROTOCOL`	Basic file system access for FAT-based file systems. Apple extends file system support for HFS+, which is the files system of choice for OS X. This protocol contains only one method — `OpenVolume()` — which returns a `FILE_PROTOCOL` to traverse the file system.
`FILE_PROTOCOL`	Returned from `EFI_SIMPLE_FILE_SYSTEM.OpenVolume()`, this allows the basic file operations — Open/Close/Delete/Read/Write, and the like.
`DISK_IO_PROTOCOL`	Provides `ReadDisk`/`WriteDisk` to access disks by logical block I/O.
`BLOCK_IO_PROTOCOL`	Raw block device abstraction.

Miscellaneous Protocols

Table 6-4 lists miscellaneous protocols used in Apple's EFI.

TABLE 6-4: Miscellaneous Protocols supported by Apple's EFI loader

PROTOCOL	NOTES
`DATA_HUB_PROTOCOL`	A protocol defined by Intel for data store and access. Used by EFI producers to fill in data on devices, and is used by `boot.efi` in the construction of the device tree.

UEFI, true to its universal nature, includes protocols for myriad devices and types, including SCSI, iSCSI, USB, ACPI, debuggers. Apple uses only a very small subset of these in their firmware, including some specific ones, which remain private (see Table 6-5):

TABLE 6-5: Protocol GUIDs for proprietary Apple protocols in UEFI

PROTOCOL GUID	USED FOR
`4FE1FC56C32332DFh-0CD249B520DBA5893`	Apple `BeepGen` protocol. This is used in `CoreStorage`, and has one known method — `AppleBeepGenBeep`.
`4A6D89C933BE0EF1h-0B916D58DDC699FBBh`	Apple Event protocol.
`45EEC4E30DFCE9F6-7A5983B61A86AA0h`	Image conversion protocol. Used in rendering bitmap images from the various PNGs used, for example, in the `CoreStorage` GUI.

EFI Runtime Services

Runtime services, like Boot Services, are available while the system is in EFI mode, but — unlike Boot Services — can persist afterwards. This means that they are still accessible after an operating system has loaded. Indeed, XNU — the kernel — sometimes draws on the runtime services.

The runtime services are more limited in scope, as it is assumed that whatever functionality they do not provide is either provided by the `BootServices`, or by whomever assumed direct control of the devices.

As Table 6-6 shows, runtime services include accessing the system time, as well as the environment variables stored in the NVRAM. One good example is the `nvram(8)` command, which communicates with EFI services from the command line (albeit through a system call and, in turn, the I/O kit NVRAM driver). NVRAM variables are used primarily during the system boot, as well as to store persistent data across reboots (like Panic data).

TABLE 6-6: EFI Runtime services

CATEGORY	SERVICE CALLS	USED FOR
Time management	`GetTime` `SetTime`	Get/Set the local time and date
Alarm clock	`GetWakeupTime` `SetWakeupTime`	Get/Set the system built-in wakeup timer
Firmware variables	`GetVariable` `GetNextVariableName` `SetVariable`	Get/Set variables by name, or walk variables by calling `GetNext()`
Miscellaneous	`ResetSystem`	Perform a soft reset of the system

NVRAM Variables

NVRAM are a powerful feature of the firmware interface, and certainly another advantage it holds over the legacy BIOS. They are semantically the same as the environment variables you know from the shell environment, but they exist in a system-wide scope, and are accessible by both the operating system, and the firmware itself.

Generally, NVRAM variables can be classified into the following categories:

- Boot-related variables: are used to figure out which kernel and root filesystem to boot, as well as pass any arguments to the kernel.
- Firmware internal variables: are used by the firmware, but generally ignored by the operating system
- Transient variables: are set and cleared based on a need, but generally do not survive across reboots.

Each variable has associated attributes. The firmware itself is agnostic as to the format or data of the variables — they are nothing more than named containers. In order to mitigate the chance of conflict between variable names, variables can be associated with specific GUIDs. Apple's `boot.efi` uses several such GUIDS (see Table 6-7):

TABLE 6-7: EFI GUIDs present in Apple's boot.efi

GUID	PURPOSE
`EFI_GLOBAL_VARIABLE_GUID` `8BE4DF61-93CA-11D2-AA0D-00E098032B8C` (defined in `<pexpert/i386/efi.h>`)	Generic EFI global variables, defined in section 3.2 of the UEFI spec. The kernel hibernation logic (`IOHibernateIO.cpp`) sets `BootNext` — the boot choice to be used in the next boot, and `Boot%04X` (where `%04X` are four hex digits). `Boot.efi` queries `BootCurrent`, `Boot0081` and `BootNext`.

APPLE_VENDOR_NVRAM_GUID 4D1EDE05-38C7-4A6A-9CC6-4BCCA8B38C1	Used for firmware internal variables, such as FirmwareFeaturesMask, gfx-saved-config-restore-status, PickerEntryReason, and others.
APPLE_BOOT_GUID 7C436110-AB2A-4BBB-A880-FE41995C9F8	Apple specific private GUID used for boot variables. This is also the only GUID which is visible through the nvram(8) command.
4AADBD3C8D63D4FE-0DFC14B97FD861D88	Used for Lion's Core Storage (And therefore not available before 10.7). Used internally with variables like "DirtyHalt-FromRevertibleCSFDE", and "last-oslogin-ident" which handle Core Storage disk encryption conversion errors, and "corestorage-passphrase".

<pexpert/i386/efi.h> also defined APPLE_VENDOR_GUID - {0xAC39C713, 0x7E50, 0x423D, {0x88, 0x9D, 0x27,0x8F, 0xCC, 0x34, 0x22, 0xB6} } — but there are no references to it in the kernel, nor apparently in the boot.efi.

The list of all variables is far more extensive than these meager pages can contain. Table 6-8, however, lists some variables of specific interest.

TABLE 6-8: EFI variables in the APPLE_BOOT_GUID space

EFI VARIABLE (APPLE_BOOT_GUID)	PURPOSE
SystemAudioVolume	Last setting of volume on Mac. EFI needs this in order to sound the familiar boot chime at just the right volume. Try changing the volume setting, and use 'nvram -p'.
boot-args	Arguments that will be passed to the kernel proper, upon invocation. These are appended to any Kernel Flags in com.apple.Boot.plist.
efi-boot-file-data efi-boot-kernel-cache-data efi-boot-mkext-data efi-boot-device efi-boot-device-data	The names of the kernel, kernel cache, and Multi Kext cache used in the boot process. (Useful for booting alternate kernel images). These are all set by bless(8), as discussed later.
aapl,panic-info	Set by kernel on crash, to save panic information in a packed format to the only safe place — the NVRAM. Unpacked upon next reboot by Core Services' DumpPanic. This variable is ignored by boot.efi.
boot-image boot-image-key boot-signature	Used when setting hibernation parameters. Defined in iokit/IOKit/IOHibernatePrivate.h and used in IOHibernateIO.cpp. The former header file also defines other memory-related keys, but those are left unused.
fmm-hostname	The machine host name, if set.

Using the `nvram(8)` command will give you access to the firmware's variables from user mode. The only visible variables, however, are the ones in Apple's Boot GUID. To get a better view as to the specific NVRAM variables in your Mac, you can download the `EFIVars.efi` utility from the book's website. Bear in mind, however, that in order to run EFI binaries on your Mac, you will need to first drop into a custom EFI shell (using an alternate booter like `rEFIT`, described later in the section titled "Count Your Blessings").

An alternative way to see the NVRAM variables is via the I/O Registry Explorer, or the command line utility `ioreg`. Again, this will only display those in the `APPLE_BOOT_GUID`.

If you peek at the XNU source code, in `iokit/Kernel/IONVRAM.cpp` you can find an array, `gOFVariables`, containing many of the legacy variables that were previously used in OpenFirmware. This array is also present in iOS kernels.

OS X AND BOOT.EFI

Even though Apple's EFI implementation is closed source, because it is still an EFI binary, it can be inspected quite easily. In addition, it is filled with meaningful debugging information, from which one can figure out its stages of operation.

Recall that Apple deviates from the verbatim EFI standard — and, indeed, one can see the very first deviation in the very format of Apple's EFI executable. Whereas a normal EFI binary begins with a PE header, an Apple EFI binary has a fat like header.

Consider the boot.efi from a Lion boot volume — `/System/Library/CoreServices/boot.efi` — looks something like Output 6-1:

OUTPUT 6-1: A hex dump of Lion's boot.efi

```
morpheus@minion (/)> od -A x -t x4 /System/Library/CoreServices/boot.efi
0000000        0ef1fab9       00000002       01000007       00000003
0000010        00000030       0006c840       00000000       00000007
0000020        00000003       0006c870       00064e40       00000000
---------------------------------------------------------------------------
0000030        00905a4d       00000003       00000004       0000ffff
...
0000070        0eba1f0e       cd09b400       4c01b821       685421cd
0000080        70207369       72676f72       63206d61       6f6e6e61
0000090        65622074       6e757220       206e6920       20534f44
...
...
006c860        624de04e       bd2b36a3       238d05f5       29d04881
--------------------------------------------------------------------------
006c870        00905a4d       00000003       00000004       0000ffff
006c880        000000b8       00000000       00000040       00000000
006c890        00000000       00000000       00000000       00000000
```

To decipher the header, we consult Table 6-9:

TABLE 6-9: EFI binary header fields

OFFSET	FIELDS (LITTLE ENDIAN!)	VALUE
`0x00`	Signature	EFI Magic value (constant `0xEF1FAB9`)
`0x04`	`NumArchs`	Number of architectures in this fat binary
`Arch+0`	Arch type	Type of processor (`0x00000007 = CPU_TYPE_X86`) (`0x01000007 = CPU_TYPE_x86_64`)
`Arch+4`	Arch subtype	Subtype of processor (`0x00000003 = CPU_SUBTYPE_I386_ALL`)
`Arch+8`	Offset to executable	Offset to executable's PE header, from beginning of this file
`Arch+C`	Length of executable	Length of the executable's binary
`Arch+10`	Alignment	Alignment, if any

In the example from Output 6-2, the EFI binary contains two architectures, which are concatenated one after the other (no alignment padding necessary). The `00905a4d` you can see corresponds to the PE signature — `MZ` (`4d5a`, but remember Intel endian-ness).

Flow of boot.efi

Apple meticulously stripped their boot.efi binary, so a disassembly only reveals one exported function — start. A disabled debug feature, however, has consistently (or, at least until the time of writing) been providing a fairly good idea of its flow. This is discussed next

Get EFI Services Pointers, Query CPUID

The first step of `boot.efi`, like any EFI program, is to obtain and hold in global variables a pointer to the EFI RuntimeServices. Then, using the cpuid assembly instruction, it checks for the presence of the AESNI bit.

InitializeConsole

The next step, `initializeConsole,` uses the `RunTimeServices` pointer to query the `Background Clear` NVRAM variable (from the `APPLE_VENDOR_NVRAM_GUID`). Then, after getting a call to `Locate-Protocol() CONSOLE_CONTROL_PROTOCOL`, it calls its `GetMode()` to obtain the current console mode.

Lion Specific Initializations

Lion calls an Apple proprietary protocol with the Mac OS X 10.7 argument, and gets/sets the ROM and MLB variables in the `APPLE_VENDOR_NVRAM_GUID`.

InitDeviceTree

The next step in the boot process is the initialization of a hierarchical, tree-based representation of the devices in the system. This representation, hence called the *Device Tree*, is later passed to the kernel in one of the members of the argument structure. XNU itself doesn't care much about this tree, but the `IOKit` subsystem relies heavily on it.

The device tree is visible in `IOKit` through a special "plane" called the `IODeviceTree` plane. The concept of device planes will be explained in depth in the chapter dealing with `IOKit`. But — for a quick idea — you can show the device tree using the `ioreg(8)` command, telling it to focus on said plane, as shown in Listing 6-2:

LISTING 6-2: A dump of the OS X device tree

```
# Using ioreg to dump the device tree:
# -p: focus on the IODeviceTree plane
# -w 0: don't clip output.
# -l: list properties
# grep -v \"IO : discard occurrences of "IO in the output -
#                i.e. disregard I/O kit properties

morpheus@Ergo (/)$ ioreg -w 0 -l -p IODeviceTree | grep -v \"IO
+-o Root  <class IORegistryEntry, id 0x100000100, retain 11>
  | {
  |   … the Root entry is the IO Plane root, not the device tree root …
  |     I/O Kit planes are discussed in depth in the chapter dealing with I/O Kit
  | }
  |
  +-o /  <class IOPlatformExpertDevice, id 0x10000010f, registered, matched, active,
 busy 0 (155183 ms), retain 25>
    | {
    |   "compatible" = <"MacBookAir3,2">
    |   "version" = <"1.0">
    |   "board-id" = <"Mac-942C5DF58193131B">
    |   "serial-number" = <…..>
    |   "clock-frequency" = <005a6b3f>
    |   "manufacturer" = <"Apple Inc.">
    |   "product-name" = <"MacBookAir3,2">
    |   "system-type" = <02>
    |   "model" = <"MacBookAir3,2">
    |   "name" = <"/">
    | }
    |
    +-o chosen  <class IOService, id 0x100000101, !registered, !matched, active, busy 0,
 retain 5>
    | | {
    | |   "boot-file-path" = <04045000… >
    | |   "boot-args" = <"arch=x86_64">
    | |   "machine-signature" = <00100000>
    | |   "boot-uuid" = <"55799E60-4F79-2410-0401-1734FF9D9E90">
    | |   "boot-kernelcache-adler32" = <aa19789d>
    | |   "boot-file" = <"mach_kernel">
    | |   "name" = <"chosen">
```

```
    | |   "boot-device-path" =  < .. >
    | | }
    | |
    | +-o memory-map  <class IOService, id 0x100000102, !registered, !matched, active,
 busy 0, retain 6>
    |     {
    |       "name" = <"memory-map">
    |       "BootCLUT" = <00a0100200030000>
    |       "Pict-FailedBoot" = <00b0100220400000>
    |     }
    |
    +-o efi  <class IOService, id 0x100000103, !registered, !matched, active, busy 0,
retain 7>
    | | {
    | |   "firmware-revision" = <0a000100>
    | |   "device-properties" = <5d09..0000010000000  …06d0065000000500000057>
    | |   "firmware-abi" = <"EFI64">
    | |   "name" = <"efi">
    | |   "firmware-vendor" = <4100700070006c0065000000>
    | | }
    | |
    | +-o runtime-services  <class IOService, id 0x100000104, !registered, !matched,
active, busy 0, retain 4>
    | |   {
    | |     "name" = <"runtime-services">
    | |     "table" = <18ae99bf00000000>
    | |   }
    | |
    | +-o configuration-table  <class IOService, id 0x100000105, !registered, !matched,
active, busy 0, retain 12>
    | | | {
    | | |   "name" = <"configuration-table">
    | | | }
    | | |
    | | +-o EB9D2D31-2D88-11D3-9A16-0090273FC14D  <class IOService, id 0x100000106,
!registered, !matched, active, busy 0, retain 4>
    | | |   {
    | | |     "name" = <"EB9D2D31-2D88-11D3-9A16-0090273FC14D">
    | | |     "guid" = <312d9deb882dd3119a160090273fc14d>
    | | |     "table" = <00a071bf00000000>
    | | |   }
    | | |
    | | +-o 8868E871-E4F1-11D3-BC22-0080C73C8881  <class IOService, id 0x100000107,
!registered, !matched, active, busy 0, retain 4>
    | | |   {
    | | |     "alias" = <"ACPI_20">
    | | |     "name" = <"8868E871-E4F1-11D3-BC22-0080C73C8881">
    | | |     "table" = <14a096bf00000000>
    | | |     "guid" = <71e86888f1e4d311bc220080c73c8881>
    | | |   }
    | | |
    | | +-o EB9D2D30-2D88-11D3-9A16-0090273FC14D  <class IOService, id 0x100000108,
!registered, !matched, active, busy 0, retain 4>
    | | |   {
```

continues

LISTING 6-2 *(continued)*

```
        | | |     "alias" = <"ACPI">
        | | |     "name" = <"EB9D2D30-2D88-11D3-9A16-0090273FC14D">
        | | |     "table" = <00a096bf00000000>
        | | |     "guid" = <302d9deb882dd3119a160090273fc14d>
        | | |   }
...
```

Allocate Memory for Kernel Call Gate

The kernel needs to be loaded from the boot-device into memory, and in order to do that, memory has to be allocated. The address of the kernel call gate resides in a global variable.

Several Additional Initializations

`InitMemoryConfig`, `InitSupportedCPUTypes`, and several other functions are called here.

Check for Hibernation Resume

`CheckHibernate` is a function which resumes the system from hibernation, if previously hibernated. If this is the case, this overrides the rest of the flow.

Process Boot Keys

`ProcessOptions` is a key function in the boot loader, responsible for figuring out all the various boot options, and eventually consolidating them into the kernel command line.

`ProcessOptions` checks the keyboard for any input keys. Apple's HT1533[3] lists the startup key combinations supported, and shown in Table 6-10:

TABLE 6-10: Intel Mac Boot-Time Keystrokes

KEYSTROKE	PURPOSE
C	Boot from CD/DVD
D	Run diagnostics — Apple Hardware Test
N	Netboot
T	Target disk mode
Option (ALT)	Display "picker" (Startup manager boot device selections)
SHIFT	Safe mode (equivalent to `boot-args -x`)
Command-R	Recovery mode (Lion only)
Command-S	Single user mode (equivalent to `boot-args -s`)
Command-V	Verbose mode (equivalent to `boot-args -v`)
3+2/6+4	Boot in 32-bit/64-bit mode

The main file used by `ProcessOptions` is `com.apple.Boot.plist`. This file, located in `/Library/Preferences/SystemConfiguration`, is the main property list used by `boot.efi`, and its man page (`com.apple.Boot.plist(5)`) provides the only documentation of note provided by apple for the boot loader, at all.

Apple documents the following parameters in the man page, as shown in Table 6-11:

TABLE 6-11: Documented boot parameters for com.apple.Boot.plist

PARAMETER	PURPOSE
`Kernel`	The name of the kernel image (by default, `mach_kernel`)
`Kernel Cache`	The path to a prelinked kernel — both kernel and kernel extensions in one big file
`Kernel Flags`	Arguments merged with `"boot-args"` from the NVRAM and passed to kernel as command line
`Kernel Architecture`	Either `i386` or `x86_64`. Can also be set as a `Kernel Flag (arch=)`
`MKext Cache`	The path to a `MultiKExt` cache, containing packaged kernel extensions (mostly drivers) to be loaded with the kernel
`Root UUID`	Unique identifier of filesystem to mount as root

The documentation neglects to mention the following, more colorful parameters, as shown in Table 6-12:

TABLE 6-12: Undocumented boot parameters for com.apple.Boot.plist

PARAMETER	PURPOSE
`Background Color`	Set background color for boot
`Boot Logo`	Path to an image for boot. This can be any PNG — Apple's EFI contains a specialized protocol for BMP conversion
`Boot Logo Scale`	Scale factor for boot logo
`RAM Disk`	Ram Disk Image. Like many UNIX kernels, XNU can be set to boot up with a filesystem image loaded into RAM, which functions as an initial root-file system. OS X rarely uses this option, but iOS relies on it when booting in recovery or update modes.

Path names in NVRAM variables are all specified with backslashes (\) instead of slashes (/) — as these arguments are processed by EFI, not the kernel.

Lion: Check CPU Is Not 32-bit Only

In Lion and later, the boot loader calls a function whose sole work is ensuring the CPU is 64-bit capable. By using the Intel `cpuid` assembly instruction, the function makes sure the CPU is not 32-bit mode only. If the CPU cannot handle 64-bit mode as well, EFI boot fails with a message stating, "this version of OS X is not supported on this platform."

This is really an artificial restriction, and the real reason Apple says Lion will not run on 32-bit only CPUs. The Lion binaries themselves are fat binaries, and even the kernel contains a 32-bit image. Starting with Mountain Lion, however, it seems that the kernel will be 64-bit only.

Lion: Check Core Storage

Lion also introduces support for CoreStorage, Apple's logical volume partitioning. If core storage is detected, the boot loader gets the partition ID and EFI handle, and then calls `LoadCoreStorageConfiguration()` to obtain the Core Storage parameters, and `UnlockCoreStorageVolumeKey()`, in case the Core Storage volume is encrypted.

SetConsoleMode

This function initializes the console to graphics mode.

DrawBootGraphics

Draws the familiar boot logo, and the animated circle. A call to an internal function, `DrawAnimation`, handles the latter by creating an EFI timer event, set to fire every 100 ms and installing a draw function as a callback.

LoadKernelCache

This function is responsible for locating and loading the pre-linked kernel, if any. This function internally calls `LoadKernel`, which can load a standard (i.e. non-pre-linked) kernel, as well. Internal functions here deal with the Mach-O format of the kernel, and parse the various load commands.

InitBootStruct

The kernel only accepts one argument — a pointer to a boot structure, which is a fairly hefty `struct` containing all the parameters the kernel needs to know — from its command line arguments (from the `boot-args` and `com.apple.Boot.plist`), to the device tree and other EFI-borne arguments. This structure is described in detail in the following section, "Booting the Kernel." `InitBootStruct` allocates and initializes this structure, which occupies a single page (4 K) in memory.

LoadDrivers

This function loads the various device drivers — KEXTs — into the kernel from `/System/Library/Extensions.mkext`, if found.

LoadRamDisk

If XNU was loaded with a `RAMDisk`, this function loads the `RAMDisk` into memory, so it is available to the kernel without the need for any drivers. It also sets the `/chosen/memory-map` `RAMDisk` attribute, which signals to XNU that a RAMDisk is ready for loading. If a RAMDisk is used, InitBootStruct, called previously, also sets the boot-ramdmg-size and boot-ramdmg-extents properties, which in turn are used by IOKit to detect the RAMDisk.

StopAnimation

Stops the EFI boot animation, by closing the Animation event set when the animation was started, and clearing the progress animation (by drawing a rectangle over it).

FinalizeBootStruct

This function wraps up the boot struct argument to the kernel (by filling in final details like the video parameters). Just before returning, this function also exits the Boot Services.

Jump to Kernel Entry Point

Finally, Start attempts to jump to the kernel gate (the same one which was allocated in the beginning). If it succeeds, this will never return. Otherwise, it exits with error 8xxxx15h, and sleeps for 10 seconds before exiting Boot Services.

Booting the Kernel

After loading the kernelcache or the kernel proper, boot.efi exits the BootServices, and transfers control to the kernel. The kernel is passed a single argument — a page containing the BootStruct, which was finalized in the last stage, from which the kernel can extract all the data required for its operation. This massive structure in the kernel sources (pexpert/pexpert/i386/boot.h), but also defined in the user-mode include file <pexpert/i386/boot.h>, shown in Listing 6-3:

LISTING 6-3: Boot_args (version 2.0) structure from Lion

```
typedef struct boot_args {
    uint16_t Revision;/* Revision of boot_args structure (Lion: 2, SL: 1) */
    uint16_t Version; /* Version of boot_args structure  (Lion: 0, SL: 6) */

    uint8_t  efiMode;    /* 32 = 32-bit, 64 = 64-bit */
    uint8_t  debugMode;  /* Bit field with behavior changes */
    uint8_t  __reserved1[2];

    char     CommandLine[BOOT_LINE_LENGTH];  /* Passed in command line */

    uint32_t MemoryMap;  /* Physical address of memory map */
    uint32_t MemoryMapSize;
    uint32_t MemoryMapDescriptorSize;
    uint32_t MemoryMapDescriptorVersion;

    Boot_Video  Video;          /* Video Information */

    uint32_t deviceTreeP; /* Physical address of flattened device tree */
```

continues

LISTING 6-3 *(continued)*

```
    uint32_t deviceTreeLength; /* Length of flattened tree */

    uint32_t kaddr;       /* Physical address of beginning of kernel text */
    uint32_t ksize;       /* Size of combined kernel text+data+efi */

    uint32_t efiRuntimeServicesPageStart;
                           /* physical address of defragmented runtime pages */
    uint32_t efiRuntimeServicesPageCount;
    uint64_t efiRuntimeServicesVirtualPageStart;
                          /* virtual address of defragmented runtime pages */

    uint32_t efiSystemTable; /* phys. Addr. of system table in runtime area */
    uint32_t __reserved2;      // defined in the user-mode header as efimode (32,64)

    uint32_t performanceDataStart; /* physical address of log */
    uint32_t performanceDataSize;

    uint32_t keyStoreDataStart; /* physical address of key store data */
    uint32_t keyStoreDataSize;
    uint64_t bootMemStart;
    uint64_t bootMemSize;
    uint64_t PhysicalMemorySize;
    uint64_t FSBFrequency;
    uint32_t __reserved4[734]; // padding to a page (2,936 bytes)

} boot_args;
```

The `boot_args` structure changes in between kernel versions, and its field locations are often shuffled around. A kernel version is therefore closely tied to a corresponding EFI loader version. Apple thus distributes, from time to time, EFI updates, which in part address the compatibility with the kernel. To ensure compatibility, the `boot_args` begin with `Revision` and `Version` fields. Versions up to Snow Leopard used 1.*x* (Snow Leopard used 1.6), and Lion uses version 2.0

Using `DTrace`, it is possible to peek at this structure. The D script in Listing 6-4 relies on the `boot_args` being accessible as a field of a global kernel variable, `PE_State`, and prints them out:

LISTING 6-4: Using dtrace(1) to dump the boot_args structure

```
#! /usr/sbin/dtrace -C  -s
#pragma D option quiet

BEGIN
{
    self->boot_args = ((struct boot_args*)(`PE_state).bootArgs);
    self->deviceTreeHead = ((struct boot_args*)(`PE_state).deviceTreeHead);
    self->video = ((PE_Video ) (`PE_state).video);
```

```
    printf("EFI: %d-bit\n", self->boot_args->efiMode);
    printf("Video: Base Addr: %p\n", self->video.v_baseAddr);
    printf("Video is in %s mode\n", (self->video.v_display == 1 ? "Graphics" : "Text"));
    printf("Video resolution: %dx%dx%d\n", self->video.v_width,
            self->video.v_height, self->video.v_depth);

    printf ("Kernel command line : %s\n", self->boot_args->CommandLine);

    printf ("Kernel begins at physical address 0x%x and spans %d bytes\n",
           self->boot_args->kaddr, self->boot_args->ksize);
    printf ("Device tree begins at physical address 0x%x and spans %d bytes\n",
           self->boot_args->deviceTreeP, self->boot_args->deviceTreeLength);

    printf ("Memory Map of %d bytes resides in physical address 0x%x",
            self->boot_args->MemoryMapSize,
            self->boot_args->MemoryMap);

#ifdef LION
    printf("Physical memory size: %d\n",self->boot_args->PhysicalMemorySize);
    printf("FSB Frequency: %d\n",self->boot_args->FSBFrequency);
#endif
}
```

As you can see, the script doesn't install any probes. In fact, the only reason to use `DTrace`, to begin with, is that it provides the simplest way to enter kernel memory, where the `boot_args` resides. Note, that the addresses in the `boot_args` structure are mostly physical addresses.

Kernel Callbacks into EFI

Recall, that the purpose of EFI is to load the kernel. Yet the kernel still has to interface with EFI, in particular with the runtime services.

The code in XNU handling EFI is in `osfmk/i386/AT386/model_dep.c`. In it, are defined three functions:

- `efi_init()` — This obtains the EFI runtime services from the kernel's boot arguments. This function in turn calls the next function.
- `efi_set_tables_[32|64] (EFI_SYSTEM_TABLE *)` — This function, in either a 32- or 64-bit version, takes as an argument a pointer to the EFI system table, validates its signature and CRC, and retrieves a pointer to the Runtime Services, which it places `int gPEEFIRun-TimeServices`, a global variable.
- `hibernate_newruntime_map (void *map, vm_size_t map_size, uint32_t sys-tem_table_offset)` — This reinitializes the runtime services table following a wakeup from hibernation.

The Mach core, however barely uses EFI — and BSD is totally oblivious to it. It is I/O Kit, on the other hand, which makes extensive use of EFI (and its device tree), as will be discussed later.

Boot.efi Changes in Lion

EFI's role has been significantly enhanced in Lion, with the advent of CoreStorage, and other changes. These include the following:

- **Dropped Features:** Despite Apple's official announcements, kernels in OS X up to and including Snow Leopard kept on maintaining a PPC image along a (very) fat binary. As a consequence, EFI in Snow Leopard still supports a "Kernel Interpreter." This has been dropped in Lion.
- **Core Storage Changes:** Lion brings a major change to storage devices — and to EFI — with its Core Storage services. A key feature of Core Storage is full disk encryption (FDE), which encrypts the entire disk and makes its data inaccessible without a special pass phrase. Because this full disk encryption affects everything — including the OS X kernel itself — Lion's `boot.efi` has been revised to add support for Core Storage password authentication. Lion's EFI boasts a full aqua-like interface to query users for their passwords, including support for VoiceOver(!). To achieve this, it utilizes a private framework, from which it obtains the `PNG` files it renders in the graphic controls. If the user authenticates with EFI (as he or she must, in order to boot), the credentials are carried forward to enable auto-login.

Boot Camp

Another important feature, which is implemented by Apple's EFI, is Boot Camp. This is the name given to Apple's dual boot solution, which allows running non-Apple operating systems (primarily, Windows) on Mac hardware. Because Apple uses its proprietary hardware and relies on EFI — whereas Windows is largely still bogged down in BIOS — Apple made in Boot Camp a complete driver package, to support its specific hardware, and modified its `boot.efi` to allow multi-OS boot. Multi-OS boot can be enabled independently by using a third party EFI boot loader, such `rEFIt` (shown in an experiment later in this chapter).

Count Your Blessings

OS X has traditionally allowed very little access to the firmware — be it the PPC's OpenFirmware or Intel's EFI. Aside from the `nvram(8)` command, the only other tool provided which touches upon the firmware is the `bless(8)` utility.

The `bless(1)` command is a utility meant to control and modify the boot characteristics of the system — essentially, define where and how the system would boot from. It has no less than six modes of operation, shown in Table 6-13.

TABLE 6-13: bless(1) modes of operation

MODE	USED FOR
Folder	Designate a specific directory as the system boot directory
Mount	Designate a file system (volume), rather than a directory. The file system argument is a mounted file system, hence the name.

Device	Designate a volume by `/dev` notation, i.e. when the file system it contains is unmounted.
NetBoot	Set server to boot from, using `-server bsdp://[interface@]a.b.c.d`, where `a.b.c.d` specifies the address of the server, and — optionally — interface specifies the local interface, in case of a multi-homed system. BSDP — the Apple "BootStrap Discovery Protocol" is an extension of DHCPv4 not used or implemented anywhere outside Apple.
Unbless	Revoke the "blessing" from a particular folder, mount, device or network boot.
Info	Merely display information.

Apple keeps `bless` open source, and it is recommended to get the source from Apple's Open Source site, if you want to get more insights as to how `bless` works in each of these modes. The following example shows a quick usage of `bless`:

```
# set bless to demonstrate net boot. Note this is just for a demonstration.
# Real netboot would require a netboot server (and a real IP address)
bash-3.2# bless --netboot --server bsdp://1.2.3.4
bash-3.2# nvram -p
efi-boot-device <array><dict><key>IOMatch</key><dict><key>IOProviderClass</key><string>
IONetworkInterface</string><key>BSD Name</key><string>en0</string></dict><key>
BLMACAddress</key><data>WFXK9EhZ</data></dict><dict><key>IOEFIDevicePathType</key>
<string>MessagingIPv4</string><key>RemoteIpAddress</key><string>1.2.3.4</string></dict>
</array>
efi-boot-device-data
     %02%01%0c%00%d0A%03%0a%00%00%00%00%01%01%06%00%00%15%01%01%06%00%00%00%03%0b%%00XU
%ca%f4HY%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00
%00%03%0c%13%00%00%00%00%00%01%02%03%04%00%00%00%00%00%00%00%7f%ff%04%00
# Quickly set bless back to the safe default!
root@Ergo (/)# bless --setBoot --folder /
root@Ergo (/)# nvram -p
efi-boot-device <array><dict><key>IOMatch</key><dict><key>IOProviderClass</key><string>
IOMedia</string><key>IOPropertyMatch</key><dict><key>UUID</key><string>DADF1195-482F-
423D-B635-CD19BAA4EE47</string></dict></dict><key>BLLastBSDName</key><string>disk0s2
</string></dict></array>
efi-boot-device-data
     %02%01%0c%00%d0A%03%0a%00%00%00%00%01%01%06%00%00%0a%03%12%0a%00%00%00%00%00%00%00
%04%01*%00%02%00%00%00(@%06%00%00%00%00%000#.%1d%00%00%00%00%95%11%df%da/H=B%b65%cd%19
%ba%a4%eeG%02%02%7f%ff%04%00
```

As the example shows, `bless(8)` sets the `efi-boot-device` and `efi-boot-device-data` variables. You can see that these are binary encoded variables (the `%xx` being hexadecimal escape sequences). If these variables are set, `boot.efi` will attempt to boot from them. Otherwise, it will seek the first HFS+ bootable partition it can find. Using `bless` in its informational mode displays the `finderInfo` field of the HFS+ volume, which is an array of eight pointers defining filesystem bootable parameters, shown in Table 6-14,

TABLE 6-14: The FinderInfo field in HFS+

FINDERINFO	NOTES
`0`	Directory ID of bootable system folder. This is an HFS+ catalog node identifier (and `inode #`), and is usually `"2"`, indicating the root folder (/)
`1`	Catalog Node ID of the bootable file. On OS X Intel-based systems, this will be the Catalog Node ID (and `inode #`) of `boot.efi`
`2`	This is the Catalog Node ID of a folder that Finder will automatically open a window to browse (similar to Windows `autorun`)
`3`	Reserved for compatibility with OS 8, or 9. On those systems, it is the same as `finderInfo[0]`
`4`	Unused
`5`	On OS X, the same as `finderInfo[0]`
`6-7`	Both these fields are used together to form a unique, 64 bit volume identifier

```
morpheus@Ergo (/) $ bless -info /
finderinfo[0]:      2 => Blessed System Folder is /
finderinfo[1]: 4600322 => Blessed System File is /System/Library/CoreServices/boot.efi
finderinfo[2]:      0 => Open-folder linked list empty
finderinfo[3]:      0 => No alternate OS blessed file/folder
finderinfo[4]:      0 => Unused field unset
finderinfo[5]:      2 => OS X blessed folder is /
64-bit VSDB volume id:  0x2410197504017D3E
root@Ergo (/)# ls -i /System/Library/CoreServices/boot.efi
4600322 /System/Library/CoreServices/boot.efi
```

Normally, `bless(8)` is one of those utilities that is best left untouched. After all, if it isn't broken, why fix it? Indeed, improper use of `bless(8)` can rend the system unbootable. However, given an EFI binary, even a non-Apple one, it is possible to use bless to bestow the holy power of booting upon it. This is especially useful if you want to inspect your Mac at the firmware level. This is shown in the next experiment.

Experiment: Running EFI Programs on a Mac

Recall, that whereas most EFI vendors provide an EFI shell, Apple does not. Fortunately, it is a simple matter to install a third party shell. There are generally two shells you can consider:

- Intel's EFI toolkit contains a shell, as well as many other EFI binaries which can be used to explore devices, and the firmware itself
- The open source project `rEFIt` contains a shell — but also a simple installer for OS X, which invokes `bless(8)` so that the firmware prefers the `rEFIt` EFI loader over the default `boot.efi`. This program functions as an alternate boot loader, which either lets you proceed normally to boot OS X (the default), or drop to the EFI shell.

> *The sequence carries a small, but non-negligible risk of making your system unbootable. Installing an alternate EFI boot handler can provide you with more insights about EFI, along the lines presented in this chapter, and is generally a simple and safe operation. That said, exercise some caution. You might want to try this in a VM environment first.*

To use the following program, you will need an EFI compiler. This is generally the same as the standard GCC, albeit with different headers, to reflect the EFI dependencies (and not the standard `libc`). GNU has an EFI toolkit you can use for this purpose. Because the programs are compiled to EFI, you can choose any version of the toolkit (for example, Linux, which is easiest to use).

After downloading and installing the GNU EFI Toolkit, you will see that it has an `apps/` directory. This directory of sample applications also contains the Makefile you need to create your own applications, such as the one shown in Listing 6-5:

LISTING 6-5: A sample program to print all the NVRAM variables on a Mac

```
#include <efi.h>
#include <efilib.h>

#define PROTOCOL_ID_ID  \
    { 0x47c7b226, 0xc42a, 0x11d2, {0x8e, 0x57, 0x0, 0xa0, 0xc9, 0x69, 0x72, 0x3b} }

static EFI_GUID SProtId                = PROTOCOL_ID_ID;

  // Simple EFI app to dump all variables, derived from one of the GNU EFI Samples

EFI_STATUS
efi_main (EFI_HANDLE image, EFI_SYSTEM_TABLE *systab)
{
        EFI_STATUS status;
        CHAR16 name[256], *val, fmt[20];
        EFI_GUID vendor;
        UINTN size;

        InitializeLib(image, systab);

        name[0] = 0;
        vendor = NullGuid;

        Print(L"GUID                                Variable Name        Value\n");
        Print(L"=================================== ==================== ========\n");
        while (1) {
                StrCpy(fmt, L"%.-35g %.-20s %s\n");
                size = sizeof(name);
                status = uefi_call_wrapper(RT->GetNextVariableName, 3, &size, name,
                                                  &vendor);
                if (status != EFI_SUCCESS)
                        break;
```

continues

LISTING 6-5 *(continued)*

```
                val = LibGetVariable(name, &vendor);
                if (CompareGuid(&vendor, &SProtId) ==0)
                {
                        StrCpy(fmt, L"%.-35g %.-20s %.-35g\n");
                        Print (fmt, &vendor, name , &val);
                }
                else
                Print(fmt, &vendor, name, val);
                FreePool(val);
        }
        return EFI_SUCCESS;
}
```

To compile this program, simply add it to the Makefile in the `apps/` directory (or overwrite one of the existing samples). The resulting binary should distinctly be an EFI binary:

```
[root@Forge gnu-efi-3.0/apps]# make
/usr/bin/gcc -I. -I./../inc -I./../inc/x86_64 -I./../inc/protocol   -O2 -fpic -Wall -
fshort-wchar -fno-strict-aliasing -fno-merge-constants -mno-red-zone -DCONFIG_x86_64 -
D__KERNEL__ -I/usr/src/sys/build/include -c printenv.c -o printenv.o
/usr/bin/ld -nostdlib -T ./../gnuefi/elf_x86_64_efi.lds -shared -Bsymbolic -L../lib -
L../gnuefi ../gnuefi/crt0-efi-x86_64.o printenv.o -o printenv.so -lefi -lgnuefi
/usr/lib/gcc/x86_64-redhat-linux/4.6.0/libgcc.a
/usr/bin/objcopy -j .text -j .sdata -j .data -j .dynamic -j .dynsym -j .rel \
           -j .rela -j .reloc --target=efi-app-x86_64 printenv.so printenv.efi
rm printenv.so printenv.o

[root@Forge gnu-efi-3.0/apps]# file printenv.efi
printenv.efi: PE32+ executable (EFI application) x86-64 (stripped to external PDB), for
MS Windows
```

Take this binary and drop it into your Mac's EFI partition. The easiest way to do so is to mount the partition while OS X is still running:

```
root@Ergo (/)# mount -t msdos /dev/disk0s1 /mnt   # Mount as a DOS (Fat) filesystem

root@Ergo (/)# ls /mnt                            # Indeed, mount is succesful
.Trashes      .fseventsd         EFI

root@Ergo (/)# du /mnt/EFI                # Show directories
30723   /mnt/EFI/APPLE/EXTENSIONS
8323    /mnt/EFI/APPLE/FIRMWARE           # Apple "Firmware update" .scap files are here
39047   /mnt/EFI/APPLE
39048   /mnt/EFI

root@Ergo (/)# cp efitest.efi /mnt/        # Copy over file to root of partition
```

To run this program, you will need to first install rEFIt[4], as otherwise Apple's `boot.efi` will just boot into OS X. The installation is a straightforward one, and should not in any way hamper your ability to boot normally into OS X. It will, however, give you an option to drop into an EFI shell.

The EFI shell greatly resembles the old fashioned DOS prompt, wherein you can execute the program amidst nostalgic PC EGA 4-bit colors. Rather than use drive letters, use `fs0:` and `fs1:` to access the EFI and the system partitions, respectively (and remember a backslash instead of a slash for directory separators). Running the program from Listing 6-4 will show you all the environment variables your NVRAM contains, as shown in Output 6-2:

OUTPUT 6-2: A dump of the EFI Variables from a Mac Mini:

```
Shell> dir fs0:                   # either ls or dir work
Directory of: fs0:\

 04/01/12   09:30a                 48,354    printenv.efi
 03/23/10   01:07a <DIR>  r           352     EFI

Shell> fs0:\printenv.efi
GUID                                 Variable Name        Value
==================================== ==================== ========
E6C2F70A-B604-4877-85BA-DEEC89E117E PchInit              <B0><FF><8E><D0>A^C

Efi                                  MemoryConfig         RLEX^K
4DFBBAAB-1392-4FDE-ABB8-C41CC5AD7D5 Setup
05299C28-3953-4A5F-B7D8-F6C6A7150B2 SetupDefaults
Efi                                  Timeout              ^E<FF><8E><D0>A^C

AF9FFD67-EC10-488A-9DFC-6CBF5EE22C2 AcpiGlobalVariable   P<FE><8E>
Efi                                  Lang                 eng<8E>
Efi                                  BootFFFF             ^A
Efi                                  BootOrder            <80>
Efi                                  epid_provisioned     ^A
Efi                                  lock_mch_s3          ^A
7C436110-AB2A-4BBB-A880-FE41995C9F8 SystemAudioVolume    h
36C28AB5-6566-4C50-9EBD-CBB920F8384 preferred-networks
36C28AB5-6566-4C50-9EBD-CBB920F8384 preferred-count      ^A
36C28AB5-6566-4C50-9EBD-CBB920F8384 current-network
4D1EDE05-38C7-4A6A-9CC6-4BCCA8B38C1 AAPL,PathProperties0 R^A
7C436110-AB2A-4BBB-A880-FE41995C9F8 aht-results
<dict><key>_name</key><string>spdiags_aht_value</string><key>spdiags_last_run_key</key>
<date>4011-09-16T18:36:02Z</date><key>spdiags_result_key</key><string>
spdiags_passed_value</string><key>spdiags_version_key</key><string>3A224</string>
</dict>7C436110-AB2A-4BBB-A880-FE41995C9F8 fmm-computer-name    Minion
Efi                                  Boot0080             ^A
7C436110-AB2A-4BBB-A880-FE41995C9F8 efi-boot-device-data ^B^A^L<D0>A^C

7C436110-AB2A-4BBB-A880-FE41995C9F8 efi-boot-device
<array><dict><key>IOMatch</key><dict><key>IOProviderClass</key><string>IOMedia</string>
<key>IOPropertyMatch</key><dict><key>UUID</key><string>50DD0659-0F10-4307-860B-
6908BD051907</string></dict></dict><key>BLLastBSDName</key><string>disk0s2</string>
</dict></array>
ShellAlias                           copy                 cp
...
```

The `nvram(8)` command only displays the variables associated with the Apple GUID (7C436110-AB2A-4BBB-A880-FE41995C9F8, as shown in Table 6-7).

You can use the other examples in the GNU EFI toolkit to explore EFI further. Additionally, you can use the EFI programs bundled with `rEFIt` (which should be accessible as `fs1:\efi\tools`), for example `dumpprot.efi`, which will dump all EFI protocols by GUID, and `dumpfv.efi`, which will dump the firmware image into the EFI system partition.

IOS AND IBOOT

Apple's i-Devices do not support EFI, and have a totally different boot process than that described above for OS X. The iOS boot process is custom built by Apple using components not found in any other system, and specifically designed to be hack-proof, so as to discourage "evil" jailbreakers from installing any operating system other than iOS.

The boot process is a multi-stage one, as is shown in Figure 6-2:

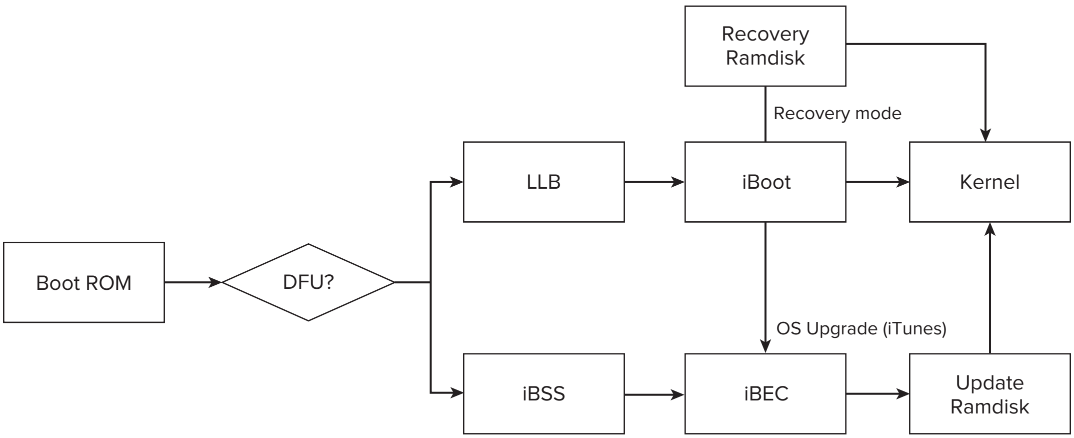


FIGURE 6-2: The iOS Boot process (high-level)

With the exception of the Boot ROM, all these steps are encrypted and digitally signed. This forms a chain of trust right up to the kernel, so that it is (theoretically) impossible to interfere with the boot process and inject any other type of code.

It appears all boot components share a common code base. The NAND FTL (Flash Translation Layer), IMG3 loading, cryptography support, USB support, and ARM low-level exception handling code are all largely identical in them. Each is, in effect, fully self-contained, and rightfully so: They precede the iOS kernel, and therefore cannot rely on its services.

Precursor: The Boot ROM

i-Devices boot using a custom ROM, which is responsible for initializing the device, and loading the Low Level Bootloader, commonly referred to as the LLB. Key in the loading operation is the verification of the digital signature by Apple which ensures the LLB has not been tampered with.

The ROM is part of the device itself and cannot be updated. This works both in Apple's favor and against it: It is extremely difficult to "dump" the ROM in order to reverse-engineer it, and it cannot be tampered with in any way. On the other hand, if it does contain a vulnerability (i.e. a buffer overflow or other code injection vector), there is nothing Apple can do to update it.

In the older generation of Apple's i-Devices — those pre-dating the A5 chip, the bootrom indeed contains an (as yet) undisclosed vulnerability. The "limera1n" exploit, due to the famous hacker geohot, has been successfully used to jailbreak all those devices, in what are known as "untethered" jailbreaks: By exploiting the vulnerability, the check for Apple's signature can be easily bypassed, enabling the uploading of custom iOS images (`.ipsw` files), and even non-iOS images (giving rise to the peculiar movement of iDroid, to install Android on i-Devices in place of iOS). Older bootrom are therefore forward-jailbreakable, as irrespective of any iOS vulnerabilities, the OS image itself can always be patched.

A5-based devices, by contrast, have a newer ROM, one in which the limera1n vulnerability, though undisclosed, was patched. As a consequence, they remain (as of yet) impervious to jailbreaking attempts.

From the boot ROM, two roads diverge: One is the path to normal boot (the default startup of the device) and/or Recovery mode ("Connect to iTunes"). The other is the Device Firmware Update (DFU), which is used to update the iOS image.

Normal Boot

Unless otherwise stated, with no user interaction the device will proceed to boot normally. This is a two-staged process, consisting of the LLB, and iBoot, both of which are responsible for eventually loading the iOS kernel.

Stage I: The Low Level Bootloader

The Low Level Bootloader is the first updateable component of the boot process. It is part of the iOS image, not the device itself, and if you peek at the image you will see it is a file called `LLB.xxxx.RELEASE.img3` in the `Firmware/all_flash/all_flash.xxxap.production/` directory. "xxx" is the model number of the i-Device, shown in Table 6-19, later in this chapter.

The LLB, like all files in the iOS image, is in the IMG3 format. As described under "iOS Software Images," in this chapter, this is an encrypted file format which is also digitally signed by Apple. Following the IMG3 header (64 bytes) is the actual raw code of the LLB. It is loaded by the bootrom into a predefined address, usually `0x84000000`.

LLB will locate its second stage, iBoot, and will attempt to load it. This is done by seeking the image in memory with the tag "ibot." If this fails, LLB contains code to drop to DFU mode, and load iBEC.

Stage II: iBoot

The main boot loader is called iBoot. It is this loader which locates, prepares, and loads the `kernelcache`. Older versions of iBoot also allowed passing command line arguments (from the boot-args variable), but due to the obvious potential for abuse, this has been removed.

> *Using various jailbreaking utilities, it is possible to choose a tethered boot on pre A5 devices, and — by patching iBoot — pass command-line arguments using custom boot-args.*

iBoot gets loaded at address 0x5FF00000. It is a fairly sophisticated boot loader. In addition to the common code shared by all components, it contains a built-in HFS+ driver, which enables it to access the iOS filesystem. iBoot is also multi-threaded, and normally spawns at least two threads:

- A "main" thread, which displays the familiar Apple logo, and proceeds to boot the system, as specified by the `auto-boot` and `boot-command` environment variables. The latter can be set to fsboot (normal file system boot, with or without ramdisk), diags (diagnostics) or upgrade. The boot may be delayed by a `bootdelay` environment variable, in which the user may intervene and abort the process.
- A "uart reader" thread, which Apple likely uses for debugging purposes. The serial ports on i-Devices are present, though require quite a bit of work to enable.[5] This thread is therefore normally idle.

During normal operation, iBoot calls its `fsboot()` function, which mounts the iOS system partition, locates the kernel, prepares its device tree, and boots it. If the boot fails (or is aborted), however, iBoot falls into recovery mode, wherein the main thread spawns several concurrent tasks:

- **The idleoff task:** Times-out after sufficient user inactivity and power off the device
- **The poweroff task:** Forces the device to power off on critical battery
- **The usb-req task:** Handles USB requests from iTunes
- **The usb-high-current and usb-no-current tasks:** Responds to USB charge (these are responsible for changing the battery glyph when the device is connected or disconnected).
- **The command task:** Enables a command-line, console interface over the serial port (that is, assuming you have a serial port connection).

Recovery Mode

Recovery mode is essentially the same as normal boot, with one important difference: The system boots using a ramdisk, rather than the flash based file system that contains the standard iOS image. The ramdisk is a complete in-memory file system, which can be used as an alternate root file system. The flash based file system can then be mounted as a secondary, and system files can be modified or updated.

You can check out the ramdisk for yourself, if you have an iOS image (IPSW). As discussed in the section "iOS Software Images" in this chapter, it is fairly straightforward to unzip and decrypt the ramdisk image. The file is usually the third DMG file in the update. It is not, however, a classic DMG in the sense of one that can be readily mounted by OSX. Rather, it is a raw filesystem image. If you have successfully decrypted it, running the `file(1)` command on it should produce something like the following:

```
morpheus@Ergo (…./iOS)$ file 5.1.restore.ramdisk.dmg
5.1.restore.ramdisk.dmg: Macintosh HFS Extended version 4 data last mounted by: '10.0',
created: Wed Feb 15 05:26:23 2012, last modified: Wed Feb 15 09:10:50 2012, last
checked: Wed Feb 15 08:26:23 2012, block size: 4096, number of blocks: 4218, free
blocks: 0
```

You can also mount the ramdisk easily on OS X by using `hdiutil(1)` with the `imagekey diskimage-class=CRawDiskImage` (this is discussed in Chapter 15, and shown in Output 15-2).

> *Using various jailbreaking utilities, you can boot iOS with an alternate ramdisk (for example, using* `redsn0w -r`*). This is an extremely useful feature for forensics, data recovery and hacking, and hours of fun and profit. It effectively exposes the entire i-Device's filesystem. A good discussion on this can be found in Jonathan Zdziarski's book.*[6]

Device Firmware Update (DFU) Mode

i-Devices have an additional, albeit lesser used boot mode: Device Firmware Update or DFU mode. In this mode, the firmware itself, in NAND flash, is updated. This occurs when a new version of iOS is installed on the device, or during jailbreaking.

iTunes can enable this mode over USB (when you select to upgrade your device), though you can do so as well. To try this, connect your device over USB, and do the following:

- Turn off the i-Device
- Press the power button, and hold. The device should appear to boot, with the Apple logo
- After three seconds, press and hold the home button (while holding the power button). The device screen should clear.
- After ten seconds, let go of the power button, but keep on holding the home button.
- Wait a few more seconds and let go.

If you did this properly, the device screen should remain blank. Otherwise, you might end up in recovery mode ("Connect to iTunes"). If the screen is indeed blank and you connect it over USB, you will see it identify itself as "Apple Mobile Device (DFU Mode)." Getting out of DFU mode is easy — all you need to do is power-cycle the device.

DFU mode involves two images — iBSS and iBEC. The first loads at `0x84000000` (on iOS 5), and is responsible for low-level initialization, and the loading of iBEC. iBEC, like its big brother iBoot, loads at `0x85000000`, and is responsible for handling iTunes upgrade commands over USB.

Downgrade and Replay Attacks

A potential vulnerability in the iOS update process which Apple invests many resources into preventing is in cases where a user might want to install an older version of iOS on the i-Device. As iOS versions progress, Apple plugs and seals various jailbreak openings. From Apple's perspective, all users should consistently upgrade to the latest and greatest versions.

When updating an i-Device, it is not enough to possess a valid iOS image. During the system upgrade (or downgrade) process, a request is made to Apple's secure server, with a Secure Hash value — often referred to as a SHSH. The request includes the device's unique chip id (the ECID value). Though the request is made over plain HTTP (to `gs.apple.com`), the reply is digitally signed. The SHSH is used in the BBTicket (required for base band, or phone logic upgrade) or the APTicket (required for upgrading the iOS firmware).

Prior to iOS 5, it was possible to capture the session, and extract the SHSH blob to save it locally (using TinyUmbrella), or by Cydia. Since then, however, Apple has improved the protocol, by adding a random nonce generated by the device. A random nonce means that now every upgrade authorization request is unique, and therefore saving the SHSH has no effect. This makes downgrading impossible once Apple closes the window on a particular iOS version and configures their server to deny signatures. For this reason, users try to get their hands on new releases of i-Devices sooner, rather than later — as Apple keeps updating iOS on devices with new shipments to their stores.

INSTALLATION IMAGES

Apple pre-installs OS X and iOS on all its hardware. Because both systems are carefully installed with all the required defaults, the average user doesn't bother much with re-installing the system. Hackers and other enthusiasts, however, often perform system wide changes, or careless mishaps as root, which can render the system unbootable. In those cases, the installation media or image needs to be dug up, and the system needs to be installed.

This section covers the installation image format of both OS X and iOS. It is of particular interest to anyone who wants to pick apart the images, extracting specific files or even modifying them to customize the installation image.

OS X Installation Process

The OS X installation begins when an installation DVD or thumb drive is inserted. The Finder automatically shows the root folder, which contains the installation app. If the user chooses to activate the application, things proceed as follows:

Step I: InstallXXX.app

The installation utility for OS X is itself an OS X application. As such, it contains a small executable responsible for the UI, and for starting the installation process. The actual system files in the installation process are shown in Table 6-15:

TABLE 6-15: Files involved in the OS X installation process

FILE	LOCATION	CONTAINS
`boot.efi`	Install media	EFI bootloader for updated kernel
`kernelcache`	Install media	Updated kernel for installed OS
`InstallESD.dmg`	Install media (SharedSupport)	The OS X installation file system image
`BaseSystem.dmg`	InstallESD.dmg	The base system image to be copied over to the target system
`/var/log/install.log`	Target system	Detailed installation log

The executable brings up the familiar Wizard-like interface of the installation (In Mountain Lion, it also dispatches an OpenCL program to the GPU, responsible for GUI effects). The GUI collects the user input choices (e.g. which volume to install on) and also validates the installation with Apple (`osrecovery.apple.com`). Assuming all went well, it proceeds to copy the `kernelcache`, `boot.efi`, and `InstallESD.dmg` to a special directory, `/Mac OS X Install Data`. It then edits `com.apple.Boot.plist` to inform the kernel it is booting with a DMG file, as can be seen in `/var/log/install.log` (Listing 6-6):

LISTING 6-6: Excerpt from install.log detailing the Installation App's work:

```
Sep 25 22:36:49 localhost Install Mac OS X Lion[343]: Extracting files from
/Volumes/Macintosh HD/Mac OS X Install Data/InstallESD.dmg
Sep 25 22:36:50 localhost Install Mac OS X Lion[343]: Extracting Boot Bits from Outer
DMG:
Sep 25 22:36:50 localhost Install Mac OS X Lion[343]: Copied kernelcache
Sep 25 22:36:50 localhost Install Mac OS X Lion[343]: Copied Boot.efi
Sep 25 22:36:50 localhost Install Mac OS X Lion[343]: Ejecting disk image
Sep 25 22:36:50 localhost Install Mac OS X Lion[343]: Generating the
                                                    : com.apple.Boot.plist file
Sep 25 22:36:50 localhost Install Mac OS X Lion[343]: com.apple.Boot.plist: {
            "Kernel Cache" = "/Mac OS X Install Data/kernelcache";            "Kernel
Flags" = "container-dmg=file:///Mac%20OS%20X%20Install%20Data/InstallESD.dmg root-
dmg=file:///Base
System.dmg";
        }
Sep 25 22:36:50 localhost Install Mac OS X Lion[343]: Done generating the
com.apple.Boot.plist file
Sep 25 22:36:50 localhost Install Mac OS X Lion[343]: Blessing /Volumes/Macintosh HD --
/Volumes/Macintosh HD/Mac OS X Install Data
Sep 25 22:36:50 localhost Install Mac OS X Lion[343]: Blessing Mount
Point:/Volumes/Macintosh HD Folder:/Volumes/Macintosh HD/Mac OS X Install Data
plist:com.apple.Boot.plist
Sep 25 22:36:50 localhost Install Mac OS X Lion[343]: *****************************
Setting Startup Disk *****************************
Sep 25 22:36:50 localhost Install Mac OS X Lion[343]: ******           Path:
/Volumes/Macintosh HD
Sep 25 22:36:50 localhost Install Mac OS X Lion[343]: ******     Boot Plist:
/Volumes/Macintosh HD/Mac OS X Install Data/com.apple.Boot.plist
Sep 25 22:36:50 localhost Install Mac OS X Lion[343]: /usr/sbin/bless -setBoot -folder
/Volumes/Macintosh HD/Mac OS X Install Data -bootefi /Volumes/Macintosh HD/Mac OS X
Install Data/boot.efi -options config="\Mac OS X Install Data\com.apple.Boot" -label Mac
OS X Installer
Sep 25 22:36:51 localhost Install Mac OS X Lion[343]: Bless on /Volumes/Macintosh HD
succeeded
```

The kernel flags — by another name, command line arguments — specify to the kernel that it is to mount `InstallESD.dmg` as a container image, which it needs to mount in order to find the actual image to use as a root file system — the `BaseSystem.dmg`. It then `bless`es the boot disk so as to make the system boot from `InstallESD.dmg`. Once the `bless` operation completes successfully, the system reboots automatically, and starts from the new image.

Step II: OSInstaller

OSInstaller is the executable responsible for the unattended portion of installation which occurs once the system reboots. The system by this point has booted into the new OS, and runs its `kernelcache`. The image instructs `launchd(8)` to run OSInstaller, which proceeds to load `minstallconfig.xml` from which it can obtain the installation data. It also brings up `diskmanagementd(8)`, which is used in case any disk "surgery" (i.e. repartitioning) is required.

Once any repartitioning is done, OSInstaller can proceed to install the system, which comes bundled in the form of several packages, as shown in Table 6-16. All these files are in the `/Packages` directory:

TABLE 6-16: OS X installation packages (all in installESD.dmg)

FILE	CONTAINS
`BaseSystemBinaries.pkg`	KEXTs, binaries, and some application binaries
`BaseSystemResources.pkg`	Resources for apps in BaseSystem
`OSInstall.mkpg`	Internationalization resources for Install
`Essentials.pkg`	Most Applications, CoreServices
`Bootcamp.pkg`	Boot-Camp (for dual boot with Windows)
`BSD.pkg`	The BSD subsystem files
`MediaFiles.pkg`	Pictures, Screensavers, etc.
`JavaTools.pkg`	The OS X bundled Java implementation
`RemoteDesktop.pkg`	Remote desktop tools
`SIUResources.pkg`	System Image Utility resources
`AdditionalEssentials.pkg`	More applications, help files, and Widgets
`AdditionalSystemVoices.pkg`	For those users who just can't do without "Princess" and "Deranged"
`AsianLanguagesSupport.pkg`	Specific support for Asian Languages
`<app>.pkg`	Miscellaneous applications, such as Automator, Mail, iChat, DVDPlayer, iTunes, Safari, etc.
`<language>.pkg`	Miscellaneous language support files (anything but English)
`X11User.pkg`	The X/11 Subsystem
`OSInstall.pkg`	Pre and post install scripts (no files)

Before installing, OSInstaller runs an `fsck(1)` on the target volume. As of Lion it also calls on `diskmanagementd` to prepare a recovery volume, which is essentially the `BaseSystem.dmg` from which OSInstaller can boot.

Once the recovery volume is set, OSInstaller uses the PackageKit and Install frameworks to open the package files one by one.

Installing .pkg files

OS X packages, listed in Table 6-17, are descendants of NextSTEP packages. The packages are archives in `xar(1)`, which is an archive format similar to `tar(1)`, but natively supporting compression.

TABLE 6-17: OS X packages

FILE	CONTAINS
`Bom`	Package "Bill Of Materials." Viewable with `lsbom(1)` and can be created with `mkbom(1)`
`PackageInfo`	A property list file specifying the package manifest
`Payload`	The actual package contents, usually compressed with `bzip(1)`
`Scripts`	Pre- and Post-install scripts, usually archived with `cpio(1)` and compressed with `gzip(1)`

The following experiment illustrates working with packages.

Experiment: Unpackaging Packages

Using the OS X installation CD or USB medium, locate the `InstallESD.dmg` file. This file is in the `SharedSupport/` folder of the Installation app. Mount the DMG, using the commands shown in Output 6-3:

OUTPUT 6-3: Locating and mounting the InstallESD.dmg

```
morpheus@Ergo (/Volumes/OS X Mountain Lion)$ cd "Install OS X Mountain Lion.app"
morpheus@Ergo (...OS X Mountain Lion.app)$ cd SharedSupport
morpheus@Ergo (.../SharedSupport)$ open InstallESD.dmg   # could also use hdid(1)
```

Once the dmg is mounted, you can `cd` to its `Packages/` directory, and locate all the packages shown previously, in Table 6-16. Pick a package to continue this experiment with (in our example, we use `BSD.pkg` — you are encouraged to pick another).

Query the package of choice with the `xar(1)` command. Its usage is very similar to `tar(1)`. Create a temporary directory, and extract the package contents to it, as shown in Output 6-4:

OUTPUT 6-4: Extracting a package

```
morpheus@Ergo(/tmp/pkgDemo)$ xar -xvf /Volumes/Mac\ OS\ X\ Install\ ESD\Packages/BSD.pkg
Bom
PackageInfo
Payload
Scripts
```

The bill of materials (bom) can be viewed with `lsbom(1)`:

```
morpheus@Ergo (/tmp/pkgDemo)$ lsbom Bom
.       40755   0/0
./Library       40755   0/0
./Library/Python        40755   0/0     Permissions             Filesize
./Library/Python/2.3    40755   0/0
./Library/Python/2.3/site-packages      40755   0/0   UID/GID                 CRC-32
./Library/Python/2.3/site-packages/Extras.pth   100644  0/0     75      316297377
./Library/Python/2.3/site-packages/README       100644  0/0     119     3290955062
./Library/Python/2.5    40755   0/0
./Library/Python/2.5/site-packages      40755   0/0
./Library/Python/2.5/site-packages/README       100644  0/0     119     3290955062
./Library/Python/2.6    40755   0/0
./Library/Python/2.6/site-packages      40755   0/0
./Library/Python/2.6/site-packages/README       100644  0/0     119     3290955062
./Library/Python/2.7    40755   0/0
./Library/Python/2.7/site-packages      40755   0/0
./Library/Python/2.7/site-packages/README       100644  0/0     119     3290955062
./System        40755   0/0
...
```

The PackageInfo is an XML file, which is rather self explanatory, as shown in Output 6-5:

OUTPUT 6-5: The PackageInfo file of the BSD.pkg

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<pkg-info format-version="2" relocatable="true" deleteObsoleteLanguages="true"
overwrite-permissions="true" identifier="com.apple.pkg.BSD" useHFSPlusCompression="true"
auth="root" version="10.8.0.1.1.1306847324">
  <payload installKBytes="736770" numberOfFiles="33989"/>
  <scripts>
    <preinstall file="preinstall"/>
    <postinstall file="postinstall"/>
  </scripts>
  <groups>
    <group>com.apple.snowleopard-repair-permissions.pkg-group</group>
    <group>com.apple.FindSystemFiles.pkg-group</group>
  </groups>
  <bundle-version>
     <bundle CFBundleVersion="10.8" CFBundleShortVersionString="10.8"
SourceVersion="6001000000000" id="com.apple.xsanmgr-filebrowser"
path="./usr/libexec/xsanmgr/bundles/xsanmgr_filebrowser.bundle"/>
     <bundle CFBundleVersion="1" CFBundleShortVersionString="1.0"
SourceVersion="6001000000000" id="com.apple.xsanmgr-sharing"
```

```
path="./usr/libexec/xsanmgr/bundles/xsanmgr_sharing.bundle"/>
    ...
       </bundle-version>
</pkg-info>
```

The installation scripts — in this case `preinstall` and `postinstall` — are packaged in the `Scripts` file, and can be viewed using `zcat(1)` and `cpio(1)`:

```
morpheus@Ergo (/tmp/pkgDemo)$  cat Scripts | zcat > A
morpheus@Ergo (/tmp/pkgDemo)$  file A
A: ASCII cpio archive (pre-SVR4 or odc)
morpheus@Ergo (/tmp/pkgDemo)$  cpio -ivd < A
.
./postinstall                                  # Perl script to run after install
./postinstall_actions                          # Various shell scripts
./postinstall_actions/dumpemacs.sh
./postinstall_actions/fixnortinst.sh
./postinstall_actions/postfixChrooted
./preinstall                                   # Perl script to prep install
./Tools
```

You can use the `installer(8)` command to install a package automatically. Other package manipulation commands are `pkgutil(1)`, which is somewhat like the Linux `rpm` command (e.g. `pkgutil --pkgs` as the equivalent to Linux's `rpm -qa`) , and `pkgbuild(1)`, which builds packages.

iOS File System Images (.ipsw)

Apple distributes updates to its various iOS devices via iTunes — and, as of iOS 5, over the air as well. If you have ever peeked at iTunes' directory (`~/Library/iTunes`), you no likely got to see directories called `<device> Software Updates`, where `<device>` is the iOS device — iPad, iPhone, or iPod. These directories usually contain the iOS updates for the device, files with an `.ipsw` extension, and the following naming convention:

*Model Generation_Major.Minor_Build*_Restore.ipsw

The file itself, aside from the unusual extension, is nothing more than a simple `.zip` file. It can be opened easily from the command line, or by renaming its extension from `.ipsw` to `.zip`. It contains the files shown in Table 6-18:

TABLE 6-18: Files in an iOS software image

TYPE	FILE NAME	FILE PURPOSE
bat0 bat1	batterylow0*.img3 batterylow1*.img3	Battery low icons. The firmware alternates between these two files to produce the low battery animation.
batF	batteryfull*.img3	Battery full icon.
chg0	batterycharging0*.img3	Battery Charging, 1/3.
chg1	batterycharging1*.img3	Battery Charging, 2/3.
Dtree	DeviceTree.<board>.img3	Device tree for this iDevice, used by iBoot and passed to the kernel.

continues

TABLE 6-18 *(continued)*

TYPE	FILE NAME	FILE PURPOSE
`glyC`	`glyphcharging*.img3`	The glyph for the battery charging.
`glyP`	`glyphplugin*.img3`	The glyph for the battery, plugged in.
`Ibot`	`iBoot.<device>ap.RELEASE.img3`	iBoot — the stage two bootloader.
`Illb`	`LLB.<device>ap.RELEASE.img3`	Low level boot loader (LLB).
`Krnl`	`kernelcache.release.<device>`	The packed kernel and kernel extensions (KEXTs).
`Logo`	`applelogo*.img3`	The familiar apple logo.
`Recm`	`recoverymode*.img`	Recovery Mode image.
`--`	`xxx-<<lowest numbered>>-yyy.dmg`	Root filesystem. (Not an img3, but decrypted using vfdecrypt)
`Rdsk`	`xxx-<<middle numbered>>-yyy.dmg`	Update file Ramdisk.
`Rdsk`	`xxx-<<highest numbered>>-yyy.dmg`	Recovery mode Ramdisk.

As you can see in the table, each file contains a type. This is an embedded four letter (32-bit) magic value used to identify and load the file. In addition, device specific files of iOS (such as the `kernelcache` and firmware files) often contain a variable identifier for the device. The identifiers are shown in the Table 6-19:

TABLE 6-19: Device identifiers

MODEL	DEVICE IDENTIFIER
iPod 2,1	n72
iPod 3,1	n18
iPod 4,1	n81
iPhone 2,1	n88
iPad 1,1	k48
iPhone 4,1	n90
iPad 2,1	k93
iPad 3,1	j1

Apple, however, has tried hard to discourage eager developers from getting their hands on those files, and therefore these files are all encrypted. This encryption — and how to defeat it — is described next.

The Img3 File Format

Apple really doesn't want anyone messing with iOS, and is making a genuinely noble effort to keep the files from prying eyes. While the `ipsw` is a simple zip archive, all its individual files are in a custom encrypted format, known as IMG3 — each with its own keys, with varying keys between devices! And "all" means — all files: Even the boot logos and the other various graphic images and glyphs are encrypted. Further, the keys to the kingdom are on the device itself — i-Devices contain on-board AES encryption modules, which are meant to discourage key recovery attempts.

The best laid schemes of mice and (Apple)-men, however, gang aft agley. As such, a certain publicly-available iPhone Wiki site contains a page with all the encryption keys readily available, at least for the pre-A5 devices (as they were obtained using the bootrom exploit). Likewise, many open source tools, most notably `xpwntool`[7] can be downloaded to decrypt the files, and `vfdecrypt`[8] for the file system images. A simple Internet search would quickly yield both the utilities and the keys. Once decrypted, the DMGs can be mounted easily on an OS X system (or converted to ISOs and mounted on Windows). The binaries can then be statically analyzed by the Mach-O tools (which we explored in Chapter 4), with certain caveats — most notably, attention to little-endian (Intel) vs. big-endian (ARM) format. As an alternative to jailbreaking iOS, downloading an `.ipsw` and decrypting its files is a close second for reverse engineering and investigating this operating environment.

The IMG3 format itself is pretty simple. It is comprised of a small header, followed by tagged fields. The tags are any of the following, shownin Table 6-20:

TABLE 6-20: Known IMG3 tags

TAG	DENOTES
TYPE	The type of the file
DATA	The actual payload of the file
KBAG	"Keybag": The key and IV for the file, to be used with the device's built-in (GID) key. Encrypted with AES256, usually
CHIP	The CPU identifier this file is for
ECID	Exclusive Chip ID (CPU unique identifier)
MODS	Security Domain
PROD	Production Mode
VERS	Version of the data file format
SEPO	Security Epoch
SHSH	The secure hash — The SHA-1 encrypted with Apple's RSA private key
CERT	Certificate — Apple's certificate, trusted by the device's hard coded certificate

The example shown here is the iOS 5 kernel cache of an iPod. The fields are, naturally, ARM-endian. Fields in bold are constant.

```
morpheus@Ergo (...)$ od -t x1 kernelcache.release.n81 |more

0000000   33 67 6d 49 | c4 e3 5d 00 | b0 e3 5d 00        | 78 db 5d 00
           3  G  M  I |(File Size)  |(Size,no header)    |(size of data)

0000020   6c 6e 72 6b | 45 50 59 54 | 20 00 00 00        | 04 00 00 00
           l  n  r  k |  E  P  Y  T |    length          |  tag data len

0000040   6c 6e 72 6b | 00 00 00 00   00 00 00 00   00 00 00 00
           l  n  r  k |       ( padding to length )

0000048   00 00 00 00 | 41 54 41 44 | 70 da 5d 00        | 64 da 5d 00
                      |  A  T  A  D |(data+data hdr)     |(actual data)
```

The header size is usually 64-bytes, though its exact size can always be determined by following the fields. The actual file data is tagged by DATA.

The book's companion website contains a tool, *imagine*, which can be used to dump the contents of an IMG3 file. It contains built-in parsers for the file format, and can also parse custom data formats like the device tree. Executing it will produce results similar to Output 6-6:

OUTPUT 6-6: Running the imagine tool on iBoot

```
morpheus@ergo (iOS/Tools)$ ./imagine iBoot.k48ap.RELEASE.img3
Ident: ibot
Tag: TYPE (54595045) Length 0x20
        Type: ibot (iBoot)
Tag: DATA (44415441) Length 0x2d00c
        Data length is 184320 bytes
Tag: VERS (56455253) Length 0x2c
        Version: iBoot-1219.62.8
Tag: SEPO (5345504f) Length 0x1c
        Security Epoch: 02 00 00 00
Tag: BORD (424f5244) Length 0x1c
        Board: 02 00 00 00
Tag: SEPO (5345504f) Length 0x1c
        Security Epoch: 02 00 00 00
Tag: CHIP (43484950) Length 0x1c
        Chip: 30 89 00 00
Tag: BORD (424f5244) Length 0x1c
        Board: 02 00 00 00
Tag: KBAG (4b424147) Length 0x4c
        Keybag: AES 256
Tag: KBAG (4b424147) Length 0x88
        Keybag: AES 256
Tag: SHSH (53485348) Length 0x8c
Tag: CERT (43455254) Length 0x7ac
```

The following experiment will walk you through the stages of unpacking and decrypting an IMG3 file.

Experiment: Decrypting the iOS 5 Kernel Cache

This exercise demonstrates decrypting an IMG3 file using two publicly available tools — `xpwn`, and `lzssdec`. The file in question is the iOS 5 kernel cache, but this can be tried on any file. The point of departure is the iOS 5 `ipsw` for iPod touch, but you can try this on any `.ipsw`, provided you can get your hands on the (also publicly available) decryption keys.

When decrypted, the IMG3 files stay in the same format, albeit with a decrypted payload. The `kernelcache` is particularly important, and is in a compressed payload, with a very simple Lempel-Ziv (UNIX `compress(1)`-like) format. The `lzssdec` (or similar utility) can be used to decompress the file. So, assuming you found the key in some iPhone Wiki site or elsewhere, the steps shown in Listing 6-6a would end up with the actual kernel cache:

LISTING 6-6A: Decompressing the iOS 5 `kernelcache` with xpwntool. Given the right IV and KEY, you can use this for any iOS image and any file therein.

```
morpheus@Ergo (...)$ export IV=...  # Set the IV, if we hypothetically knew it
morpheus@Ergo (...)$ export KEY=... # Set key, if hypothetically we knew, too..

# Run xpwntool, specifying the in file
# (in this case, kernelcache.release.n81) to be decrypted
morpheus@Ergo (...)$ xpwntool kernelcache.release.n81 kernelcache.decrypted -iv
$IV  -k $KEY -decrypt

# The resulting file is still an Img3—but, if you squint hard, makes sense
morpheus@Ergo (...)$ more kernelcache.decrypted
3gmI... ... ... ... ... ... .....lnrkEPYT...lnrk.....complzss... ... ... ....
... ... ...<CE><FA><ED><FE>.............................................
... ... ... ... ..._TEXT... ... ... ... ... ... ...cstring... ... ... ... ...
```

Because the `kernelcache` is compressed — and even uncompressed, would still be binary — it takes some sifting to pick out the meaningful Mach-o header and some section/segment names. Using `od(1)` makes life somewhat easier, and certainly spares you the effort of parsing the IMG3 header (Listing 6-6b):

LISTING 6-6B (CONTINUED): Using od(1) to find the beginning of the actual data

```
morpheus@Ergo (...)$ od -A d -t x1 kernelcache.decrypted |more
0000000    33  67  6d  49  f8  e2  5d  00  e4  e2  5d  00  ac  da  5d  00
0000016    6c  6e  72  6b  45  50  59  54  20  00  00  00  04  00  00  00
0000032    6c  6e  72  6b  00  00  00  00  00  00  00  00  00  00  00  00
0000048    00  00  00  00  41  54  41  44  70  da  5d  00  64  da  5d  00
                     ---------- End of IMG3 Header ----------
                ---------- Beginning of complzss Header ----------
0000064    63  6f  6d  70  6c  7a  73  73  b9  05  fc  53  00  a7  00  00
0000080    00  5d  d8  e4  00  00  00  00  00  00  00  00  00  00  00  00
0000096    00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
*
                   ---------- CompLZSS data begins ----------
0000448    ff  ce  fa  ed  fe  0c  00  00  00  d5  09  f3  f0  02  f3  f0
0000464    0b  f3  f0  1c  08  a7  00  00  01  f3  f0  06  01  14  fa  f0
0000480    5f  9f  5f  54  45  58  54  f3  f0  18  05  10  9f  00  80  00
```

The IMG3 payload starts at offset 64, and is a compressed file (as indicated by the "complzss" signature). The Adler-32 compression actually leaves the first couple of bytes uncompressed, and you can see the Mach-O 32-bit header (`0xFEEDFACE`), at offset 448. One last step remains: to decompress the file. If this works, you end up with a perfectly plaintext ARM Mach-O file — the iOS kernel cache (Listing 6-6c):

LISTING 6-6C (ENDED): Arriving at the goal — the kernel Cache has been decompressed and decrypted.

```
morpheus@Ergo (...)$ lzssdec -o 448 < kernelcache.decrypted  > mach_kernelcache.arm
# If we have this right, the resulting file should start with 0xFEEDFACE
morpheus@Ergo (...)$ file mach_kernelcache.arm
mach_kernelcache.arm: Mach-O executable arm         # Success!
```

You are encouraged to try this on other files, as well. Files such as the DeviceTree, iBEC, iBSS, and iBoot are not compressed, and their data starts right at offset `0x40`.

The iOS Device Tree

Similar to EFI and OS X on Intel, iBoot and iOS on ARM use a device tree. The device tree is part of the firmware files, and you can get it by decrypting the `DeviceTree.<model>.img3` file from the `ipsw`.

The format is obviously undocumented, but — given that the kernel needs to parse it — it isn't far off from the device tree format prepared by EFI. The `ioreg` command on a jailbroken device will display the tree, as will the imagine tool, if applied to a decrypted tree. This is shown in Listing 6-7:

LISTING 6-7: The device tree from the author's iPod, as shown by the imagine tool

```
morpheus@Ergo (/tmp)$ imagine -d iOS/DeviceTree.n81ap.img3
Device Tree has 15 properties and 13 children
Properties:
device-tree
|  +--compatible Length 23
|  +--secure-root-prefix Length 3
|  +--AAPL,phandle Length 4
|  +--config-number Length 32
|  +--model-number Length 32
|  +--platform-name Length 32
|  +--serial-number Length 32
|  +--device_type Length 8
|  +--#size-cells Length 4
|  +--clock-frequency Length 4
|  +--mlb-serial-number Length 32
|  +--#address-cells Length 4
|  +--region-info Length 32
|  +--model Length 8
|  +--name Length 12
   +--chosen
|  |  +--firmware-version Length 256
```

```
|  |  +--display-scale Length 4
|  |  +--system-trusted Length 4
|  |  +--AAPL,phandle Length 4
|  |  +--production-cert Length 4
...  (output truncated for brevity)
```

SUMMARY

This chapter presented, in depth, the EFI stage of booting OS X — the precursor to booting the kernel. EFI is the successor to the PowerPC's OpenFirmware architecture, and follows similar concepts, albeit a different implementation.

Similar to EFI, but much less documented, is Apple's iOS boot-loader, iBoot, on the various i-Devices. The chapter discussed, as much as is possible, the stages of iOS boot: from the Bootrom, through the Low Level Bootloader (LLB), the main bootloader (iBoot), and the DFU mode loaders (iBEC and iBSS).

Additionally, OS X and iOS installation images were described in great detail. OS X uses packages, and iOS uses an `.ipsw` archive, containing all the components of the operating system.

The chapter deliberately left out what happens next — booting the kernel. The kernel boot process is complicated and lengthy — and well deserves a dedicated chapter. Likewise, what follows the kernel — user mode startup — is long enough for a chapter of its own. You are encouraged to choose your own adventure:

- Fall through to the next chapter (default) — describing the user mode startup.
- Skip to Chapter 8, describing the kernel's life, and often premature demise (i.e. panics).

REFERENCES AND FURTHER READING

1. Intel's EFI 1.10 specification — `www.intel.com/content/www/us/en/architecture-and-technology/unified-extensible-firmware-interface/efi-homepage-general-technology.html`
2. The UEFI standard — `www.uefi.org/specs/`
3. Apple Support — Startup key combinations for Intel-based Macs (HT1533): `http://support.apple.com/kb/ht1533`
4. rEFIt — `http://refit.sourceforge.net`
5. Esser, Stephen (i0nic). "*Targeting the iOS Kernel*" — a presentation for Syscan 2011, Singapore: `www.syscan.org`
6. Zdziarski, Jonathan. *Hacking and Securing iOS Applications: Stealing Data, Hijacking Software, and How to Prevent It* (New York: O'Reilly, 2012)
7. The xpwn tool — downloadable from `http://theiphonewiki.com/`
8. VFDecrypt — downloadable from `http://theiphonewiki.com/`

7

The Alpha and the Omega — launchd

When you power on your Mac or i-Device, the boot loader (OS X: EFI, iOS: iBoot), described in the previous chapter is responsible for finding the kernel and starting it up. The kernel boot is described in detail in Chapter 7. The kernel, however, is merely a service provider, not an actual application. The user mode applications are those which perform the actual work in a system, by building on kernel primitives to provide the familiar user environment rich with files, multimedia, and user interaction. It all has to start somewhere, and in OS X and iOS — it starts with launchd.

LAUNCHD

launchd is OS X's and iOS's idea of what other UN*X systems call *init*. The name may be different, but the general idea is the same: It is the first process started in user mode, which is responsible for starting — directly or indirectly — every other process in the system. In addition, it has OS X and iOS idiosyncratic features. Even though it proprietary, it still falls under the classification of Darwin, and so it is fully open source[1].

Starting launchd

launchd is started directly by the kernel. The main kernel thread, which is responsible for loading the BSD subsystem, spins off a thread to execute the `bsdinit_task`. The thread assumes PID 1, with the temporary name of "init," a legacy of its BSD origins. It then invokes `load_init_program()`, which calls the `execve()` system call (albeit from kernel space) to execute the daemon. The name — `/sbin/launchd` — is hard coded as the variable `init_program_name`.

The daemon is designed to be started in this way, and this way only; It cannot be started by the user. If you try to do so, it will complain, as shown in Listing 7-1.

LISTING 7-1: Attempting to start launchd will result in failure

```
root@Minion (/)# /sbin/launchd
launchd: This program is not meant to be run directly.
```

Although launchd cannot be started, it can be tightly controlled. The `launchctl(1)` command may be used to interface with launchd, and direct it to start or stop various daemons. The command is interactive, and has its own help.

launchd is usually started with no arguments, but does optionally accept a single command line argument: `-s`. This argument is propagated to it by the kernel, if the latter was started with `-s`, either through its `boot-args`, or by pressing Option-S during startup.

launchd can be started with several logging and debugging features, by creating special dot files in `[/private]/var/db`. The files include `.launchd_log_debug`, `.launchd_log_shutdown` (output to `/var/tmp/launchd-shutdown.log`), and `.launchd_use_gmalloc` (enabling `libGMalloc`, as discussed in Chapter 3). launchd also checks for the presence of the `/AppleInternal` file (on the system root) for some Apple internal logging.

launchd's loading of `libGMalloc` on iOS (if `/var/db/.launchd_use` has been used by the jailbreaker comex in what is now known as the interposition exploit. launchd executes with root privileges, and by crafting a Trojan library, code can be injected into userland root — one step closer to subverting the kernel.

System-Wide Versus Per-User launchd

If you use `ps(1)` or a similar command on OS X, you will see more than one instance of launchd: The first is PID 1, which was started by the kernel in the manner described previously. If anyone is logged on, there will be another launchd, forked from the first, and owned by the logged in user, shown in Listing 7-2. You may also see other instances, belonging to system users (e.g. spotlight - uid 89).

LISTING 7-2: Two instances of launchd

```
morpheus@ergo (/)$  ps -ef | grep sbin/launchd
    0     1     0   0   6:32.43 ??         6:37.98 /sbin/launchd
  501    95     1   0   0:06.44 ??         0:11.07 /sbin/launchd
```

The per-user launchd is executed whenever a user logs in, even remotely over SSH (though once per logged in user). On iOS there is only one instance of launchd, the system-wide instance.

It is impossible to stop the system-wide launchd (PID 1). In fact, launchd is the only immortal process in the system. It cannot be killed, and that makes sense. There is absolutely no reason to terminate it. In most UN*X, if the init process dies unexpectedly the result is a kernel panic. launchd is also the last process to exit, when the system is shut down.

Daemons and Agents

The core responsibility of launchd is, as its name implies, launching other processes, or jobs, on a scheduled or on-demand basis. launchd makes a distinction between two types of background jobs:

- Daemons are, like the traditional UNIX concept, background services that normally have no interaction with the user. They are started automatically by the system, whether or not any users are logged on.
- Agents are special cases of daemons that are started only when a user logs on. Unlike daemons, they may interface with the user, and may in fact have a GUI.
- iOS does not support the notion of a user login, which is why it only has LaunchDaemons (though an empty `/Library/LaunchAgents` does exist).
- Both daemons and agents are declared in their individual property list (`.plist`) files. As described in Chapter 2, these are commonly XML (in OS X) or binary (in iOS). A detailed discussion of the valid plist entries in the verbose man page — `launchd.plist(5)`, though it should be noted the man page does leave out a few undocumented keys. The rest of this chapter demonstrates the plist format through various examples. The complete list of job keys (including useful keys for sandboxing jobs) can be found in launchd's `launch_priv.h file.`

The list of daemons and agents can be found in the locations noted in Table 7-1.

TABLE 7-1: Launch Daemon locations

DIRECTORY	USED FOR
`/System/Library/LaunchDaemons`	Daemon plist files, primarily those belonging to the system itself.
`/Library/LaunchDaemons`	Daemon plist files, primarily third party.
`/System/Library/LaunchAgents`	Agent plist files, primarily those belonging to the system itself.
`/Library/LaunchAgents`	Other agent plist files, primarily third party. Usually empty.
`~/Library/LaunchAgents`	User-specific launch agents, executed for this user only.

launchd uses the `/private/var/db` directory for its runtime configuration, creating `com.apple.launchd[.peruser.%d]` files for runtime override and disablement of daemons.

The Many Faces of launchd

launchd is the first process to emerge to user mode. When the system is at its nascent stage, it is (briefly) the only process. This means that virtually every aspect of system startup and function is either directly or indirectly dependent on it. In OS X and iOS, launchd serves multiple roles, which in other UN*X are traditionally delegated to several daemons.

init

The first, and chief role played by launchd is that of the daemon init. The job description of the latter involves setting up the system by spawning its myriad daemons, then fading to the background, and ensuring these daemons are alive. If one dies, launchd can simply respawn it.

Unlike traditional init, however, the launchd implementation is somewhat different, and considerably improved, as shown in Table 7-2:

TABLE 7-2: init vs. launchd

RESPONSIBILITY	TRADITIONAL INIT	LAUNCHD
Function as PID 1, great ancestor of all processes	init is the first process to emerge into user mode, and forks other processes (which in turn may fork others). Resource limits it sets for itself are inherited by all of its descendants.	Same. launchd also sets Mach exception ports, which are used by the kernel internally to handle exception conditions and generate signals (see Chapter 8).
Support "run levels"	Traditional init supports run levels: 0 – poweroff 1 – single user 2 – multi-user 3 – multi-user + NFS 5 – halt 6 – reboot	launchd does not recognize run levels and allows only for individual per-daemon or per-agent files. There is, however, a distinction for single-user mode.
Start system services	init runs services in order, per files listed in `/etc/rc?.d` (corresponding to run level), in lexicographic order.	launchd runs both system services (daemons), and per-user services (agents).
System service specification	init runs services as shell scripts, unaware and oblivious to their contents.	launchd processes property list files, with specific keywords.
Restart services on exit	init recognizes the `respawn` keyword in `/etc/inittab` for restart.	launchd allows a KeepAlive key in the daemon or agent's property list.
Default user	Root.	Root, but launchd allows a username key in the property list.

Per-User Initialization

Traditional UN*X has no mechanism to run applications on user login. Users must resort to shell and profile scripts, but those quickly get confusing since each shell uses different files, and not all shells are necessarily login shells. Additionally, in a GUI environment it is not a given that a shell would be started, at all (as is indeed the case with most OS X users, who remain unaware of the `Terminal.app`).

By using LaunchAgents, launchd enables per-user launching of specific applications. Agents can request to be loaded by default in all sessions, or only in GUI sessions, by specifying the `LimitLoadToSessionType` key with values such as `LoginWindow` or `Aqua`, or `Background`.

atd/crond

UN*X traditionally defines two daemons — `atd` and `crond` — to run scheduled jobs, as in executing a specified command at a given time. The first daemon, `atd`, serves as the engine allowing the `at(1)` command for one-time jobs, whereas the second, `crond`, provides recurring job support.

Apple is gradually phasing out `atd` and `crond`. The `atd` is no longer a stand-alone daemon, but is now started by launchd. This service, defined in `com.apple.atrun.plist`, (shown in Listing 7-3) is usually disabled:

LISTING 7-3: The com.apple.atrun.plist

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>Label</key>
    <string>com.apple.atrun</string>
    <key>ProgramArguments</key>
    <array>
            <string>/usr/libexec/atrun</string>
    </array>

    <key>StartInterval</key>
    <integer>30</integer>

    <key>Disabled</key>
    <true/>
</dict>
</plist>
```

```
launchd starts atrun(8) every 30
seconds, if enabled
```

```
Disabled by default. Setting Disabled:false
(or removing key) enables
```

The `atrun` plist must be enabled to allow the `at(1)` family of commands to work. Otherwise, it will schedule jobs, but they will never happen (as the author learned the hard way, once relying on it to set a wake-up alarm).

The `crond` service is still supported (in `com.vix.crond.plist`), although launchd has its own set of `StartCalendarInterval` keys to replace it. Apple supplies `periodic(8)` as a replacement. Listing 7-4 shows `com.apple.periodic-daily`, one of the several cron-substitutes (along with `-weekly` and `-monthly`):

LISTING 7-4: com.apple.periodic-daily.plist

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
  "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>Label</key>
        <string>com.apple.periodic-daily</string>
        <key>ProgramArguments</key>
        <array>
                <string>/usr/sbin/periodic</string>
                <string>daily</string>
        </array>
        <key>LowPriorityIO</key>
        <true/>
        <key>Nice</key>
        <integer>1</integer>
        <key>StartCalendarInterval</key>
        <dict>
                <key>Hour</key>
                <integer>3</integer>
                <key>Minute</key>
                <integer>15</integer>
        </dict>
        <key>AbandonProcessGroup</key>
        <true/>
</dict>
</plist>
```

In iOS, an alternate method of specifying periodic execution is with the `StartInterval` key. The `/usr/sbin/daily` service, for example, specifies a value of 86,400 seconds (24 hours). Other services, such as `itunesstored` and `softwareupdateservicesd` also use this method.

inetd/xinetd:

In UN*X, inetd (and its successor, xinetd) is used to start network servers. The daemon is responsible for binding the port (UDP or TCP), and — when a connection request arrives — it starts the server on demand, and connects its input/output descriptors (`stdin`, `stderr`, and `stdout`) to the socket.

This approach is highly beneficial to both the network server, and the system. The system does not need to keep the server running if there are no active requests to be serviced, thereby reducing system load. The server, on its part, remains totally agnostic of the socket handling logic, and can be coded to use only the standard descriptors. In this way, an administrator can whimsically reassign port numbers to services, and essentially run any CLI command, even a shell, over a network port.

launchd integrates the inetd functionality into itself*, by allowing daemons and agents to request a particular socket. All the daemon has to do is ask, using a `Sockets` key in its plist. Listing 7-5 shows an example of requesting TCP/IP socket 22, from `ssh.plist`:

* Technically, the inetd functionality is handled by launchproxy(8), also part of the launchd project. The manual page has been promising the two would be merged eventually, but it has yet to happen.

LISTING 7-5: ssh.plist, demonstrating IP socket registration

```
<plist version="1.0">
<dict>
     <key>Disabled</key>
     <true/>

     <key>Label</key>
     <string>com.openssh.sshd</string>

     <key>Program</key>
     <string>/usr/libexec/sshd-keygen-wrapper</string>
     <key>ProgramArguments</key>
     <array>
         <string>/usr/sbin/sshd</string>
         <string>-i</string>
     </array>


     <key>Sockets</key>
     <dict>
        <key>Listeners</key>
        <dict>
           <key>SockServiceName</key>
           <string>ssh</string>

           <key>Bonjour</key>
           <array>
             <string>ssh</string>
             <string>sftp-ssh</string>
           </array>
        </dict>
     </dict>
     <key>inetdCompatibility</key>
     <dict>
          <key>Wait</key>
          <false/>
     </dict>

     <key>StandardErrorPath</key>
     <string>/dev/null</string>

     <key>SHAuthorizationRight</key>
     <string>system.preferences</string>
</dict>
</plist>
```

Disabled by default. Setting Disabled:false (or removing key) enables

"Label" defines the service internally (for launchctl(8))

"Program" specifies path to execute. Command line arguments are specified in an array

SockServiceName refers to /etc/services: ssh 22/tcp # SSH Remote Login Protocol

Bonjour advertises the service(s) over multicast

inetdCompatibility allows porting from the legacy inetd.conf (here, "nowait", allowing multiple instances)

StandardErrorPath redirects stderr to /dev/null.

Unlike inetd, the socket the daemon is requesting may also be a UNIX domain socket. Listing 7-6, an excerpt from `com.apple.syslogd.plist`, demonstrates this:

LISTING 7-6: com.apple.syslogd.plist, demonstrating UNIX socket registration

```
...
<key>ProgramArguments</key>
        <array>
                <string>/usr/sbin/syslogd</string>
        </array>
        <key>Sockets</key>
        <dict>
                <key>AppleSystemLogger</key>
                <dict>
                        <key>SockPathMode</key>
                        <integer>438</integer>
                        <key>SockPathName</key>
                        <string>/var/run/asl_input</string>
                </dict>
                <key>BSDSystemLogger</key>
                <dict>
                        <key>SockPathMode</key>
                        <integer>438</integer>
                        <key>SockPathName</key>
                        <string>/var/run/syslog</string>
                        <key>SockType</key>
                        <string>dgram</string>
                </dict>
        </dict>
```

The two socket families — UNIX and INET — are not mutually exclusive, and may be specified in the same clause. The previous `syslogd` plist, for example, can easily be modified to allow syslog to accept messages from UDP 514 by adding a `SockServiceName:syslog` key (and optionally appending `-udp_in` and `1` to the `ProgramArguments` array). The iOS daemon `lockdownd` listens in this way on TCP port 62078 and the UNIX socket `/var/run/lockdown.sock`.

mach_init

True to its NEXTStep origins and before the advent of launchd in OS X 10.4, the system startup process was called `mach_init`. This daemon was actually responsible for later spawning the BSD style init, which was a separate process. The two were fused into launchd, and it has assumed `mach_init`'s little documented, but chief role of the bootstrap service manager.

Mach's IPC services rely on the notion of "ports" (vaguely akin to TCP and UDPs), which serve as communication endpoints. This is described (in great detail) in Chapter 10. For the moment, however, it is sufficient to consider a port as an opaque number that can also be referenced by a fully qualified name. Servers and clients alike can allocate ports, but servers either require some type of locator service to allow clients to find them, or otherwise need to be "well-known."

Enter: the bootstrap server. This server is accessible to all processes on the system, which may communicate with it over a given port — the `bootstrap_port`. The clients can then request, over this port, that the server lookup a given service by its name and match them with its port. (UNIX has a similar function in its RPC portmapper, also known as `sunrpc`. The mapper listens on a well-known port (TCP/UDP 111) and plays matchmaker for other RPC services)[1].

Prior to launchd, `mach_init` assumed the role of `bootstrap_server`. launchd has since taken over this role and claims the port (aptly named `bootstrap_port`) during its startup. Since all processes in the system are its progeny, they automatically inherit access to the port. `bootstrap_port` is declared as an `extern mach_port_t` in `<servers/bootstrap.h>`.

Servers wishing to register their ports with the bootstrap server can use the port to do so, using functions defined in `<servers/bootstrap.h>`. These functions (`bootstrap_create_server` and `bootstrap_create_service`) are still supported, but long deprecated. Instead, the service can be registered with launchd in the server's plist, and a simpler function — `bootstrap_check_in()` — remains to allow the server to request launchd to hand over the port when it is ready to service requests:

```
kern_return_t bootstrap_check_in(mach_port_t bp,            // bootstrap_port
                                 const name_t service_name, // name of service
                                 mach_port_t *sp);          // out: server port
```

launchd pre-registers the port when processing the server's plist. The server port is usually ephemeral, but can also be well known if the key `HostSpecialPort` is added. (This is discussed in more detail in Chapter 10, under "Host Special Ports"). launchd can be instructed to wait for the server's request, as is shown in Listing 7-7. `com.apple.windowserver.active` will be advertised to clients only after WindowServer checks in with launchd using functions from `<launch.h>`.

LISTING 7-7: com.apple.WindowServer.plist

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
  "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>Label</key>
        <string>com.apple.WindowServer</string>
        <key>ProgramArguments</key>
        <array>

           <string>/System/Library/Frameworks/ApplicationServices.framework/Frameworks/
            CoreGraphics.framework/Resources/WindowServer</string>
           <string>-daemon</string>
        </array>
        <key>MachServices</key>
        <dict>
                <key>com.apple.windowserver</key>
                <true/>
                <key>com.apple.windowserver.active</key>
                <dict>
```

continues

[1]Readers familiar with Android will note the similarity to its Binder mechanism, which (among other IPC related tasks) also allows system services to be published, albeit using a character device, /dev/binder, rather than a port.

LISTING 7-7 *(continued)*

```
                        <key>HideUntilCheckIn</key>
                        <true/>
                </dict>
        </dict>
</dict>
</plist>
```

Any clients wishing to connect to a given service, can then look up the server port using a similar function:

```
kern_return_t bootstrap_look_up(
                mach_port_t bp,              // always bootstrap_port
                const name_t service_name,   // name of service
                mach_port_t *sp);            // out: server port
```

If the server's port is available and the server has checked in, it will be returned to the client, which may then send and receive messages (using `mach_msg()`, also discussed in Chapter 10). The Mach messages for the bootstrap protocol are defined in the launchd source in `.defs` files, which are preprocessed by the Mach Interface Generator (MIG) (also discussed in Chapter 10). You can view a list of the active daemons using the `bslist` subcommand of `launchctl(1)`. The list prints out a flattened view of the hierarchical namespace of bootstrap servers visible in the current context. The `bstree` subcommand displays the full hierarchical namespace (but requires root privileges). In Lion and later, `bstree` also shows XPC namespaces (discussed later in this chapter).

The bootstrap mechanism is now implemented over launchd's `vproc`, a new library introduced in Snow Leopard, which also provides for the next feature, transactions.

Transaction Support

launchd is smarter than the average init. Unlike init, which can just start or stop its daemons, launchd supports transactions, a useful feature exported by launchd's `vproc`, which daemons can access through the public `<vproc.h>`. Daemons using this API can mark pending transactions by encapsulating them between `vproc_transaction_begin`, which generates a transaction handle, and `vproc_transaction_end` on that handle, when the transaction completes. A transaction-enabled daemon can also indicate the `EnableTransactions` key in its plist, which enables launchd to check for any pending transactions when the system shuts down, the user logs out, or after a specified timeout. If there are no outstanding transactions (the process is *clean*), the daemon will be shot down (with a `kill -9`) instead of gracefully terminated (`kill -15`), speeding up the shutdown or logout process, or freeing system resources after sufficient inactivity.

Resource Limits and Throttling

launchd can enforce self-imposed resource limits on its jobs. A job (daemon or agent) can specify `HardResourceLimits` or `SoftResourceLimits` dictionaries, which will cause launchd to call `setrlimit(2)`. The `Nice` key can be used to set the job's `nice` value, as per `nice(1)`. Additionally, a job can be marked with the `LowPriorityIO` key which causes launchd to call `iopolicysys` (system call #322, discussed in Chapter 14) and lower the job's I/O priority. Lastly, launchd is integrated with iOS's Jetsam mechanism (also known as memorystatus, and discussed in Chapter 14), which can enforce virtual memory utilization limitations, a feature that is especially important in iOS, which has no swap space.

Autorun Emulation and File System Watch

One of Windows' most known (and often annoying) features is autorun, which can automatically start a program when removable media (such as a CD, USB storage, or hard disk) is attached. launchd offers the `StartOnMount` key, which can trigger a daemon to start up any time a file system is mounted. This can not only emulate the Windows functionality, but is actually safer, as the autorun feature in Windows has become a vector for malware propagation. launchd's daemon are run from the permanent file system, rather than the removable one.

launchd can also be made to watch a particular path, not necessarily a mount point, for changes, using the `WatchPaths` or the `QueueDirectories` keys. This is very useful, as it can react in real time to file system changes. This functionality is achieved by listening on kernel events (`kqueues`), as discussed in Chapter 3. Daemons may be further extended to support `FSEvents` as well (described in Chapter 4), by specifying a `LaunchEvents` dictionary with a `com.apple.fsevents.matching` dict of matching cases.

I/O Kit Integration

A new feature in Lion is the integration of launchd with I/O Kit. I/O Kit is the runtime environment of device drivers. Launch daemons or agents can request to be invoked on device arrival by specifying a `LaunchEvents` dictionary containing a `com.apple.iokit.matching` dictionary. For the specifics of I/O Kit and its matching dictionaries, turn to Chapter 19. A high-level example, however, can be seen in Listing 7-8, which shows an excerpt from the `com.apple.blued.plist` launch daemon, which is triggered by the to handle Bluetooth SDP transactions.

LISTING 7-8: **com.apple.blued.plist, demonstrating I/O Kit triggers**

```
<plist version="1.0">
<dict>
        <key>EnableTransactions</key>
        <true/>
        <key>KeepAlive</key>
        <dict>
                <key>SuccessfulExit</key>
                <false/>
        </dict>
        <key>Label</key>
        <string>com.apple.blued</string>
        <key>MachServices</key>
        <dict>
                <key>com.apple.blued</key>
                <true/>
                <key>com.apple.BluetoothDOServer</key>
                <dict>
                        <key>ResetAtClose</key>
                        <true/>
                </dict>
```

continues

LISTING 7-8 *(continued)*

```
        </dict>
        <key>Program</key>
        <string>/usr/sbin/blued</string>
  <key>LaunchEvents</key>
        <dict>
                <key>com.apple.iokit.matching</key>
                <dict>
                        <key>com.apple.bluetooth.hostController</key>
                        <dict>
                                <key>IOProviderClass</key>
                                <string>IOBluetoothHCIController</string>
                                <key>IOMatchLaunchStream</key>
                                <true/>
                        </dict>
                </dict>
        </dict>
</dict>
</plist>
```

Experiment: Setting up a Custom Service

One of the niftiest features of UNIX inetd was its ability to run virtually any UNIX utility on any port. The combination of the inetd's handling of socket logic on the one hand, and the ability to treat a socket as any other file descriptor on the other, provides this powerful functionality.

This is also possible, if a little more complicated with launchd. First, we need to create a launchd plist for our program. Fortunately, this is a simple matter of copy, paste, and modify, as Listing 7-5 can do just fine if you change the `Label`, `Program`, `ProgramArguments`, and `Sockets` keys to whatever you wish.

But here, we encounter a problem: launchd does allow the running of any arbitrary program in response to a network connection, but supports only the redirection of `stdin`, `stdout`, and `stderr` to files. We want the application's `stdin`, `stdout`, and `stderr` to be connected to the socket that launchd will set up for us. This means the program we launch has to be launchd-aware and request the socket handoff.

To solve this, we need to create a generic wrapper, as is shown in Listing 7-9.

LISTING 7-9: A generic launchd wrapper

```
#include <stdio.h>
#include <sys/socket.h>
#include <launch.h> // LaunchD related stuff
#include <stdlib.h> // for exit, and the like
#include <unistd.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <netdb.h> // for getaddrinfo
#include <fcntl.h>
```

```
#define JOBKEY_LISTENERS "Listeners"
#define MAXSIZE 1024
#define CMD_MAX 80


int main (int argc, char **argv)
{
  launch_data_t checkinReq, checkinResp;
  launch_data_t  mySocketsDict;
  launch_data_t  myListeners;

  int fdNum;
  int fd;
  struct sockaddr sa;
  unsigned int    len = sizeof(struct sockaddr);
  int    fdSession ;


  /* First, we must check-in with launchD.  */
  checkinReq = launch_data_new_string(LAUNCH_KEY_CHECKIN);
  checkinResp = launch_msg(checkinReq);

  if (!checkinResp) {
        // Failed to checkin with launchd - this can only be because we are run outside
        // its context. Print a message and exit
        fprintf (stderr,"This command can only be run under launchd\n");
        exit(2);
    }


  mySocketsDict = launch_data_dict_lookup(checkinResp, LAUNCH_JOBKEY_SOCKETS);

  if (!mySocketsDict)
   { fprintf (stderr, "Can't find <Sockets> Key in plist\n"); exit(1); }

  myListeners = launch_data_dict_lookup(mySocketsDict, JOBKEY_LISTENERS);

  if (!myListeners)
   {fprintf (stderr, "Can't find <Listeners> Key inside <Sockets> in plist\n");
   exit(1);



   fdNum = launch_data_array_get_count(myListeners);
   if (fdNum != 1)  {
        fprintf (stderr,"Number of File Descriptors is %d - should be 1\n", fdNum);
        exit(1);
   }

   // Get file descriptor (socket) from launchd
   fd = launch_data_get_fd(launch_data_array_get_index(myListeners,0));

   fdSession = accept(fd, &sa, &len);

   launch_data_free(checkinResp); // be nice..
```

continues

LISTING 7-9 *(continued)*

```
    // Print to stderr (/var/log/system.log) before redirecting..

    fprintf (stderr, "Execing %s\n", argv[1]);

    dup2(fdSession,0);     // redirect stdin
    dup2(fdSession,1);     // redirect stdout
    dup2(fdSession,2);     // redirect stderr
    dup2(fdSession,255);   // Shells also like FD 255.

    // Quick and dirty example - assumes at least two arguments for the wrapper,
    // the first being the path to the program to execute, and the second (and later)
    // being the argument to the launchd program
    execl(argv[1], argv[1], argv[2], NULL);


    // If we're here, the execl failed.
    close(fdSession);

    return (42);
}
```

As the listing shows, the wrapper uses `launchd_` APIs (all clearly prefixed with `launch_` and defined in `<launch.h>`) to communicate with launchd and request the socket. This is done in several stages:

- **Checking in with launchd —** This is done by sending it a special message, using the `launch_msg()` function. Since checking in is a standard procedure, it's a simple matter to craft the message using `launch_data_new_string(LAUNCH_KEY_CHECKIN)` and then pass that message to launchd.
- **Get our plist parameters —** Once launchd has replied to the check-in request, we can use its APIs to get the various settings in the plist. Note that there are two ways to pass parameters to the launched daemons, either as command-line arguments (the `ProgramArguments` array), or via environment variables, which are passed in an `EnvironmentVariables` dictionary, and read by the daemon using the standard `getenv(3)` call.
- **Get the socket descriptor —** Getting any type of file descriptor is a little tricky, since it's not as straightforward to pass between processes as strings and other primitive data types are. Still, any complexity is well hidden by `launch_data_get_fd`.

Once we have the file descriptor (which is the socket that launchd opened for us), we call `accept()` on it, as any network server would. This will yield a connected socket with our client on the other end. All that's left to do is to use the `dup2()` system call to replace our `stdin`, `stdout`, and `stderr` with the accepted socket, and `exec()` the real program. Because `exec()` preserves file descriptors, the new program receives these descriptors in their already connected state, and its `read(2)` and `write(2)` will be redirected over the socket, just as if it would have called `recv(2)` and `send(2)`, respectively.

To test the wrapper, you will need to drop its plist in `/System/Library/LaunchDaemons` (or another `LaunchDaemons` directory) and use `launchctl(1)` to start it, as shown in Output 7-1. The wrapper in this example was labeled `com.technologeeks.wrapper`, and was placed in an eponymous plist. Note in the output, that `launchctl(1)` isn't the chatty type and no comment implies the commands were successful.

OUTPUT 7-1: Using launchctl(1) to start a LaunchDaemon

```
root@Minion (~)# launchctl
launchd% load /System/Library/LaunchDaemons/com.technologeeks.wrapper.plist
launchd% start com.technologeeks.wrapper
launchd% exit
```

Because the wrapper is intentionally generic, you can specify any program you want, assuming this program uses `stdin`, `stdout`, and `stderr` (which all command line utilities do, anyway). This enables nice backdoor functionality, as you can easily set up a root shell on any port you want. Setting the command line arguments to your wrapper to `/bin/zsh -i` will result in output similar to Output 7-2:

OUTPUT 7-2: Demonstrating a launchd-wrapped root shell

```
root@Minion (~)# telnet localhost 1024 # or whereever you set your SockServiceName
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
zsh# id;
uid=0(root) gid=0(wheel) groups=0(wheel),401(com.apple.access_screensharing),
402(com.apple.sharepoint.group.1),1(daemon),
2(kmem),3(sys),4(tty),5(operator),8(procview),9(procmod),12(everyone),
20(staff),29(certusers),
33(_appstore),61(localaccounts)80(admin),98(_lpadmin),100(_lpoperator),
204(_developer)
zsh: command not found: ^M
zsh# whoami;
root
zsh: command not found: ^M
```

Note that a semicolon must be appended to shell commands. This is because you are working directly over the shell's `stdin`, and not a terminal, so the enter key is sent out as a literal Ctrl-M. The semicolon added terminates the command so the shell can parse it, making the Ctrl-M into a separate, invalid command. A minor annoyance in exchange for remote root capabilities.

LISTS OF LAUNCHDAEMONS

There are an inordinate amount of LaunchDaemons in OS X and iOS. Indeed, many sites devote countless HTML pages and SMTP messages to debating the purpose and usefulness of the daemons and agents, especially in iOS, where unnecessary CPU cycles not only impact performance, but also dramatically shorten battery life. The following section aims to elucidate the purpose of these daemons and agents.

iOS and OS X share some common LaunchDaemons. All plists (and their Mach service entries) have the `com.apple` prefix, and usually run their binaries from `/usr/libexec`. They are shown in Table 7-3:

TABLE 7-3: Daemons common to iOS and OS X

LAUNCHDAEMON (/USR/LIBEXEC)	MACH SERVICES (COM.APPLE.*)	NOTES
`DumpPanic` `(CoreServices)`	`DumpPanic`	When kernel boots, collects any leftover panic data from a previous panic. Runs with `RunAtLoad=true`.
`appleprofilepolicyd`	`appleprofilepolicyd`	System profiling. Communicates with profiling kernel extensions. Registers `HostSpecialPort 16`.
`aslmanager`	`---`	Apple system Llog. Runs `/usr/bin/aslmanager`, and sets a WatchPath on `/var/log/asl/SweepStore`.
`Backupd` `(MobileBackup framework)`	`Backupd`	`RunAtLoad = true`.
`chud.chum`		Runs `/Developer/usr/libexec/chum`, the CHUD helper daemon allowing access to privileged kernel interfaces from user mode.
`configd`	`SCNetworkReachability` `Configd`	`KeepAlive = true`.
`AppleIDAuthAgent` `(CoreServices)`	`coreservices.appleid.authentication` `coreservices.appleid.passwordcheck`	Handles AppleID-related requests. Whereas iOS has both services, OS X version only has the second service, which runs with a `–checkpassword` switch.
`cvmsServer`	`cvmsServ`	Internal to OpenGL(ES) framework.
`fseventsd`	`FSEvents`	In OS X, `fseventsd` is run from the CarbonCore framework, which is internal to CoreServices.
`locationd`	`locationd.registration` `locationd.simulation (i)` `locationd.spi (i)` `locationd.synchronous (i)` `locationd.agent (SL)` `locationd.services(SL)`	`Location services.`

LAUNCHDAEMON (/USR/LIBEXEC)	MACH SERVICES (COM.APPLE.*)	NOTES
`mDNSResponder`	`mDNSResponder`	Multicast DNS listener. Core part of Apple's "Bonjour."
`mDNSResponderHelper`	`mDNSResponderHelper`	Provides privilege separation for `mDNSResponder`.
`notifyd` (`/usr/sbin`)	`system.notification_center`	System notification center: handles kernel and other notifications.
`racoon` (`/usr/sbin`)	`Racoon`	Open source VPNd. Thanks to this daemon iOS5 proved jail-breakable (twice).
`ReportCrash` (`/System/Library/CoreServices`)	`ReportCrash.*` (OS X has ReportCrash., iOS has JetSam, SafetyNet, SimulateCrash, and StackShot.)	The default crash handler, which intercepts all application crashes. Runs automatically on crash by setting job's Mach exception ports (discussed in Chapter 11).
`sandboxd`	`Sandboxd`	Also uses HostSpecialPort 14.
`securityd`	`Securityd` `SecurityServer (SL)`	Handles key access and authorization. Written by Perry the Cynic, apparently. `OnDemand`.
`syslogd`	`system.logger`	Passes messages to ASL via the `asl_input` socket (discussed in Chapter 4).

A list of OS X specific LaunchDaemons (and a host of LaunchAgents), is too large and tedious to fit in these pages, but is maintained on the book's companion website.

iOS launchdaemons

Table 7-4 details some of the daemons specific to iOS, in alphabetical order:

TABLE 7-4: Some of the iOS daemons in /System/Library/LaunchDaemons

LAUNCHDAEMON (/USR/LIBEXEC)	MACH SERVICES (COM.APPLE.*)	NOTES
`accessory_device_arbitrator`	`mobile.accessory_device_arbitrator`	Handles accessories plugged into i-Device, such as docks. Set to respond to events from I/O Kit on the `IOUSBInterface`, so it can be started whenever such an accessory is connected. Formerly `accessoryd`.
`Accountsd (Accounts.framework)`	`accountsd.accountmanager` `accountsd.oauthsigner`	Single sign-on. Runs as mobile.
`Amfid`	`MobileFileIntegrity`	Discouraging any attempt to run unsigned, un-entitled code in iOS. Arch-nemesis of all jailbreakers. Uses `HostSpecialPort` 18.
`Apsd (ApplePushService .framework)`	`Apsd`	Apple Push Service Daemon (the APS private framework). Runs as mobile.
`Assetsd (AssetsLibrary.framwork)`	`PersistentURLTranslator .Gatekeeper` `assetsd.*`	Runs as mobile.
`Atc`	`Atc`	Air traffic controller.
`Calaccessd (EventKit.framework/ Support)`	`Calaccessd`	The EventKit's calendar access daemon. Runs as mobile.
`crash_mover`	`crash_mover`	Moves crashes to `/var/Mobile/Library/Logs`.
`fairplayd.XXX`	`Fairplayd` `Unfreed`	User mode helper for Apple's "FairPlay" DRM. This daemon is hardware specific (the plist contains a `LimitedToHardware` key), with *`XXX`* specifying the board type (e.g., N81 for iPod 4,1).
`Itunesstored (iTunesStore.framework/ Support)`	`iTunesStore.daemon.*` `itunesstored.*`	The iTunes Store server. Mostly known for the app store badge notifications. Runs as mobile.
`Lockbot`	`---`	Listens on `/var/run/lockbot`. Assists in jailing the device.

LAUNCHDAEMON (/USR/LIBEXEC)	MACH SERVICES (COM.APPLE.*)	NOTES
`Lockdownd`	`lockdown.host_watcher`	See next section of this chapter.
`Mobileassetd`	`Mobileassetd`	Runs with `-t 15`.
`mobile.installd`	`mobile.installd`	Runs with `-t 30` as mobile.
`mobile.installd .mount_helper`	`mobile.installd .mount_helper`	Mounts the developer image when device is selected for development.
`mobile_obliterator`	`mobile.obliteration`	Remotely obliterate (that is, wipe) the device.
`Pasteboard (UIKit.framework/ Support/)`	`UIKit.pasteboardd`	Cut/paste support. Runs as mobile. Close relative of OS X's as `pboard(8)`, which is a LaunchAgent (q.v., `pbcopy(1)`, `pbpaste(1)`).
`SpringBoard (/System/Library/ CoreServices)`	`CARenderServer` `SBUserNotification` `UIKit.statusbarserver` `bulletinboard.*` `chatkit .clientcomposeserver.xpc` `iohideventsystem` `smsserver` `springboard.*`	The chief UI of i-Devices. Described in its own section in this chapter.
`Twitterd (Twitter.Framework)`	`twitter.authenticate` `twitterd.server`	Twitter support introduced in iOS 5.
`Vsassetsd (VoiceServices .framework/Support)`	`Vsassetd`	Responsible for voice assets. Runs as mobile.

Glancing over the table, you may have noticed two special Daemons in iOS: `SpringBoard` and `lockdownd`. `SpringBoard` is the GUI Shell and is described later in this Chapter. `lockdownd` deserves more detail, and is described next.

lockdownd

`lockdownd` is the arch-nemesis of jailbreakers everywhere, being the user mode cop charged with guarding the jail. It is started by launchd and handles activation, backup, crash reporting, device syncing, and other services. It registers the `com.apple.lockdown.host_watcher` Mach service, and listens on TCP port 62078, as well as the `/var/run/lockdown.sock` UNIX domain socket. It is also assisted by a rookie, `/usr/libexec/lockbot`.

Lockdownd is, in effect, a mini-launchd. It maintains its own list of services to start in /System/Library/Lockdown/Services.plist, as shown in Listing 7-10.

LISTING 7-10: An excerpt from lockdownd's services.plist

```
<plist version="1.0">
<dict>
        <key>com.apple.afc</key>
        <dict>
                <key>AllowUnactivatedService</key>
                <true/>
                <key>Label</key>
                <string>com.apple.afc</string>
                <key>ProgramArguments</key>
                <array>
                        <string>/usr/libexec/afcd</string>
                        <string>--lockdown</string>
                        <string>-d</string>
                        <string>/var/mobile/Media</string>
                        <string>-u</string>
                        <string>mobile</string>
                </array>
        </dict>
        <key>com.apple.afc2</key>
        <dict>
                <key>AllowUnactivatedService</key>
                <true/>
                <key>Label</key>
                <string>com.apple.afc2</string>
                <key>ProgramArguments</key>
                <array>
                        <string>/usr/libexec/afcd</string>
                        <string>--lockdown</string>
                        <string>-d</string>
                        <string>/</string>
                </array>
</dict>
```

The listing shows an important service — afc — which is responsible for transferring files between the iTunes host and the i-Device. This is required in many cases, for synchronization as well as moving crash and diagnostic data. The second instance of the same service (afc2) is automatically inserted in the jailbreak process, and differs only in its lack of the -u mobile command line argument to the afc, which makes it retain its root privileges instead of dropping to the non-privileged user mobile. lockdownd (just like launchd) runs as root and can drop privileges before running another process if the UserName key is specified.

GUI SHELLS

When the user logs in on the console (either automatically or by specifying credentials), the system starts a graphical shell environment. OS X uses the Finder, whereas iOS uses SpringBoard, but the two are often more similar than they let on. From launchd's perspective, both Finder and SpringBoard are just one or two more agents in the collection of over 100 daemons and agents they need to start and juggle. But for the user, these programs constitute the first (and often final) frontier for interaction with the operating system.

Finder (OS X)

Finder is OS X's equivalent of Windows' Explorer: It provides the graphical shell for the user. It is started as a launch agent upon successful login, from the `com.apple.Finder.plist` property list (in `/System/Library/LaunchAgents`)

Finder has dependencies on no less than 30 libraries and frameworks, some of them private, which you can easily display by using `otool(1) -l`. Doing so also reveals a peculiarity: Finder is a rare case of an encrypted binary. OS X supports code encryption, as described in Chapter 4 and detailed further in Chapter 13, but there are fairly few encrypted binaries. Output 4-3 demonstrated using `otool -l` to view the encrypted portion of Finder. Using `strings(1)` or trying to disassemble Finder is, therefore, a vain effort (unless the encryption is defeated, for example by a tool like corerupt, presented in Chapter 12). You can also use GDB to attach to Finder once it is running (yet again, defeating the whole purpose of the binary protection), and trace its threads (usually only three of them).

Finder is so tightly integrated with the system that the very design of the native file system, HFS+, has been built around it. The file and folder data, and indeed the volume data itself, contains special finder information fields. These fields enable many features, such as reopening folder windows in the exact dimensions and location the user placed them last. Finder additionally makes use of extended attributes to store information, such as color labels and aliases. These features are all discussed in Chapter 16 (which is entirely devoted to HFS+).

With a Little Help from My Friends

All the work of supporting the rich GUI can prove overwhelming for any one process, which is why the GUI handling is actually split between several processes, which are all in `/System/Library/CoreServices`.

The `Dock.app` is responsible for the familiar tray of icons usually found at the bottom of the desktop, as its name implies, but also sets the wallpaper (what X would call the "root window"), as can be witnessed when the process is killed. It is assisted by `com.apple.dock.extra`, which connects the UI actions to the Dock action outlets.

The `SystemUIServer.app` is responsible for the menu extras (right hand) side of the status bar, which it loads from `/System/Library/CoreServices/Menu Extras`. Note that there, menu extras may also be created programmatically (using `[NSStatusBar systemStatusBar]` and its `setImage/setMenu` methods), in which case these extras are the responsibility of the app which created them.

Due to their important role (and Apple's desire to keep their UI theirs for as long as possible before others "adopt" it), Finder's assistants (as well as other `CoreServices` apps) are also protected binaries.

Experiment: Figuring Out Who Owns What in the GUI

Using a shell (preferably over SSH) and the UNIX `kill(1)` command, you can quickly determine which process owns what part of the GUI. Your options are to either kill the process violently (using `kill -9`) or just pause the process (using `kill -STOP` and `kill -CONT`). Doing so on the various processes — Finder, Dock and SystemUIServer — will either briefly make their UI assets disappear (if killed, until the processes are automatically restarted by launchd) or hang with the spinning beachball of death (as long as the processes are stopped) or a "fast forward" effect (when the processes are resumed, and all the queued UI messages are delivered). Menu extras created by apps will be unaffected by SystemUIServer's suspension or premature demise.

You might want to use `killall(1)` instead of kill, as it will send a signal by name, rather than by PID. If you use it this way to kill the same process repeatedly, launchd throttles the processes, which after a few seconds are respawned.

SpringBoard (iOS)

What Finder is to OS X, SpringBoard is for iOS. In iOS the system need not logon, so SpringBoard is started automatically, to provide the familiar icon based UI of the system. This UI has served as the inspiration to Lion's LaunchPad, which uses the same GUI concepts and is essentially a back port of SpringBoard into OS X — a fact that is evident as some SpringBoard-named files can be found in LaunchPad binary (which is technically part of the dock). Much like its OS X GUI counterpart (Finder), SpringBoard is loaded from `/System/Library/CoreServices/`.

All by Myself (Sort of)

Unlike Finder, SpringBoard handles almost everything by itself, and there are only a few loadable bundles in the CoreServices directory. Finder's 30 dependencies are dwarfed by SpringBoard, which has about 80, as you can see with `otool -l`, which will also reveal that SpringBoard is (surprisingly) an unprotected binary.

SpringBoard nonetheless does turn to additional bundles for certain tasks. `/System/Library/SpringBoardPlugins` contains three types of loadable bundles (as of iOS 5):

- **`lockbundle`** — Lock bundles provide lock screen functionality. The `NowPlayingArtLockScreen.lockbundle` is responsible for providing the lock screen when the music player (Music~iphone or MobileMusicPlayer) is active and the screen is locked. The `PictureFramePlugin` shows pictures from the user's photo library. The iPhone also has a bundle for `VoiceMemosLockScreen` (to show voice messages and missed call indicators)
- **`servicebundle`** — Helps SpringBoard with various tasks, such as `ChatKit.servicebundle`, `IncomingCall.servicebundle`, and `WiFiPicker.servicebundle`.
- **`bundle`** — The original extension before iOS 5. Still exists for `NikeLockScreen.bundle` and `ZoomTouch.bundle`.

Creating the GUI

SpringBoard creates its GUI by enumerating the apps in `/Applications /var/mobile/Applications` and displaying icons for them on the i-Device. Icon enumeration is performed automatically when SpringBoard starts. Each app's `Info.plist` is read, and the app is displayed on one of the home screens with the icon specified in its `CFBundleIcons` property, unless it contains the `SBAppTags` key with a `hidden` array entry). Examples of hidden apps are Apple's own `DemoApp.app`, `iOS Diagnostics.app`, `Field Test.app`, `Setup.app`, and `TrustMe.app`.

> *iOS devices start* `Setup.app` *when first launched to configure the device, register, and activate it. This has been rumored to annoy certain types of people. A nice way to get past it is to jailbreak the device and boot it (tethered or untethered doesn't matter), then* `ssh` *into it and simply rename (*`mv`*)* `/Applications/Setup.app` *(the new name doesn't matter). Then, restart SpringBoard (*`killall SpringBoard`*), and that setup screen is gone. iTunes will still complain about device registration when syncing, but there are ways to bypass that, as well.*

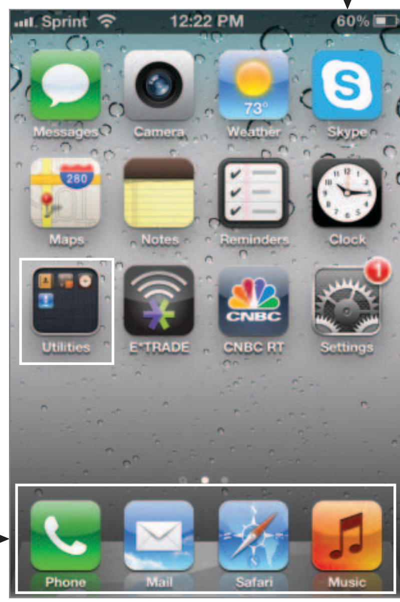
Icon grouping and the button bar settings are saved to `/var/mobile/Library/SpringBoard/IconState.plist`, with general home screen settings (as well as ringtones and other audio effects) in `/var/mobile/Library/Preferences/com.apple.springboard`. A third file, `applicationstate.plist`, controls application settings like badges. Figure 7-1 shows the mapping between the files and the home screen.

~/Library/Springboard/IconState.plist:

```
<plist version="1.0">
<dict>
  <key>buttonBar</key>
  <array>
   <string>com.apple.mobilephone</string>
   <string>com.apple.mobilemail</string>
   <string>com.apple.mobilesafari</string>
   <string>com.apple.mobileipod</string>
  </array>
<key>iconLists</key>
 <array>
  <array>
   <string>com.apple.MobileSMS</string>
    ...
   <string>com.apple.mobiletimer</string>
   <dict>
     <key>defaultDisplayName</key>
     <array>
      <string>com.apple.MobileAddressBook</string>
      <string>com.apple.calculator</string>
      <string>com.apple.compass</string>
      <string>com.apple.VoiceMemos</string>
     </array>
     <key>listType</key>
     <string>folder</string>
    </dict>
    <string>com.etrade.mobileproiphone</string>
    <string>com.nbcuni.cnbc.cnbcrt</string>
    <string>com.apple.Preferences</string>
   </array>
   <array>
      // Next home screen(s) follow ...
     ...
   </array>
</dict>
</plist>
```

~/Library/Preferences/com.apple.springboard:

```
<key>SBShowBatteryPercentage</key>
<true/>
```



~/Library/Springboard/applicationstate.plist:

```
<key>com.apple.Preferences</key>
<dict>
   <key>SBApplicationBadgeKey</key>
   <integer>1</integer>
   ......
```

FIGURE 7-1: SpringBoard's files and how they lay out the iOS home screen.

Experiment: Unhiding (or Hiding) an iOS App

It's a simple matter to hide or unhide apps on a jailbroken device. All it takes is editing the App's `Info.plist` and toggling the `SBAppTags` key. This is demonstrated in this simple experiment. You can use the method here to unhide or hide any app you wish.

For the app you choose, take the `Info.plist` and copy it to `/tmp`. Then, convert it to the more readable XML format (or, if you prefer, JSON) using `plutil(1)`. Edit the file to either add or remove the `SBAppTags` key with an array, containing a single string value of '`hidden`'. Finally, restart SpringBoard.

Performing the sequence of operations described here on DemoApp, we would have the sequence shown in Output 7-3:

OUTPUT 7-3: Toggling the visibility of an iOS app

```
root@padishah (/)# cp /Applications/DemoApp.app/Info.plist /tmp
root@padishah (/)# plutil -convert xml1 /tmp/Info.plist
Converted 1 files to XML format
root@padishah (/)# cat /tmp/Info.plist
…
     <key>SBAppTags</key>                        Add or remove this value
    <array>
        <string>hidden</string>
     </array>
…

root@padishah (/)# plutil -convert binary1 /tmp/Info.plist
Converted 1 files to binary format


root@padishah (/)# cp /tmp/Info.plist /Applications/DemoApp.app/
root@padishah (/)# killall SpringBoard
```

Handling the UI

Finder and SpringBoard are both in charge of presenting the UI, but Springboard's responsibilities extend above and beyond. SpringBoard is apparently responsible for every type of action in iOS. Even if it is not the foreground application, if it is stopped (by signal) no UI events get to the active app, and when it is continued all the events queued are delivered to the app.

Springboard is a multithreaded application. It has far more threads than Finder. Apple's developers were kind enough to name some of them (using the `pthread_setname_np`). The names reveal two Web related threads (WebCore and WebThreads), at least two belonging to `coremedia.player`, one for the WiFiManager callbacks (responsible for the WiFi indicator on the status bar), and three or more threads used for CoreAnimation. Debugging the process requires getting past a system watchdog, which reboots the system if SpringBoard is not responsive for more than a few minutes.

More information can be gleaned from Springboard's launchd registration, i.e., the `com.apple.SpringBoard.plist` entry in `/System/Library/LaunchDaemons`, shown in Listing 7-11. Since all

Mach port registrations go through launchd, this lists the (many) ports which SpringBoard requests launchd to register.

LISTING 7-11: SpringBoard's registered Mach ports

```
<plist version="1.0">
<dict>
        <key>EmbeddedPrivilegeDispensation</key>
        <true/>
        <key>HighPriorityIO</key>
        <true/>
        <key>KeepAlive</key>
        <true/>
        <key>Label</key>
        <string>com.apple.SpringBoard</string>
        <key>MachServices</key>
        <dict>
                <key>PurpleSystemEventPort</key>
                <dict>
                        <key>ResetAtClose</key>
                        <true/>
                </dict>
                <key>com.apple.CARenderServer</key>
                <dict>
                        <key>ResetAtClose</key>
                        <true/>
                </dict>
                <key>com.apple.SBUserNotification</key>
                <true/>
                <key>com.apple.UIKit.statusbarserver</key>
                <true/>
                <key>com.apple.bulletinboard.observerconnection</key>
                <true/>
                <key>com.apple.bulletinboard.publisherconnection</key>
                <true/>
                <key>com.apple.bulletinboard.settingsconnection</key>
                <true/>
                <key>com.apple.chatkit.clientcomposeserver.xpc</key>
                <true/>
                <key>com.apple.iohideventsystem</key>
                <dict>
                        <key>ResetAtClose</key>
                        <true/>
                </dict>
                <key>com.apple.smsserver</key>
                <dict>
                        <key>ResetAtClose</key>
                        <true/>
                </dict>
                <key>com.apple.springboard</key>
                <dict>
                        <key>ResetAtClose</key>
                        <true/>
```

continues

LISTING 7-11 *(continued)*

```
                </dict>
                <key>com.apple.springboard.UIKit.migserver</key>
                <dict>
                        <key>ResetAtClose</key>
                        <true/>
                </dict>
                <key>com.apple.springboard.alerts</key>
                <dict>
                        <key>ResetAtClose</key>
                        <true/>
                </dict>
                <key>com.apple.springboard.appstatechanged</key>
                <dict>
                        <key>HideUntilCheckIn</key>
                        <true/>
                </dict>
                <key>com.apple.springboard.backgroundappservices</key>
                <dict>
                        <key>HideUntilCheckIn</key>
                        <true/>
                        <key>ResetAtClose</key>
                        <true/>
                </dict>
                <key>com.apple.springboard.blockableservices</key>
                <dict>
                        <key>ResetAtClose</key>
                        <true/>
                </dict>
                <key>com.apple.springboard.processassertionservices</key>
                <dict>
                        <key>HideUntilCheckIn</key>
                        <true/>
                        <key>ResetAtClose</key>
                        <true/>
                </dict>
                <key>com.apple.springboard.processinvalidation</key>
                <dict>
                        <key>HideUntilCheckIn</key>
                        <true/>
                </dict>
                <key>com.apple.springboard.remotenotifications</key>
                <dict>
                        <key>ResetAtClose</key>
                        <true/>
                </dict>
                <key>com.apple.springboard.services</key>
        <dict>
                       <key>HideUntilCheckIn</key>
                       true/>
                       <key>ResetAtClose</key>
                       <true/>
                <key>com.apple.springboard.watchdogserver</key>
```

```
                    <true/>
            </dict>
            <key>ProgramArguments</key>
            <array>
                  <string>/System/Library/CoreServices/SpringBoard.app/SpringBoard</string>
            </array>
            <key>ThrottleInterval</key>
            <integer>5</integer>
            <key>UserName</key>
            <string>mobile</string>
    </dict>
    </plist>
```

Chief among all these ports is the `PurpleSystemEventPort`, which handles the UI events as `GSEvent` messages. This is understandably undocumented by Apple, but has been reverseengineered[2]. The main thread in Springboard calls processes `GSEventRun()`, which is the CF RunLoop that handles the UI messages. The other threads are in similar run loops over the other Mach ports in Springboard, but due to the opaque nature of these ports, it's difficult to tell which thread is on which port without the right symbols.

XPC (LION AND IOS)

XPC is a set of lightweight interprocess communication primitives first introduced in Lion and iOS 5. XPC is fairly well documented in Apple Developer[3]. It is also tightly integrated with the Grand Central Dispatcher (GCD). XPC enables a developer to break down applications into separate components. This improves both application stability and security, as vulnerable (or unstable) functionality can be contained in an XPC service, which is managed externally — another responsibility happily assumed by launchd.

Just as with its own `LaunchDaemons`, launchd takes on the tasks of starting XPC services on demand, watching over them (restarting on crash), and terminating them (the hard way, with a `kill -9`) when they are done or idle. The launchd uses `xpcd(8)`, `xpchelper(8)`, and `xpcproxy(8)` to assist with the XPC services. It maintains XPC services alongside standard Mach services, in separate XPC domains — per-user, private, and singleton. This can be seen in the output of `launchctl`'s `bstree` subcommand, as shown in Output 7-4:

OUTPUT 7-4: XPC Service Domains

```
root@Simulacrum (/)# launchctl bstree | grep Domain
com.apple.xpc.domain.com.apple.dock.[231] (XPC Private Domain)/
    com.apple.xpc.domain.Dock[175] (XPC Private Domain)/
    com.apple.xpc.domain.peruser.501 (XPC Singleton Domain)/
    com.apple.xpc.domain.imagent[214] (XPC Private Domain)/
    com.apple.xpc.domain.com.apple.audio[203] (XPC Private Domain)/
    com.apple.xpc.domain.peruser.202 (XPC Singleton Domain)/
    com.apple.xpc.domain.coreaudiod[108] (XPC Private Domain)/
    com.apple.xpc.system (XPC Singleton Domain)/
        ...
```

XPC services and client applications link (either directly or through Cocoa) with `libxpc.dylib`, which provides the various C-level XPC primitives (such as Mountain Lion's `NSXPCConnection`). The library remains closed source at the time of this writing, but Apple does provide the `<xpc/*>` includes which expose the APIs, whose internals are discussed in this section. XPC also relies on the private frameworks of `XPCService` and `XPCObjects`. The former handles runtime aspects of services, and the latter provides encoding and decoding services for XPC objects. iOS contains a third private framework, `XPCKit`.

XPC Object Types

XPC wraps and serializes various datatypes in a manner akin to the `CoreFoundation` framework. `<xpc/xpc.h>` defines the object and data types supported by XPC, shown in Table 7-5. The type names are `#defined` as `XPC_TYPE_`*`typename`* macros wrappings pointers to the corresponding types in the table, and can be instantiated with `xpc_`*`typename`*`_create` functions. Objects can be retrieved from messages in most cases using `xpc_`*`typename`*`_get_value`. Two special object types are dictionaries and arrays, which serve as containers for other object types (which may be created in or accessed from from them using `xpc_[array|dictionary]_[get|set]_`*`typename`*.

TABLE 7-5: XPC Object and data types

TYPE	REPRESENTS
`connection`	An XPC connection, over which messages can be sent and received. A connection can be created using `xpc_connection_create()`, specifying an anonymous or named connection, or from a given endpoint, through a call to `xpc_connection_create_from_endpoint()`.
`endpoint`	Serializable form of a connection. Effectively a connection factory.
`null`	A null object reference (constant) for comparisons.
`bool`	A Boolean.
`true/false`	Boolean true/false values (constants) for comparisons.
`int64/uint64`	Signed/Unsigned 64-bit integers.
`double`	Double precision floats.
`date`	Date intervals (UNIX time). Can be instantiated from the present time by a call to `xpc_date_create_from_current`.
`data`	Array of bytes. The recipient can obtain a pointer to the data by calling `xpc_data_get_bytes_ptr`.
`string`	Null terminated C-String (wraps char *). Strings may be created with a format string, and even with variable arguments (similar to `vsprintf(3)`). The recipient can obtain a pointer to the string by calling `xpc_string_get_string_ptr`.

TYPE	REPRESENTS
uuid	Universally Unique Identifier. The recipient can obtain the UUID by a call to xpc_uuid_get_bytes.
fd	File descriptor. The descriptor can be used by the client by calling xpc_fd_dup.
shmem	Shared memory. The shared memory can be mapped into the receipient's address space by calling xpc_shmem_map.
array	Indexed array of XPC objects. An array may contain any number of other object types, which may be added to it or retrieved from it using xpc_array_[get|set]_*typename*.
dictionary	Associative array of XPC objects. A dictionary may contain any number of other object types, which may be added to it or retrieved from it using xpc_dictionary_[get|set]_*typename*.
error	Error objects. Used for returning errors. Cannot be instantiated by clients.

Any of the XPC objects can be handled as an opaque xpc_object_t, and manipulated by functions described in xpc_object(3). These include xpc_retain/release, xpc_get_type (which returns one of the XPC_TYPEs corresponding to Table 7-5), xpc_hash (used to provide a hash value of an object for array indexing), xpc_equal (for comparing objects) and xpc_copy.

XPC Messages

Objects may be sent or received in messages. Messages are sent using one of several functions from <xpc/connection.h>, as shown in Table 7-6:

TABLE 7-6: XPC Messaging functions in <xpc/connection.h>

FUNCTION	USAGE
xpc_connection_send_message (xpc_connection_t *connection*, xpc_object_t *message)*;	Send *message* asynchronously on *connection*.
xpc_connection_send_barrier (xpc_connection_t *connection*, dispatch_block_t *barrier)*;	Execute *barrier* block after last message is sent on *connection*.
xpc_connection_send_message_with_reply (xpc_connection_t *connection*, xpc_object_t *message*, dispatch_queue_t *replyq*, xpc_handler_t *handler)*;	Send message, but also asynchronously execute *handler* in dispatch queue *replyq* when a reply is received.

continues

TABLE 7-6 *(continued)*

`xpc_object_t` **`xpc_connection_send_message_with_reply_sync`** `(xpc_connection_t connection, xpc_object_t message);`	Send *`message`*, blocking until a reply is received, and return reply as the `xpc_object_t` return value

By default, messages are sent asynchronously, and are handled by dispatch queues (i.e., GCD), as shown in Figure 7-2. By using *barriers*, the programmer may provide a block to be executed when all the messages on a particular connection have been sent. Messages may expect replies, which are again asynchronous, though the `_reply_sync` function may be used to block until a message is received.

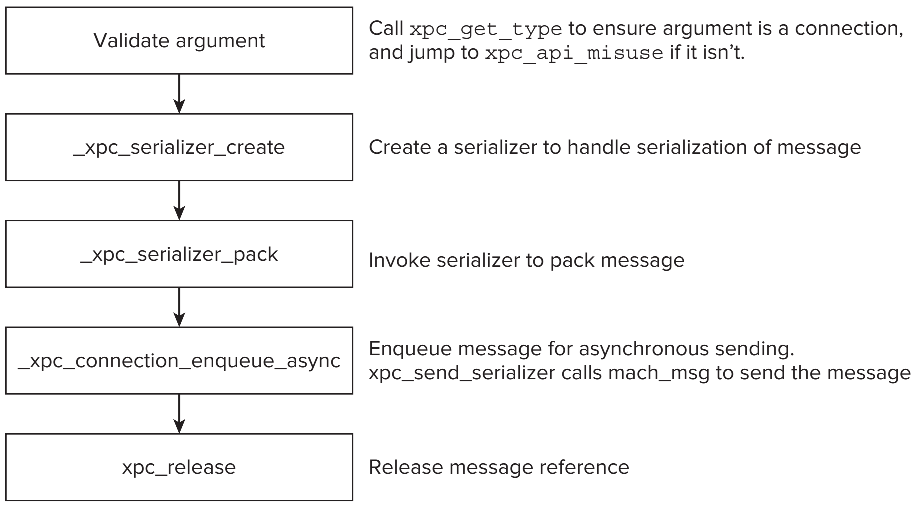


FIGURE 7-2: Flow of xpc_connection_send_message

XPC messages are implemented over Mach messages and make use of the Mach Interface Generator (MIG) facility, which provides the `xpc_domain` subsystem. This subsystem contains messages to check in, load, or add services, and get the name of a service, similar to the bootstrap protocol described earlier in this chapter (XPC can be considered a subset of bootstrap, and makes use of it internally). Mach messages and in particular MIG are detailed in Chapter 10.

XPC services

XPC services can be created in Objective-C, or in C/C++. In either case, the services are started by a call to `libxpc.dylib`'s `xpc_main`. C/C++ services' main is just a simple wrapper, which invokes `xpc_main` (declared in `<xpc/xpc.h>`) with the event handler function (`xpc_connection_handler_t`). Objective-C services also call on `xpc_main()`, albeit indirectly through `NSXPCConnection`'s resume method.

The event handler function takes a single argument, an `xpc_connection_t`. (Objective-C wraps this object with `Foundation.framework`'s `NSXPCConnection`.) The XPC connection is treated as an opaque object, with miscellaneous xpc_connection_* functions. In <xpc/connection.h> used as getters for its properties, and setters for its event handler and target queue. A connection's name, effective UID and GID, PID and Audit Session ID can all be queried.

The normal architecture of an XPC service involves calling dispatch_queue_create to create a queue for the incoming messages from the client and using xpc_connection_set_target_queue to assign the queue to the connection. The service also sets an event handler on the connection, calling xpc_connection_set_event_handler with a handler block (which may wrap a function). The handler is called whenever the service receives a message. A service may create a reply (by calling xpc_dictionary_create_reply) and send it.

A well-documented example of XPC is SandBoxedFetch, which is available from Apple Developer[4], alleviating the need for an example in this book.

XPC Property Lists

XPC services are defined in their own bundles, contained in an XPCServices subfolder of its parent application or framework. As with all bundles, they have an Info.plist, which they use to declare various service properties and requirements:

- The CFBundlePackageType property is defined as "XPC!"
- The CFBundleIdentifier property defines the name of the XPCService. This is set to be the same as the bundle's name.
- The XPCService property defines a dictionary, which can specify the ServiceType property (Application. User or System), and RunLoopType (dispatch_main or NSRunLoop), which dictates which run loop style xpc_main() adopts. The dictionary may also contain the JoinExistingSession Boolean property, to redirect auditing to the application's existing audit session.
- The XPCService dictionary may be used to specify additional properties, prefixed by an underscore. These include _SandboxProfile (which allows the optional specification of a sandbox profile to enforce on the XPC service, as discussed in Chapter 4) and _AllowedClients, which can specify the identifiers of applications which are allowed to connect to the service.

SUMMARY

This chapter discussed launchd, the OS X and iOS replacement to the traditional UNIX init. launchd fills many functions in both operating systems: both those of UNIX daemons, and those of Mach. The Mach roles will be discussed further when the concept of Mach messages is elaborated on in Chapter 10.

The chapter ended with a review of the GUI of both OS X (Finder) and iOS (SpringBoard), in as much detail as possible on these intentionally undocumented binaries.

REFERENCES AND FURTHER READING

1. launchd Sources, `http://opensource.apple.com/tarballs/launchd/launchd-392.38.tar.gz` or later.
2. GSEvent iPhone Development Wiki, `http://iphonedevwiki.net/index.php/GSEvent`
3. Apple Developer, "Daemons and Services Programming Guide" `http://developer.apple.com/library/mac/#documentation/MacOSX/Conceptual/BPSystemStartup/Chapters/CreatingXPCServices.html`
4. Apple Developer, "Sandboxed Fetch" `http://developer.apple.com/library/mac/#samplecode/SandboxedFetch/`

PART II

The Kernel

- ▶ **CHAPTER 8:** Some Assembly Required: Kernel Architectures
- ▶ **CHAPTER 9:** From the Cradle to the Grave — Kernel Boot and Panics
- ▶ **CHAPTER 10:** The Medium Is the Message: Mach Primitives
- ▶ **CHAPTER 11:** Tempus Fugit — Mach Scheduling
- ▶ **CHAPTER 12:** Commit to Memory: Mach Virtual Memory
- ▶ **CHAPTER 13:** BS"D — The BSD Layer
- ▶ **CHAPTER 14:** Something Old, Something New: Advanced BSD Aspects
- ▶ **CHAPTER 15:** Fee, FI-FO, File: File Systems and the VFS
- ▶ **CHAPTER 16:** To B (-Tree) or Not to Be — The HFS+ File Systems
- ▶ **CHAPTER 17:** Adhere to Protocol: The Networking Stack
- ▶ **CHAPTER 18:** Modu(lu)s Operandi — Kernel Extensions
- ▶ **CHAPTER 19:** Driving Force — I/O Kit

8

Some Assembly Required: Kernel Architectures

Before we delve into the OS X kernel internals, we present the basic ideas and architectures associated with and shared by all operating systems on all platforms: user mode, kernel mode, hardware separation, and a focus on the kernel's tight programming constraints and real-mode environment.

The kernel is the most critical part of any operating system. As such, it has to be highly optimized to take advantage of all the features and capabilities of the underlying CPU. Kernels are, for the most part, written in C in order to be as close as possible to the machine, while keeping the code maintainable. In some cases, however, there is no choice but to get closer still, and use-architecture-specific assembly.

Likewise, there is little choice left for those wishing to understand the kernel, but to wade into the quagmire that is assembly. The outputs and listings in this chapter contain a fair share of assembly — both Intel (for OS X) and ARM (for iOS). Unfortunately, the two variants are distinct languages, as foreign to each other as English is to Mandarin. A complete explanation of either is well beyond the scope of the book. The intrepid reader, however, is more than encouraged to check out the Intel[1] and ARM[2] manuals for the complete syntax, or consult the appendix in this book for a quick overview and comparison of both architectures.

KERNEL BASICS

All modern operating systems incorporate in their design a component called the *kernel*. This, like the kernel (or seed) of a fruit, is the innermost part of the system — its core. The kernel *is* the operating system. From a high-level view, the applications you run — from word processors to games — are all effectively *clients* of the kernel, which provides various services, or *system calls*.

The reasoning for a kernel becomes readily apparent when the developer's point of view is considered — if a developer had to write applications that would work on all types of hardware, and all classes of environments, she would find herself bogged down in a quagmire of decision-making. How does one interface with the hard drive? The network? The graphics adapter? The average developer could not care less about the idiosyncrasies of hardware devices. What's more, if the developer had to build, from scratch, the code required for device and file access every time, it would inflate both the size of the programs, as well as the time required to code them. There needs to be, therefore, some level of abstraction, which enables a developer to write code that is portable across the same operating system, but over different types of hardware. The kernel thus provides a level of *virtualization*. This is accomplished by an API that deals with abstract objects — in particular, virtual memory, network interfaces, and generic devices.

The kernel also serves as a *scheduler*. All modern operating systems are *preemptive multitasking* systems — with "multitasking" meaning they allow several programs, or tasks, to run concurrently. In actuality, though, the number of programs is far greater than the number of processors (or cores). The kernel therefore has to decide which program (process, or thread) can run on which processor/core.

The kernel is an *arbiter* — when programs seek to access shared devices, like the hard drive, display, or network adapters, there needs to be some form of scheduling, to avoid access conflicts or bottlenecks.

Another set of services offered by the kernel are *security services* — most often noticeable by the user as *permissions* and *rights,* these are mechanisms to ensure the integrity, privacy, and fair use of the system's various resources. As an added layer to arbitration, any potentially sensitive operation (and practically all access to system resources) must first pass through a security check. The kernel is responsible for performing that check, and enforcing the various permissions, though the system administrator can toggle and tweak the actual permissions themselves.

Kernel Architectures

All operating system designs include kernels, but the kernels are designed differently. There are three classes of kernels, and they are discussed next.

Monolithic Kernels

The Monolithic architecture is the "classic" kernel architecture, and is still predominant in the UNIX and Linux realms. The term "monolithic" comes from Greek — meaning "single rock" or "single chunk." A monolithic kernel follows the approach of putting all the kernel functionality — whether fundamental or advanced — in one address space. In this way, thread scheduling, and memory management are squeezed alongside file systems, security management, and even device drivers.

To better understand the monolithic architecture, consider the layout of the Linux kernel, which is very close in its implementation to the standard UN*X kernel. This is shown in Figure 8-1.

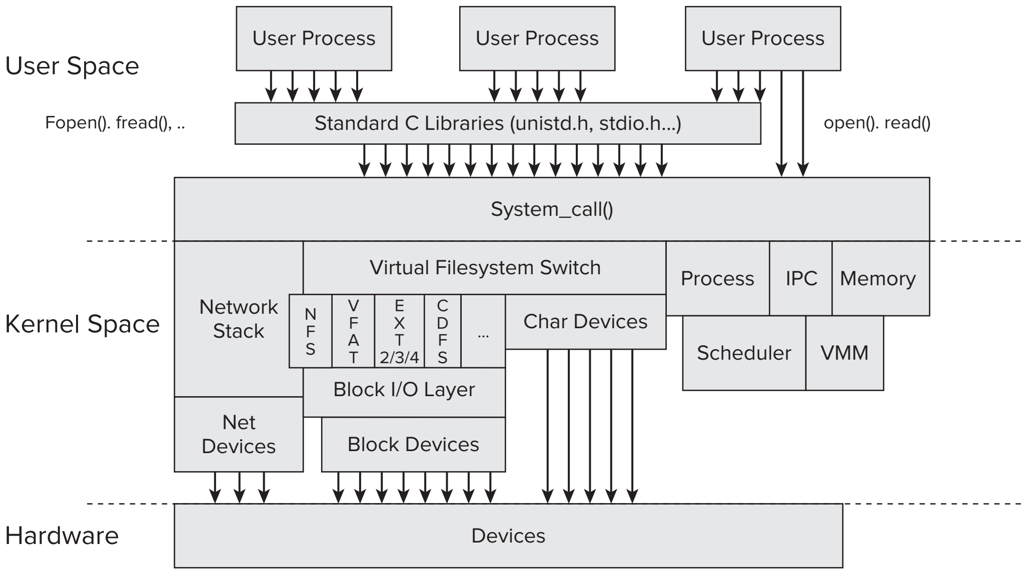


FIGURE 8-1: The Linux kernel architecture

All the kernel functionality is implemented in the same address space. To further optimize, monolithic kernels not only group all functionality into the same address space, but further map that address space into every processes' memory. This is shown in Figure 8-2. In Linux, for example, of the 4 GB of addressable memory in a 32-bit application, 1 GB is sacrificed in the name of the kernel (On Windows 32-bit: 2 GB). Trying to set a pointer to an address above `0xC0000000` (Windows: `0x80000000`) will cause a memory violation (segmentation fault), as the memory is inaccessible from user mode.

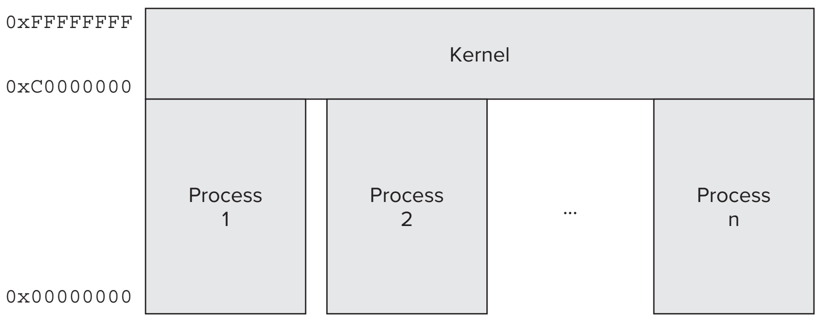


FIGURE 8-2: The monolithic kernel architecture

Sacrificing so much memory — which, in 32-bit mode, makes for one quarter of the entire available amount — only makes sense if there is a significant advantage, and indeed there is: switching from user mode to kernel mode in a monolithic architecture is highly efficient, essentially as costly as a thread switch. This is due to the kernel's memory pages being resident in all processes, so that — aside from the kernel/user hardware enforced separation — there is really no difference between the two. All processes, regardless of owner or function, contain a copy of the kernel memory, just as they would contain copies of shared libraries. Further, these copies (again, like shared libraries) are all mapped to the same set of physical pages, which are resident. This not only saves precious RAM, but means that no significant costs (such as page faults) are associated with performing a system call. This is especially important, given the ubiquity of system calls in user code.

In 64-bit architectures the reservation is larger by several orders of magnitude: the top 40–48 bits, depending on OS configuration, accounting for a whopping 1–256 TB of virtual memory. Unlike the 32-bit case, however, this really isn't restrictive, since user mode has a like amount of addressable memory, which processes don't even begin to scratch the surface of, and RAM alone could not back anyway.

Microkernels

While less common, The microkernel architecture is of special interest to us, as Mach, the innermost component of XNU, is built this way.

A microkernel consists of only the core kernel functionality, in a minimal code-base. Only the critical aspects — usually task scheduling and memory management — are carried out by the kernel proper, with the rest of the functionality exported to external (usually user mode) servers. There exists complete isolation between the individual servers, and all communication between them is carried out by *message passing*: a mechanism allowing the delivery of (usually opaque) message structures and their subsequent queuing in each server's queue, from which said component can later de-queue and process each, in turn. Figure 8-3 shows this architecture:

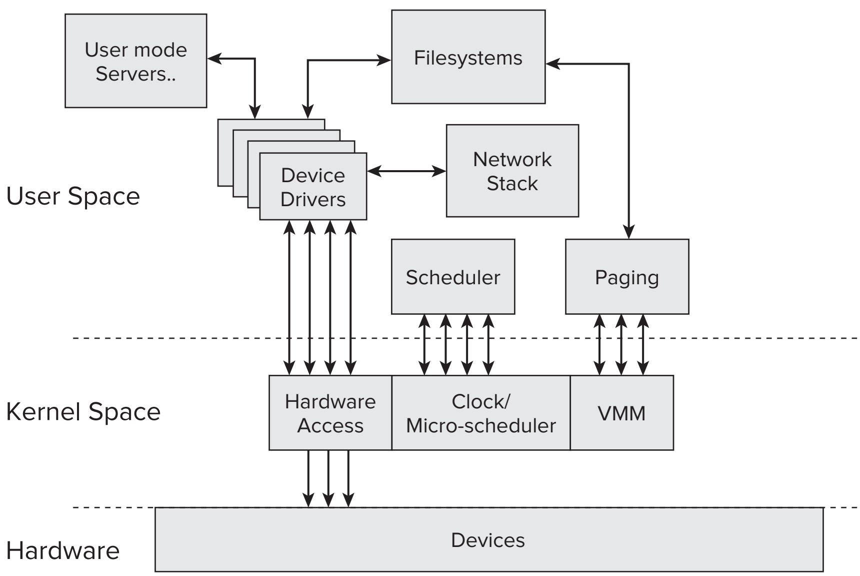


FIGURE 8-3: The microkernel architecture

Microkernels offer several distinct advantages, which their monolithic brethren cannot. The first is *correctness*: being a small code base allows for the verification, by traversal of all code paths, of correct functionality. What follows is stability and *robustness*, as a microkernel has very few points of possible failure, if any. Since all the additional functionality is provided by external and independent servers, any failure is contained, and can be easily overcome by restarting the affected server component. This is really not that different than a failure in a user process (think, when your browser or other application crashes), wherein that process can be restarted. By contrast, monolithic kernel failures more often than not trigger a complete kernel panic.

Another advantage of microkernels is their flexibility, and adaptability to different platforms and architectures. Because their functionality is so well defined, it is relatively straightforward to port it to other architectures. This can, in theory, be further extended to remote components (that is, a true network-based operating system), as there is no real constraint that message passing be confined to a single node.

Advantages on the one hand, there is one specific disadvantage on the other which outweighs most of them — and that is performance. Microkernel message passing translates to memory-copy operations, and several context-switch operations, neither of which are cheap in terms of computational speed. This disadvantage is so significant, that "pure" microkernels are still largely academic, and not used commercially, much less so in contemporary operating systems. This calls for a third, synthetic approach — hybridization.

Hybrid Kernels

Hybrid kernels attempt to synthesize the best of both worlds. The innermost core of the kernel, supporting the lowest level services of scheduling, inter-process communication (IPC) and virtual memory, is self-contained, as would be a microkernel. All other services are implemented outside this core, though also in kernel mode and in the same memory space as the core's.

Another way to look at this is as if the kernel contains within it a smaller autonomous core. Unlike a true microkernel design, however, this does not mandate message passing. The "kernel-within" is often just a self-contained modular executable, meaning other components may call on it for services, but it does not call out. Note, however, that a hybrid kernel does not enjoy the robustness of a microkernel, having sacrificed it in return for the efficiency of the monolithic kind.

IS XNU A MICRO, MONOLITHIC, OR HYBRID KERNEL?

Technically, XNU is a *hybrid kernel*. The Windows kernel is also classified as a hybrid, yet the differences between them are so significant that using "hybrid" to describe both is a very loose and possibly misleading term.

Windows does contain a microkernel like core, but the executive, NTOSKRNL (or NTKRNLPA), itself is closer to a monolithic kernel. The kernel APIs make a distinction between the Ke prefixed functions (the kernel core) and all the rest, but all are in the same address space: kernel space is reserved by default in the upper 2 GB of every process (44 or 48 bits in 64-bit mode), exactly as it would be in a monolithic architecture. A crash in kernel mode, such as a bug in a driver, leads to the infamous "blue screen of death," just like a kernel panic in UNIX.

continues

(continued)

OS X's XNU is also a hybrid, but is somewhat closer to a microkernel than Windows is. Mach, its core, was originally a true microkernel, and its primitives are still built around a message passing foundation. The messages, however, are often passed as pointers, with no expensive copy operations. This is because most of its servers now execute in the same address space (thereby classifying as monolithic). Likewise, the BSD layer on top of Mach, which was always a monolith, is in that same address space.

Still, unlike Windows or Linux, OS X applications in 32-bit (Intel) used to enjoy a largely unfettered address space with virtually no kernel reservation — that is, the kernel had its own address space. Apple has conformed, however, and in 64-bit mode OS X behaves more like its monolithic peers: the kernel/user address spaces are shared, unless otherwise stated (by setting the `-no-shared-cr3` boot argument on Intel architectures). The same holds true in iOS, wherein XNU currently reserves the top 2 GB of the 4 GB address space (prior to iOS version 4 the separation was 3 GB user/1 GB kernel).

USER MODE VERSUS KERNEL MODE

The kernel is a trusted system component. As we have seen, it controls the most critical functions. There needs to be a strict separation between the kernel functionality, and that of applications. Otherwise, application instability might bring down the system. In the Microsoft realm, this was quite common in the days of DOS and Windows, before the advent of Windows NT based systems (such as NT, 2000, XP, and later). Further, this strict separation needs to be enforced by the hardware, as software-based enforcement is both costly (in terms of performance), and unreliable.

Intel Architecture — Rings

Intel-based systems provide the required hardware based separation. Beginning with the 286 processor (with major enhancements in the 386 processors), Intel introduced the notion of "protected mode." Intel x86 systems still boot in "real mode" (for compatibility), but all kernels switch the CPU to protected mode upon startup. This is accomplished by setting one of the four special-purpose Control Registers — `CR0` — and toggling on its least-significant bit. This operation is always performed by assembly instructions — C and other languages have no access to the Control Registers. The code to do so in XNU is in `start.s`, for both `i386` and `x86_64` branches, shown in Listing 8-1:

LISTING 8-1: osfmk/x86_64/start.s

```
Entry(real_mode_bootstrap_base)
        cli
        LGDT(EXT(protected_mode_gdtr))
        /* set the PE bit of CR0 */
        mov     %cr0, %eax      ; can't operate on CRs directly
        inc %eax                ; add 1 toggles on the least significant bit
        mov     %eax, %cr0      ; update CR0
```

Protected mode enforces 4 "rings." These "rings" are privilege levels, numbered 0 through 3. They are modeled in a concentric fashion, with the innermost ring being ring 0, and the outermost ring 3. Ring 0 is the most sensitive, and is often referred to as Supervisor mode. Code on the processor running in ring 0 is the most trusted, and virtually omnipotent. As the ring levels increase, so do security restrictions and privileges — so that code in ring 3 is least trusted, and most restricted.

Ring 0 naturally maps to kernel mode, and ring 3 — to user mode. Rings 1 and 2 are reserved for operating system services, but — in practice — are unused. The rings are implemented by two bits in the CS register, and two corresponding bits in the EFLAGS register, to set the "user privilege level" and "current privilege level" as part of the thread state. It is therefore not uncommon to see code in the kernel check the bits in CS, and bitwise-AND them with 0x3, as a way to check user/kernel mode on kernel entry.

Certain assembly instructions are disallowed anywhere but ring 0. These include direct access to hardware, manipulating the control registers, accessing protected memory regions, and many others. If a program attempts to execute such operations, the CPU generates a *general protection fault* (Interrupt #13), and further execution of that code is forbidden. (If protected mode were not enforced at the hardware level, any program that could access the control registers could switch between rings).

Code in a lower ring can easily switch to a higher ring, but moving from a higher ring to a lower ring is impossible, unless a *call gate* mechanism has been previously established by the lower ring. We will cover these in "Kernel/User Transition Mechanisms," later.

> *Virtualization note: newer processors, which support hardware based virtualization, (such as Intel Vt-X and AMD-V) also offer an inner ring, "ring -1," or "hypervisor mode." This ring allows virtualization-enabled operating systems, such as VMWare ESX, to load prior to the guest operating systems, and offer their kernels full ring 0 functionality.*

ARM Architecture: CPSR

ARM processors use a special register, the *current program status register (CPSR)* to define what mode they are in. The processors have no less than seven distinct modes of operation, but as Table 8-1 shows, there is still a clear dichotomy:

TABLE 8-1: ARM processor modes

MODE	MODE BITS	PURPOSE
`USR`	`10000`	User — Non-privileged operations
`SVC`	`10011`	Supervisor mode (default kernel mode)
`SYS`	`11111`	System — As user, but the CPSR is writable

continues

TABLE 8-1 *(continued)*

MODE	MODE BITS	PURPOSE
`FIQ`	`10001`	Fast Interrupt Request
`IRQ`	`10010`	Normal Interrupt request
`ABT`	`10111`	Abort — Failed memory access
`UND`	`11011`	Undefined — Illegal/unsupported instruction

USR is the only non-privileged mode. All other modes are privileged, though the kernel usually operates in SVC. In any of the privileged mode, the CPSR can be accessed directly, so switching modes is as trivial as setting the mode bits. From user mode, one of the user/kernel transition mechanisms (discussed next) must be used. The other modes of IRQ and FIQ are used for interrupt processing (ARM distinguishes between normal interrupts and fast ones. In IRQ mode, normal interrupts are masked, but fast ones may still interrupt the processor. In FIQ mode, both interrupts are masked). ABT is used only on memory faults, and UND is used for operations which are either illegal or unsupported, allowing predefined handlers to take over and emulate any instructions, which the hardware does not natively support.

KERNEL/USER TRANSITION MECHANISMS

As the previous section showed, the separation between kernel mode and user mode is critical, and thus provided by the hardware. But applications frequently need kernel services, and therefore the transition between the two modes needs to be implemented in a manner that is highly effective, but at the same time highly secure.

There are two types of transfer mechanisms between user mode and kernel mode:

- *Voluntary* — When an application requires a kernel service, it can issue a call to kernel mode. By using a predefined hardware instruction, a switch to kernel mode may be initiated. These services are called *system calls (*recall our discussion in 2.8)
- *Involuntary* — When some execution exception, interrupt or processor trap occurs, code execution is suspended, frozen at the exact state when the fault occurred. Control is transferred to a predefined fault handler or interrupt service routine (ISR) in kernel mode.

Another dichotomy of control transfers often used is of asynchronous versus synchronous. The synchronous control transfer occurs "in sync" with the program flow — and is the result of some instruction, which resulted in a runtime anomalous condition. The asynchronous control transfer, by contrast, occurs when the program is interrupted by an external source (the interrupt controller). This is "out of sync" with the program, which would have continued normally if not for the interruption, which must be handled.

Whichever classification you choose to view them by, all types of control transfer are secure, in that they must be predefined by kernel mode code, and user mode code has no way whatsoever of changing them. User mode, in fact, is completely oblivious to the kernel "taking over," especially in involuntary control transfers.

The kernel sets the predefined entry points in an interrupt dispatch table (IDT) (per the Intel nomenclature), or the exception vector (per that of ARM. The two terms refer to the same idea: a one-dimensional array wherein the predefined function pointers are stored. Much like a user-mode `setlongjmp()` or signal handler, the CPU will jump to the function pointer and execute the function — with the additional effect of moving to supervisor mode.

Trap Handlers on Intel

The Intel architecture defines an interrupt vector of 255 entries, or *cells*. This vector is populated by the kernel when the system boots.

Exceptions — Traps/Faults/Aborts

On Intel, the first 20 cells of the Intel interrupt vector are defined for *exceptions*; these are all kinds of special abnormal conditions that can be encountered by the processor while executing code. They are shown in Table 8-2, along with their corresponding XNU handler names:

TABLE 8-2: Intel exceptions — traps and faults

#	EXCEPTION	OCCURS WHEN	XNU HANDLER NAME
0	Divide error fault	`DIV` and `IDIV` fail (e.g. zero divide)	`idt64_zero_div`
3	Break point trap	Debugger breakpoint	`idt64_int3`
4	Overflow trap	`INT 0` opcode	`idt64_into`
5	Bound range exceeded fault	`BOUND` opcode	`idt64_bounds`
6	Invalid opcode fault	Illegal instructions	`idt64_invop`
7	Math CoProcessor fault	FPU errors	`idt64_nofpu`
8	Double fault (abort)	Generated the second time a fault occurs on the same instruction	`idt64_double_fault` or `idt64_db_task_dbl_fault`
9	FPU Overflow	FPU overflow condition	`idt64_fpu_over`
10	Invalid TSS fault	Bad Task State Segment	`idt64_inv_tss`
11	Segment not present fault	Accessing protected segments	`idt64_segnp`

continues

TABLE 8-2 *(continued)*

#	EXCEPTION	OCCURS WHEN..	XNU HANDLER NAME
12	Stack segment fault	Stack segment errors	`idt64_stack_fault` or `idt64_db_task_stk_fault`
13	General Protection fault	Memory fault, or other access check	`idt64_gen_prot`
14	Page fault	Page not accessible, page swapped out	`idt64_page_fault`
16	Math fault	FPU generated	`Idt64_tfpu_err`
17	Alignment check fault	Data is unaligned on a `DWORD` or other boundary	`idt64_trap11`
18	Machine check abort	Hardware reported errors	`idt64_mc`
19	SIMD Floating point fault	SSEx instructions	`idt64_sse_err`

As you can see from the table, there are three types of exceptions:

- *Faults* — Occur when an instruction encounters an exception that can be corrected and the instruction can be restarted by the processor. A common example is a page fault, which occurs when a virtual memory address is not present in physical RAM. The fault handler is executed, and returns to the very same instruction that generated the fault.
- *Traps* — Are similar to faults, but the fault address returns to the instruction *after* the trap.
- *Aborts* — Cannot be restarted. In the table above, a "double fault" (#8) is an abort, as if a fault is triggered twice in the same instruction, it does not make sense to retry.

Interrupts

The second kind of involuntary user/kernel transition occurs on an *interrupt*. An interrupt is generated by a special sub-component of the CPU, called a Programmable Interrupt Controller (PIC), or — in the more modern version — Advanced PIC (APIC). The PIC receives messages from the devices on the system bus, and multiplexes them to one of several Interrupt Request (IRQ) lines. When an interrupt is generated, the PIC marks the corresponding interrupt line as active. The line remains active until the interrupt is *handled* or serviced by a function (appropriately called the *Interrupt Handler*, or *Interrupt Service Routine*). It is up to that function to reset the line.

Legacy PICs, (called XT-PICs), only had 16 lines, ranging from 0 to 15. Modern APICs, however, allow for up to 255 such lines. IRQ lines can be *shared* by more than one device, if the need arises.

The IRQ lines were once reserved for certain devices, as shown in Table 8-3, which in some cases still use their "well known" lines. The PCI bus, however, dynamically allocates most IRQs.

TABLE 8-3: Traditional IRQ reservations (for non PCI or legacy devices)

IRQ	TRADITIONALLY USED FOR
0	Timer — the kernel can set this interrupt to occur at a fixed frequency, forming the basis for task scheduling
1	Keyboard — dating back to the old days where the user could actually generate keystrokes faster than the processor could handle them
3	Serial ports (Com 2 and Com 4)
4	Serial ports (Com 1 and Com 3)
14	Primary IDE
15	Secondary IDE

The general rule of thumb is, that interrupts can be dispatched as long as:

- The corresponding interrupt request line is not currently busy (indicating a previous interrupt has not yet been serviced) or masked (indicating the processor or core is ignoring this interrupt line)
- No lower numbered interrupt lines are busy
- The local CPU/core has not disabled all interrupts (by low-level CLI/STI assembly).

For example, a core will not receive an interrupt on `IRQ3` until `IRQ0`, `1` and `2` are all clear. While it is servicing `IRQ3`, interrupts `4` and higher (i.e. of lower priority) will not be delivered to the CPU. The timer interrupt (`IRQ0` or, on APICs, the dedicated local timer IRQ line) is always the one with the highest priority, as it is used to drive thread scheduling.

On a multi-core/SMP system, interrupts are dispatched per core (or processor), and the kernel may set "interrupt affinity" by temporarily or permanently masking specific interrupt lines of a core. The APIC is "smart" enough to dispatch interrupts to CPUs or cores which are not busy. If an interrupt cannot be dispatched, the APIC can usually queue it. But queuing capabilities are very limited. Interrupts that are "lost" or "dropped" may result in loss of data, or even system hangs, as a device may be reporting some critical event via an interrupt. Interrupts are therefore handled with the utmost priority of any other processing in the system — preempting everything else — and their handlers run for the minimum time necessary.

In Intel architectures, the IRQ lines are mapped to the processor's Interrupt Vectors, at a location higher than the first 32 entries (20 of which are from the Table 8-2 above, with the other 12 reserved).

Handling Traps and interrupts in XNU on Intel

XNU registers its trap handlers in `/osfmk/i386/idt.s` or `/osfmk/x86_64/idt_table.h`, as shown in Listing 8-2:

LISTING 8-2: XNU IDT Table, from osfmk/x86_64/idt_table.h

```
TRAP(0x00,idt64_zero_div)
TRAP_SPC(0x01,idt64_debug)
INTERRUPT(0x02)                  /* NMI */
USER_TRAP(0x03,idt64_int3)
USER_TRAP(0x04,idt64_into)
USER_TRAP(0x05,idt64_bounds)
TRAP(0x06,idt64_invop)
TRAP(0x07,idt64_nofpu)
       ..
 // handler registrations corresponding to table faultXXX
```

Rather than install separate handlers individually for every trap, most kernels usually install one handler for all the traps, and have that handler `switch()`, or jump according to a predefined table. XNU does exactly that by defining the `TRAP` and `USER_TRAP` macros (in `osfmk/x86_64/idt64.s`). These macros build on other macros (`IDT_ENTRY_WRAPPER` and `PUSH_FUNCTION`), to set up the stack as illustrated in Figure 8-4:

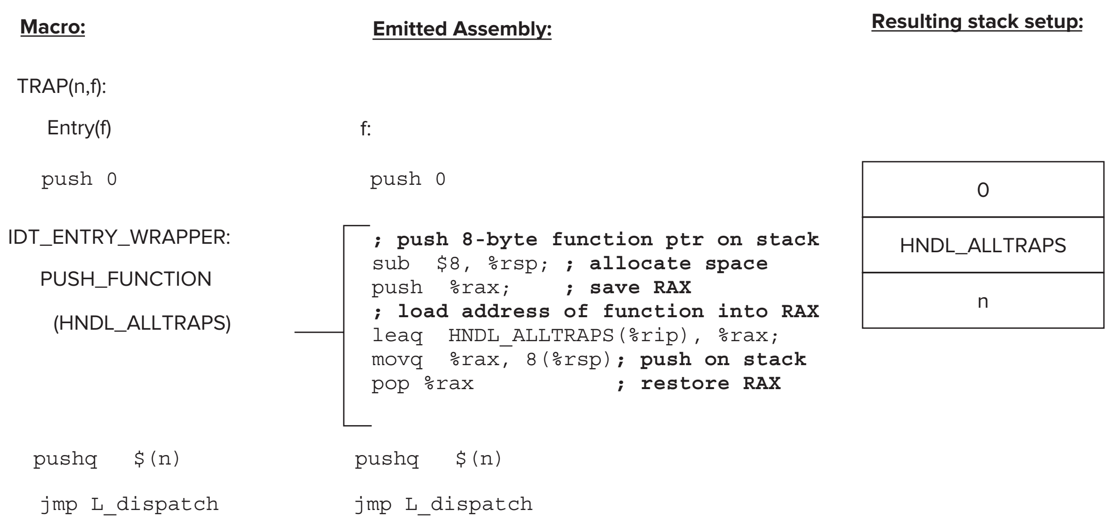


FIGURE 8-4: The `TRAP` macro expansion

In plain words, the `TRAP` macro simply defines the handler function as an entry point, pushes zero (or an error code, if any) on the stack, and pushes the address of the common trap handler — `HNDL_ALLTRAPS`, using the `IDT_ENTRY_WRAPPER` macro. Because the trap handler is a common one, the macro also pushes the trap number (`n`). It then jumps to `L_dispatch`, which serves as a common dispatcher, and flows according to Figure 8-5:

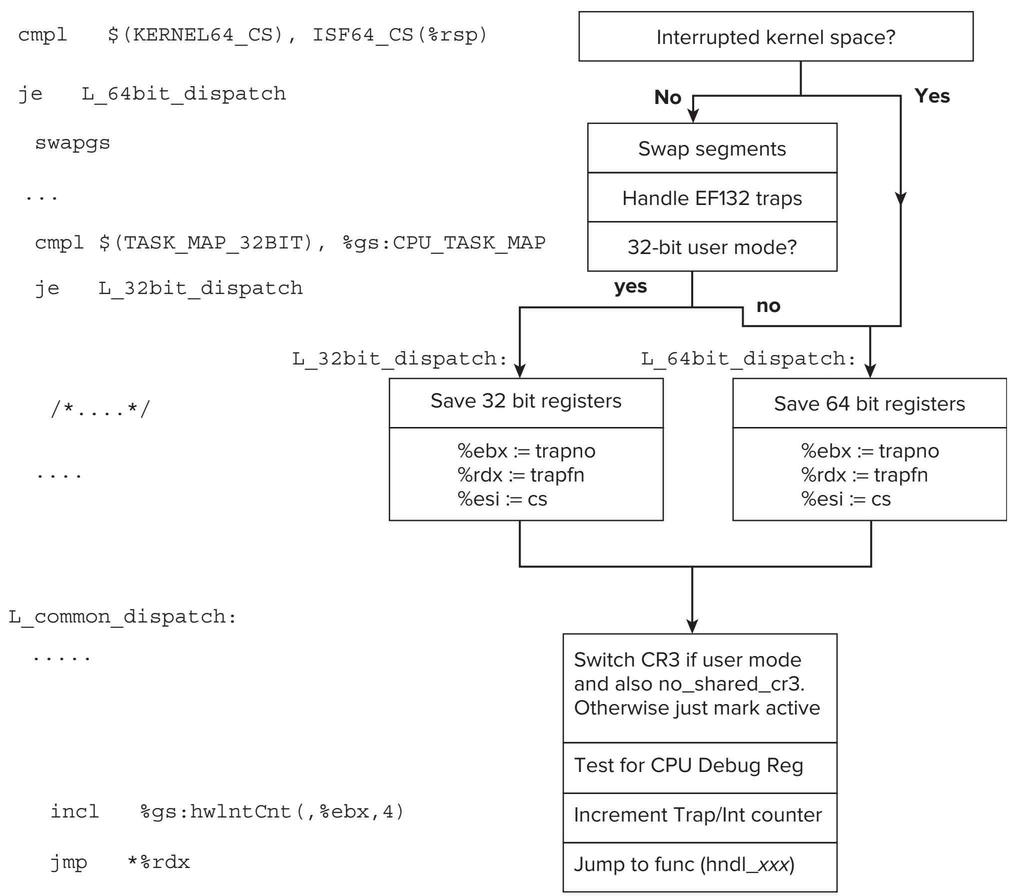


FIGURE 8-5: The common dispatcher

The last step in this flow is jumping to the handler function, which was defined on the stack (and loaded into RDX). In the case of a trap, this is `hndl_alltraps`, shown in Listing 8-3:

LISTING 8-3: hndl_alltraps, the common trap handler

```
Entry(hndl_alltraps)
        mov     %esi, %eax
        testb   $3, %al
        jz      trap_from_kernel

        TIME_TRAP_UENTRY

        movq    %gs:CPU_ACTIVE_THREAD,%rdi
        movq    %rsp, ACT_PCB_ISS(%rdi)        /* stash the PCB stack */
        movq    %rsp, %rdi                     /* also pass it as arg0 */
        movq    %gs:CPU_KERNEL_STACK,%rsp      /* switch to kernel stack */
```

continues

LISTING 8-3 *(continued)*

```
        sti

        CCALL(user_trap)                     /* call user trap routine */

         // user_trap is very likely to generate a Mach exception, and NOT return
         // (it suspends the currently active thread). In some cases, however, it
         // does return, and execution falls through

       /* user_trap() unmasks interrupts */
        cli                                  /* hold off intrs - critical section */
        xorl    %ecx, %ecx                   /* don't check if we're in the PFZ */

      // Fall through to return_from_trap.
```

The `user_trap` function, implemented in `i386/trap.c`, handles the actual traps. This is a C function, and the `CCALL` family of macros, defined in `idt64.s`, bridge from assembly to C by setting up the arguments on the stack. The `user_trap` function handles traps with specific handlers, or generates a generic exception — by calling `i386_exception` — which, in turn, usually converts it to a Mach exception, by calling `exception_triage`. Mach exceptions are covered in detail in Chapter 11, "Mach Scheduling." At this point, however, the important point is that `exception_triage` does not return, effectively ending the code path.

Interrupts are handled in a similar way to traps, only with `hndl_allintrs`, instead:

```
#define INTERRUPT(n)                             \
        Entry(_intr_ ## n)                       ;\
        pushq   $0                               ;\
        IDT_ENTRY_WRAPPER(n, HNDL_ALLINTRS)
```

The resulting stack is very similar to the `TRAP` macro's stack, as shown in Figure 8-4. The only difference is that the handler is now `HNDL_ALLINTRS`, instead of `HNDL_ALLTRAPS`, where `HNDL_ALLINTRS` is defined as shown in Listing 8-4:

LISTING 8-4: hndl_allintrs, the common interrupt handler

```
#define HNDL_ALLINTRS           EXT(hndl_allintrs)
Entry(hndl_allintrs)
        /*
         * test whether already on interrupt stack
          */
       movq    %gs:CPU_INT_STACK_TOP,%rcx
       cmpq    %rsp,%rcx
       jb      1f
       leaq    -INTSTACK_SIZE(%rcx),%rdx
       cmpq    %rsp,%rdx
       jb      int_from_intstack
1:
       xchgq   %rcx,%rsp             /* switch to interrupt stack */
```

```
        mov     %cr0,%rax              /* get cr0 */
        orl     $(CR0_TS),%eax         /* or in TS bit */
        mov     %rax,%cr0              /* set cr0 */
        subq    $8, %rsp               /* for 16-byte stack alignment */
        pushq   %rcx                   /* save pointer to old stack */
        movq    %rcx,%gs:CPU_INT_STATE /* save intr state */

        TIME_INT_ENTRY                 /* do timing */

incl    %gs:CPU_PREEMPTION_LEVEL
        incl    %gs:CPU_INTERRUPT_LEVEL

        movq    %gs:CPU_INT_STATE, %rdi

        CCALL(interrupt)               /* call generic interrupt routine */

        cli                            /* just in case we returned with intrs
enabled */
        xor     %rax,%rax
        movq    %rax,%gs:CPU_INT_STATE /* clear intr state pointer */

 // Falls through to return_to_iret, which returns to user mode via an iret
instruction
```

In the above code, `Interrupt` (in `osfmk/i386/trap.c`) is the generic kernel interrupt handler. This goes on to direct interrupt handling to either `lapic_interrupt` (in `osfmk/i386/lapic.c`) or `PE_incoming_interrupt` (in `pexpert/i386/pe_interrupt.c`, part of the Platform Expert), which passes it to the any registered I/O Kit interrupt handler. I/O Kit is described in more detail in its own chapter.

Putting this all together, and picking up where Figure 8-5 left off, we have the rest of the flow depicted in Figure 8-6.

As you can see, the trap handling in the kernel is pretty complicated, even when somewhat simplified and broken down into separate figures. If that's not flabbergasting enough, consider this logic occurs *on every trap and interrupt*, which can sometimes amount to more than thousands of times per second!

Looking at the figure, you will note references to the Preemption Free Zone (PFZ), and Asynchronous Software Traps (ASTs). ASTs are a mechanism in XNU somewhat akin to Linux's software IRQs. These are emulated traps, used primarily by the task scheduler, but not while the code is in the PFZ, which is a special region of text wherein preemptions are disabled. Both are covered in more detail in Chapter 11, "Mach Scheduling."

Trap Handlers on ARM

The ARM architecture is much simpler than that of Intel. From the ARM perspective, any non-user mode is entered through an exception, or interrupt. System calls are thus invoked via a simulated interrupt, with the SVC instruction. SVC is an acronym for "SuperVisor Call," though its previous name — SWI, or SoftWare Interrupt, was more accurate: when this instruction is called, the CPU automatically transfers control to the machine's trap vector, wherein a pre-defined kernel instruction, usually a branch to some specific handler, awaits.

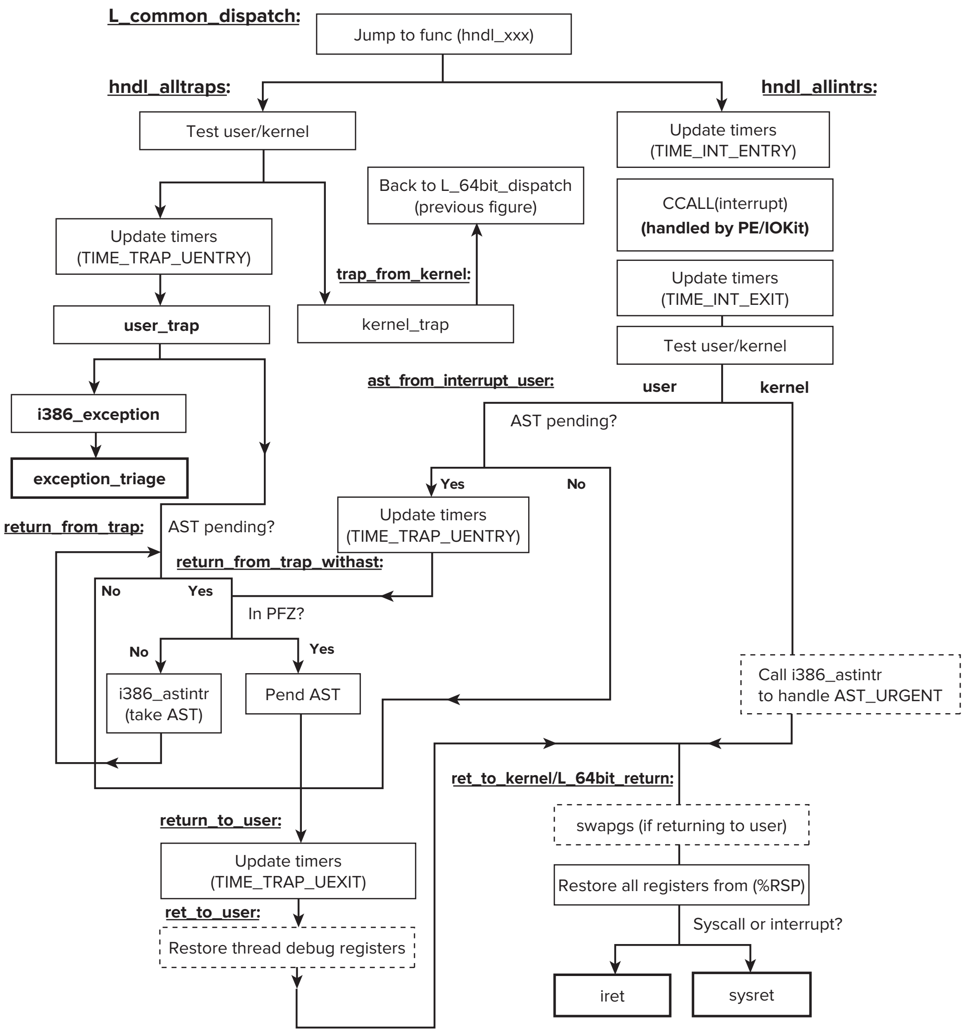


FIGURE 8-6: The common dispatcher, continued.

It is the kernel's responsibility to set up the trap handlers in ARM for all the modes the CPU can support. The iOS kernel does just that, by setting up an `ExceptionVectorsBase` as shown in Table 8-4:

TABLE 8-4: Registered trap handlers in iOS

OFFSET	EXCEPTION	HANDLED BY
`0x00`	Reset	`_fleh_reset`
`0x04`	Undefined Instruction	`_fleh_undef`
`0x08`	Software Interrupt	`_fleh_swi`

OFFSET	EXCEPTION	HANDLED BY
0x0C	Prefetch abort	_fleh_prefabt
0x10	Data abort	_feh_dataabt
0x14	Address exception	_fleh_addrexc
0x18	Interrupt Request	_fleh_irq
0x1c	Fast Interrupt Request	_fleh_fiq

These symbols were still visible (and even exported!) in the iOS *3.x* kernels, but have since been understandably removed in *4.x* and later. It remains, however, fairly easy to find them, as the following experiment shows.

Experiment: Finding the ARM trap handles in an iOS kernel

The `ExceptionVectorsBase` symbol is no longer exported, but — thanks to their unique structure of ARM handlers — it is trivial to find. The addresses of the trap handlers are loaded directly into the ARM Program Counter using an `LDR PC, [PC, #24]` command, which repeats seven times, for all handlers but FIQ, followed by a `MOV PC, R9` (where `_fleh_fiq` would be), the addresses themselves, and several NOPs (`0xE1A00000`). These commands are unique, so using `grep(1)` on their binary representation (or the string itself) quickly reveals them, as shown in Listing 8-5:

LISTING 8-5: Using otool(1) and grep(1) to find the ExceptionVectorsBase

```
morpheus@ergo (~)$ otool -tV ~/iOS/4.2.1.kernel.iPad1 | grep e59ff018
80064000       e59ff018       ldr    pc, [pc, #24] @ 0x80064020 ; points to fleh_reset
80064004       e59ff018       ldr    pc, [pc, #24] @ 0x80064024 ; points to fleh_undef
80064008       e59ff018       ldr    pc, [pc, #24] @ 0x80064028 ; points to fleh_swi
8006400c       e59ff018       ldr    pc, [pc, #24] @ 0x8006402c ; points to fleh_prefabt
80064010       e59ff018       ldr    pc, [pc, #24] @ 0x80064030 ; points to fleh_dataabt
80064014       e59ff018       ldr    pc, [pc, #24] @ 0x80064034 ; points to fleh_addrexc
80064018       e59ff018       ldr    pc, [pc, #24] @ 0x80064038 ; points to fleh_irq
morpheus@ergo (~)$ otool -tV ~/iOS/5.1.kernel.iPod4G | grep e59ff018
80078000       e59ff018       ldr    pc, [pc, #24] @ 0x80078020 ; points to fleh_reset
80078004       e59ff018       ldr    pc, [pc, #24] @ 0x80078024 ; points to fleh_undef
80078008       e59ff018       ldr    pc, [pc, #24] @ 0x80078028 ; points to fleh_swi
8007800c       e59ff018       ldr    pc, [pc, #24] @ 0x8007802c ; points to fleh_prefabt
80078010       e59ff018       ldr    pc, [pc, #24] @ 0x80078030 ; points to fleh_dataabt
80078014       e59ff018       ldr    pc, [pc, #24] @ 0x80078034 ; points to fleh_addrexc
80078018       e59ff018       ldr    pc, [pc, #24] @ 0x80078038 ; points to fleh_irq
```

The effect of directly loading an address into the program counter is tantamount to jumping to that address. These addresses are, in order, the address of the exception handlers shown previously in Table 8-4.

Using `otool(1)` once more, this time seeking to the address revealed by the `grep(1)` command, (continuing Listing 8-5) you reveal the actual addresses. The disassembly will be nonsensical — but you can clearly see the kernel-space addresses. Continuing the previous listing, Listing 8-6 examines the iOS 5.1 kernel:

LISTING 8-6: The Exception Vector addresses

```
8007801c        e1a0f009        mov     pc, r9
80078020        80078ff4        strdhi  r8, [r7], -r4      ; fleh_reset
80078024        80078ff8        strdhi  r8, [r7], -r8      ; fleh_undef
80078028        80079120        andhi   r9, r7, r0, lsr #2 ; fleh_swi
8007802c        80079370        andhi   r9, r7, r0, ror r3 ; fleh_prefabt
80078030        800794a4        andhi   r9, r7, r4, lsr #9 ; fleh_dataabt
80078034        80079678        andhi   r9, r7, r8, ror r6 ; fleh_addrexec
80078038        8007967c        andhi   r9, r7, ip, ror r6 ; fleh_irq
8007803c        8007983c        andhi   r9, r7, ip, lsr r8 ; ...
80078040        e1a00000        nop                     (mov r0,r0)
```

> *The* joker *tool, on the book's companion website, can be used for various educational tasks on the iOS kernel. It can automatically find the addresses of the ExceptionVectors in a decrypted kernel.*
>
> *You might want to also try the disassembly of iBoot, iBSS, and iBEC, as discussed in Chapter 6 "The OS X Boot Process". All the low-level components initialize the exception vectors in this way.*

The exception handlers can be disassembled in ARM mode. If you try to disassemble `fleh_reset`, for example, you'll reveal that it is effectively a halt instruction, jumping to itself in an endless loop. The most important of all the handlers is `fleh_swi`, which is the handler in charge of system calls—as those are triggered through the software interrupt mechanism. The code in it somewhat resembles the `hndl_syscall` code from the Intel XNU, discussed earlier, and is detailed later in the ARM subsection which follows.

Voluntary kernel transition

When user mode requires a kernel service, it issues a system call, which transfers control to the kernel. There are two ways of actually implementing a system call request. The first, by means of simulating an interrupt, is a legacy of the traditional Intel architecture, and is still used on ARM (by the `SVC`/`SWI` instruction). The second, using a dedicated instruction (Intel's `SYSENTER`/`SYSCALL`) is unique to Intel.

Simulated Interrupts

Any of the exceptions listed in Table 8-2 can be triggered by specifying their number as an argument to the `INTerrupt` command. This is also sometimes refers to as a *synchronous interrupt*, to distinguish it from a normal, unpredictable, and asynchronous interrupt.

For example, the debugger breakpoint operation is implemented on Intel architectures by the `INT 3` instruction. This instruction, which conveniently takes only one byte (opcode `0xCC`, with no operands), can be placed in memory by a debugger when the user specifies a breakpoint at some address. In this way, user mode can request a kernel service voluntarily — an exception is triggered, the CPU switches to privileged/supervisor mode, and the corresponding exception handler is automatically executed. The exception handler, set by the kernel, recognizes that this is a request, and can process specific arguments from the registers (The system call number is in `EAX`/`RAX` on Intel, `R12` on ARM).

Operating systems reserve a particular interrupt number for their own mechanism of entering kernel mode: DOS used `0x21`, NT through XP used `0x2E`, and most Intel UN*X-based systems used `0x80`. On Intel, this was also the mechanism used by OS X for system calls, and — although it has been largely deprecated in favor of `SYSCALL` (see the following section), there are still some traces of it.

SYSENTER/SYSCALL

Since user/kernel transition occurs so frequently, the Intel architecture introduced a more efficient instruction for it, called `SYSENTER`, beginning with the Pentium II architecture. In 64-bit architecture a slightly different instruction, `SYSCALL`, is used. Using these, rather than interrupt gates, is faster, as it employs a set of *model specific registers*, or MSRs. Rather than saving the key registers prior to entering kernel mode, and restoring them on exit, the MSRs allow the CPU to switch to the separate set on kernel mode, and back to the normal ones on user mode. `SYSENTER` or `SYSCALL` function similarly to a `CALL` instruction — though the instructions need not save the return address on the stack, since the User Mode Instruction Pointer will remain untouched. A corresponding call to `SYSEXIT` restores the user mode registers.

As the name implies, there are many model specific registers (and different processors have different sets). They are all defined in `proc_reg.h`, and the relevant ones for `SYSENTER` are shown in Table 8-5:

TABLE 8-5: Model-Specific Registers of the Intel Architecture

REGISTER #	#DEFINE	PURPOSE
`0x174`	`MSR_IA32_SYSENTER_CS`	Code Segment
`0x175`	`MSR_IA32_SYSENTER_ESP`	Stack Pointer — set by kernel to kernel stack
`0x176`	`MSR_IA32_SYSENTER_EIP`	Instruction Pointer — set to kernel entry point
`0xC0000081`	`MSR_IA32_STAR`	Contains base selector for `SYSCALL`/`SYSRET`, `CS`/`SS`, and EIP
`0xC0000082`	`MSR_IA32_LSTAR`	Contains `SYSCALL` entry point

During the boot process the kernel initializes the MSRs. The initialization is performed by `cpu_mode_init()` (called from `vstart()`, as discussed in the next chapter). The `cpu_mode_init()` function calls `wrmsr64` — which is a C wrapper to an identical assembly routine. The function loads the three model specific registers with the values, which will be used for the kernel stack and code. This is shown in Listing 8-7:

LISTING 8-7: Setting MSRs for SYSENTER and SYSCALL (osfmk/i386/mp_desc.c)

```
/*
 * Set MSRs for sysenter/sysexit and syscall/sysret for 64-bit.
 */
static void
fast_syscall_init64(__unused cpu_data_t *cdp)
{
       // Registers used for SYSENTER (32-bit mode on 64-bit architecture)
```

continues

LISTING 8-7 *(continued)*

```
        wrmsr64(MSR_IA32_SYSENTER_CS, SYSENTER_CS);
        wrmsr64(MSR_IA32_SYSENTER_EIP, UBER64((uintptr_t) hi64_sysenter));
        wrmsr64(MSR_IA32_SYSENTER_ESP, UBER64(current_sstk()));

        /* Enable syscall/sysret */
        wrmsr64(MSR_IA32_EFER, rdmsr64(MSR_IA32_EFER) | MSR_IA32_EFER_SCE);


        /*
         * MSRs for 64-bit syscall/sysret
         * Note USER_CS because sysret uses this + 16 when returning to
         * 64-bit code.
         */
        wrmsr64(MSR_IA32_LSTAR, UBER64((uintptr_t) hi64_syscall));
        wrmsr64(MSR_IA32_STAR, (((uint64_t)USER_CS) << 48) |
                                (((uint64_t)KERNEL64_CS) << 32));

        ...
}
```

The entry point `hi64_sysenter` defined in `idt64.s`, is used for 32-bit `sysenter` compatibility. It switches to kernel mode, and invokes, through the common handler shown in Figure 8-5, the generic `hndl_sysenter`, to invoke the system call (the flow merges with the common handler in `L_32bit_dispatch`). This handler, in turn, tests the system-call type, treating it as a 32-bit value, with Mach calls as negative. A similar implementation is in `hi64_syscall`, which is invoked for 64-bit syscall instructions, and calls on `HNDL_SYSCALL`, as shown in Listing 8-8:

LISTING 8-8: The idt64/hi64_syscall entry point

```
Entry(hi64_syscall)
Entry(idt64_syscall)
        swapgs                          /* Kapow! get per-cpu data area */
L_syscall_continue:
        mov     %rsp, %gs:CPU_UBER_TMP  /* save user stack */
        mov     %gs:CPU_UBER_ISF, %rsp  /* switch stack to pcb */
          ..
leaq    HNDL_SYSCALL(%rip), %r11;
        movq    %r11, ISF64_TRAPFN(%rsp)
        jmp     L_64bit_dispatch        /* this can only be a 64-bit task */
```

Voluntary kernel transition on ARM

The ARM architecture has no dedicated system-call instructor, and still uses the system-call gate technique. The kernel, when loaded, overwrites all the trap handlers (as shown in Table 8-4), of which the Software Interrupt (SWI) handler is one. When the ARM assembly instruction of SVC is executed in user mode, control is transferred immediately to the handler, `fleh_swi`, and the CPU enters kernel mode.

The `fleh_swi` handler (whose address was found in the previous experiment) is highly optimized, but still displays the basic structure shared by the Intel version of XNU. This is shown in Listing 8-9. If your ARM assembly isn't what it used to be — you can just read through the comments:

LISTING 8-9: The SWI handler from iOS 5.0 and 5.1, iPod4,1 kernel

```
0x80079120 _fleh_swi
  text:80079120   CMN     R12, #3
__text:80079124   BEQ     loc_80079344  ; Branches off to ml_get_timebase if R12==3
;
; Largely irrelevant ARM Assembly omitted for brevity
; jumps to another section of the function which handles Machine Dependent calls
;
; What is relevant: R11 holds the system call number
;
__text:80079184   BLX     get_BSD_proc_and_thread_and_do_kauth
 ;
 ; Set R9 to the privileged only Thread and Process ID Register
 ; We need this for UNIX system calls, later
 ;
__text:80079188   MRC     p15, 0, R9,c13,c0, 4
 ;
 ; Remember that Mach calls are negative. The following separates Mach from UNIX
 ;
__text:8007918C   RSBS    R5, R11, #0 ; Reverse substract with carry
__text:80079190   BLE     _is_unix
 ;
 ; Fall through on Mach. This is what in Intel would be a call to mach_munger
 ; but on ARM just directly gets the Mach trap
 ;
 ; KERNEL_DEBUG_CONSTANT(MACHDBG_CODE(DBG_MACH_EXCP_SC,
 ; (call_number)) | DBG_FUNC_START);
 ;
__text:80079194   LDR     R4, =_kdebug_enable ; recall kdebug was discussed in Ch. 5
__text:80079198   LDR     R4, [R4]
__text:8007919C   MOVS    R4, R4  ; test kdebug_enable
__text:800791A0   MOVNE   R0, R8
__text:800791A4   MOVNE   R1, R5
__text:800791A8   BLNE    ____kernel_debug_mach_func_entry
__text:800791AC   ADR     LR, _return_from_swi   ; Set our return on error
   ;
   ; Increment Mach trap count (at offset 0x1B4 of thread structure)
   ;
__text:800791B0   LDR    R2, [R10,#0x1B4] ; get Mach trap count
__text:800791B4   CMP    R5, #128         ; Compare Mach trap to MACH_TRAP_TABLE_COUNT
__text:800791B8   ADD    R2, R2, #1       ; increment Mach trap count
__text:800791BC   STR    R2, [R10,#0x1B4] ; and store
__text:800791C0   BGE    do_arm_exception ; if syscall number > MACH_TRAP_TABLE_COUNT...
   ;
   ; If we are here, R5 holds the Mach trap number - dereference from mach_trap_table:
   ; R1 = mach_trap_table[call_number].mach_trap_function
   ;
__text:800791C4   LDR    R1, =_mach_trap_table
__text:800791C8   ADD    R1, R1, R5,LSL#3  ; R1 = R1 + call_num * sizeof(mach_trap_t)
__text:800791CC   LDR    R1, [R1,#4]       ; +4, skip over arg_count
   ;
   ; if (mach_call == (mach_call_t)kern_invalid)
   ;
__text:800791D0   LDR    R2, =(_kern_invalid+1)
```

continues

LISTING 8-9 *(continued)*

```
__text:800791D4   MOV    R0, R8
__text:800791D8   TEQ    R1, R2
__text:800791DC   BEQ    do_arm_exception
   ;
   ; else just call trap from R1
   ;
__text:800791E0   BX      R1      ;  Do Mach trap (jump to table pointer)
   ; returning from trap
__text:800791E4   STR     R1, [R8,#4]
return_from_swi
__text:800791E8   STR     R0, [R8]
__text:800791EC   MOVS    R4, R4
__text:800791F0   MOVNE   R1, R5
 ;
 ; KERNEL_DEBUG_CONSTANT(MACHDBG_CODE(DBG_MACH_EXCP_SC, (call_number)) | DBG_FUNC_END);
 ;
__text:800791F4   BLNE    ____kernel_debug_mach_func_exit
 ;
 ; iOS's load_and_go_user is like OS X's thread_exception_return();
 ;
__text:800791F8   BL      __load_and_go_user
__text:800791FC   B       loc_800791FC       ; HANG ENDLESSLY - Not Reached
 ;
 ; arm_exception(EXC_SYSCALL,call_number, 1);
 ;
do_arm_exception:  ; Generates a Mach exception (discussed in Chapter 10)
__text:80079200   MOV     R0, #EXC_SYSCALL
__text:80079204   SUB     R1, SP, #4
__text:80079208   MOV     R2, #1
__text:8007920C   BLX     _exception_triage  ; as i386_exception, direct fall through
__text:80079210   B       loc_80079210       ; HANG ENDLESSLY - Not reached
 ;
 ; For UNIX System calls:
 ;
_is_unix
 ;
 ; Increment UNIX system call count for this thread
 ; (at offset 0x1B8 of thread structure)
 ;
__text:80079220   LDR     R1, [R10,#0x1B8]
__text:80079224   MOV     R0, R8           ; out of order: 1st argument of unix_syscall
__text:80079228   ADD     R1, R1, #1
__text:8007922C   STR     R1, [R10,#0x1B8]
 ;
 ;
 ;
__text:80079230   MOV     R1, R9           ; 2nd argument of unix_syscall
__text:80079234   LDR     R2, [R9,#0x5BC]  ; 3rd argument of unix_syscall
__text:80079238   LDR     R3, [R10,#0x1EC] ; 4th argument of unix_syscall
 ;
 ; Call _unix_syscall
 ;
__text:8007923C   BL      _unix_syscall
__text:80079240   B    loc_80079240          ; HANG ENDLESSLY - Not reached
```

SYSTEM CALL PROCESSING

Most people are familiar with POSIX system calls. In XNU, however, the POSIX system calls make up only one of four possible system call classes, as shown in Table 8-6:

TABLE 8-6: XNU system call classes

SYSCALL_CLASS	HANDLED BY	ENCOMPASSES
`UNIX (1)`	`unix_syscall[64]` (`bsd/dev/i386/systemcalls.c`)	POSIX/BSD system calls: the "classic" system calls, interfacing with XNU's BSD APIs.
`MACH (2)`	`mach_call_munger[64]` (`osfmk/i386/bsd_i386.c`)	Mach traps: calls that interface directly with the Mach core of XNU.
`MDEP (3)`	`machdep_syscall[64]` (`osfmk/i386/bsd_i386.c`)	Machine dependent calls: used for processor specific features.
`DIAG (4)`	`diagCall[64]` (`osfmk/i386/Diagnostics.c )`	Diagnostic calls: used for low-level kernel diagnostics. Enabled by the `diag` boot argument.

In 32-bit architectures, the UNIX system calls are positive, whereas the Mach traps are negative. In 64-bit, all call types are positive, but the most significant byte contains the value of `SYSCALL_CLASS` from the preceding table. The value is checked by shifting the system call number `SYSCALL_CLASS_SHIFT (=24)` bits, as you can see in Listing 8-10:

LISTING 8-10: The XNU 64-bit common system call handler

```
Entry(hndl_syscall)
        TIME_TRAP_UENTRY

        movq    %gs:CPU_KERNEL_STACK,%rdi
        xchgq   %rdi,%rsp                     /* switch to kernel stack */
        movq    %gs:CPU_ACTIVE_THREAD,%rcx    /* get current thread     */
        movq    %rdi, ACT_PCB_ISS(%rcx)
        movq    ACT_TASK(%rcx),%rbx           /* point to current task  */

        /* Check for active vtimers in the current task */
        TASK_VTIMER_CHECK(%rbx,%rcx)

        /*
         * We can be here either for a mach, unix machdep or diag syscall,
         * as indicated by the syscall class:
         */
        movl    R64_RAX(%rdi), %eax           /* syscall number/class */
        movl    %eax, %edx
        andl    $(SYSCALL_CLASS_MASK), %edx   /* syscall class */
        cmpl    $(SYSCALL_CLASS_MACH<<SYSCALL_CLASS_SHIFT), %edx
        je      EXT(hndl_mach_scall64)
```

continues

LISTING 8-10 *(continued)*

```
cmpl    $(SYSCALL_CLASS_UNIX<<SYSCALL_CLASS_SHIFT), %edx
je      EXT(hndl_unix_scall64)
cmpl    $(SYSCALL_CLASS_MDEP<<SYSCALL_CLASS_SHIFT), %edx
je      EXT(hndl_mdep_scall64)
cmpl    $(SYSCALL_CLASS_DIAG<<SYSCALL_CLASS_SHIFT), %edx
je      EXT(hndl_diag_scall64)

/* Syscall class unknown */
CCALL3(i386_exception, $(EXC_SYSCALL), %rax, $1)
```

All handlers are prototyped in the same way — as C functions which take one argument, which is a pointer to an architecture specific saved state, which is really nothing more than a structure containing a dump of all the processor registers. In OS X, this is an `x86_saved_state_t` (defined in `osfmk/mach/i386/thread_status.h`), which holds (as a union) either a 32-bit or a 64-bit state. The kernel sources leak an `arm_saved_state_t` as well.

The handlers are expected to never return. Indeed, on OS X all of the handlers end by calling `thread_exception_return()`(defined in `osfmk/x86_64/locore.s`, which falls through to `return_from_trap()`, as discussed earlier in this chapter. In iOS, `load_and_go_user()` is used instead, and returns to user mode by restoring the CPSR to user.

POSIX/BSD System calls

The main personality exposed by XNU is that of POSIX/BSD. These are internally referred to as "UNIX system calls" or "BSD calls," even though they contain quite a few Apple-specific calls.

unix_syscall

The BSD system call handler has a straightforward implementation. Both 32- and 64-bit handlers (in `bsd/dev/i386/systemcalls.c`) get the saved state as an argument and operate in the same manner, namely:

1. Make sure the saved state matches the architecture.
2. Get the BSD process structure from the `current_task`. Make sure that the BSD process actually exists.
3. If a `syscall` number is 0, it is an indirect system call. Fix arguments accordingly.
4. Arguments are expected to be passed as 64-bit values. For 64-bit handler, this only requires work if they cannot all be passed in registers (i.e. cases where there are more than six arguments). The remaining arguments then need to be copied onto the stack. In the 32-bit handler, arguments need to be "munged." Munging refers to the process of copying the arguments from user mode, while addressing 32/64-bit compatibility.
5. Execute system calls from the `sysent` table. All system calls are executed in the same way.

   To notify the auditing subsystem of the call:

   ```
   AUDIT_SYSCALL_ENTER(code, p, uthread);
   ```

To actually execute the call:

```
error = (*(callp->sy_call))((void *) p, uargp, &(uthread->uu_rval[0]));
```

To notify the auditing system of the call exit:

```
AUDIT_SYSCALL_EXIT(code, p, uthread, error);
```

In other words, `syscall`s are subject to auditing and are all called with the first argument being the `current_proc()`.

6. In rare cases, the system call might indicate it needs to be restarted, which is handled by `pal_syscall_restart()`.
7. The "error" (the system call return code) is handled to fit in the return register (for Intel this is EAX/RAX, and for ARM it's R0).
8. The system call returns through `thread_exception_return()` (for iOS, `load_and_go_user`), which is the same handling as `return_from_trap()`, taking any ASTs along the way.

sysent

BSD system calls are maintained in the `sysent` table. This table is an array of similarly-named structures and is defined in `bsd/sys/sysent.h` as shown in Listing 8-11:

LISTING 8-11: The sysent table

```
struct sysent {                   /* system call table */
      int16_t     sy_narg;        /* number of args */
      int8_t      sy_resv;        /* reserved  */
      int8_t      sy_flags;       /* flags */
      sy_call_t  *sy_call;        /* implementing function */
      sy_munge_t *sy_arg_munge32; /* system call arguments munger for 32-bit
                                  process */
      sy_munge_t *sy_arg_munge64; /* system call arguments munger for 64-bit
                                  process */
      int32_t     sy_return_type; /* system call return types */
      uint16_t    sy_arg_bytes;   /* Total size of arguments in bytes for
                                   * 32-bit system calls
                                   */
};

#ifndef __INIT_SYSENT_C__
extern struct sysent sysent[];
#endif  /* __INIT_SYSENT_C__ */

extern int nsysent;
#define NUM_SYSENT      439     // # of syscalls (+1) in Lion. (SL: 434, ML: 440, iOS5: 439)
```

The `sysent` table is populated during compile time by a shell script, `bsd/kern/makesyscalls.sh`, which is invoked during the building of the kernel. This script parses the system call template file, `bsd/kern/syscalls.master`, wherein all the system calls are defined, as shown in Listing 8-12.

LISTING 8-12: The bsd/kern/syscalls.master file

```
#include <sys/param.h>
#include <sys/systm.h>
#include <sys/types.h>
#include <sys/sysent.h>
#include <sys/sysproto.h>

0       AUE_NULL        ALL     { int nosys(void); }   { indirect syscall }
1       AUE_EXIT        ALL     { void exit(int rval) NO_SYSCALL_STUB; }
2       AUE_FORK        ALL     { int fork(void) NO_SYSCALL_STUB; }
3       AUE_NULL        ALL     { user_ssize_t read(int fd, user_addr_t cbuf, user_size
_t nbyte); }
4       AUE_NULL        ALL     { user_ssize_t write(int fd, user_addr_t cbuf, user_size
_t nbyte); }
5       AUE_OPEN_RWTC   ALL     { int open(user_addr_t path, int flags, int mode) NO
_SYSCALL_STUB; }
...
... // many more system calls omitted here
... //
433     AUE_NULL        ALL     { int pid_suspend(int pid); }
434     AUE_NULL        ALL     { int pid_resume(int pid); }
#if CONFIG_EMBEDDED
435     AUE_NULL        ALL     { int pid_hibernate(int pid); }
436     AUE_NULL        ALL     { int pid_shutdown_sockets(int pid, int level); }
#else
435     AUE_NULL        ALL     { int nosys(void); }
436     AUE_NULL        ALL     { int nosys(void); }
#endif
437     AUE_NULL        ALL     { int nosys(void); } { old shared_region_slide_np }
438     AUE_NULL        ALL     { int shared_region_map_and_slide_np(int fd, uint32_t
count, const struct shared_file_mapping_np *mappings, uint32_t slide, uint64_t*
slide_start, uint32_t slide_size) NO_SYSCALL_STUB; }

   // Mountain Lion also contains 439 - kas_info
```

The system call table whets the appetite of many a hacker (and security researcher alike), because intercepting system calls means complete control of user mode. As a result, the symbol is no longer exported, not on OS X and certainly not on iOS. A common technique suggested by Stefan Esser[3] relies on the table being in close proximity to the `kdebug` public symbol. A more reliable technique, however, can quickly reveal the `sysent` structure's unique signature even in a binary dump with no symbols. The *joker* tool, available on the book's companion website, was written especially for this purpose, and zeroes in on the signature shown in Listing 8-13. The signature is actually the same for OS X and iOS, with only minor modifications for `sizeof(void *)` between 32- and 64-bit (and, of course, the system call addresses themselves).

LISTING 8-13: A disassembly of an iOS 5.1 kernel, showing the system call table

```
802CCBAC_sysent DCD 0                   ; Called from unix_syscall+C4
...
802CCBC4        DCW 1                   ; int16_t    sy_narg; (exit has one argument)
```

```
802CCBC6        DCB 0                   ; int8_t     sy_resv;
802CCBC7        DCB 0                   ; int8_t     sy_flags;
802CCBC8        DCD_exit+1              ; sy_call_t *sy_call = exit(int);
802CCBCC        DCD 0                   ; sy_munge_t *sy_arg_munge32;
802CCBD0        DCD 0                   ; sy_munge_t *sy_arg_munge64;
802CCBD4        DCD SYSCALL_RET_NONE    ; int32_t    sy_return_type; (0 = void)
802CCBD8        DCW 4                   ; uint16_t   sy_arg_bytes; (1 arg = 4 bytes)
802CCBDA        DCW 0                   ; Padding to 32-bit boundary
              ; ------------------------------------------
802CCBDC        DCW 0                   ; int16_t    sy_narg; (fork has no arguments)
802CCBDE        DCB 0                   ; int8_t     sy_resv;
802CCBDD        DCW 0                   ; int8_t     sy_flags;
802CCBE0        DCD fork+1              ; sy_call_t *sy_call = pid_t fork();
802CCBE4        DCD 0                   ; sy_munge_t *sy_arg_munge32;
802CCBE8        DCD 0                   ; sy_munge_t *sy_arg_munge64;
802CCBEC        DCD SYSCALL_RET_INT_T   ; int32_t    sy_return_type; (pid_t is an int)
802CCBF0        DCW 0                   ; uint16_t   sy_arg_bytes; (fork has none)
802CCBF2        DCW 0                   ; Padding to 32-bit boundary
               ;----------------------------------------
802CCBF4        DCB 3                   ; int8_t     sy_narg; (read(2) has three args)
802CCBF5        DCB 0                   ; int8_t     sy_flags;
802CCBF6        DCW 0                   ; padding to 32-bit boundary
802CCBF8        DCD _read+1             ; sy_call_t *sy_call = read(int,void *, size_t);
802CCBFC        DCD 0                   ; sy_munge_t *sy_arg_munge32;
802CCC00        DCD 0                   ; sy_munge_t *sy_arg_munge64;
802CCC04        DCD SYSCALL_RET_SSIZE_T; int32_t     sy_return_type;
802CCC08        DCW 0xC                 ; uint16_t    sy_arg_bytes; (3 args = 12 bytes)
.. //
.. // and on, and on , and on…
.. //
802CF4D4_nsysent DCD 0x1B7              ; NUM_SYSENT
```

The system calls are also generated with their names hard-coded into the binary. In OS X that doesn't make too much of a difference, but in iOS this feature is quite useful. iOS's system calls are largely the same as those of OS X, with a few notable exceptions (for example, the "ledger" system call, #373, unavailable on OS X prior to Mountain Lion, and the `pid_shutdown_sockets` system call). A more detailed discussion of the specific system calls can be found in the online appendix.

Mach Traps

If the system call number is negative (on 32-bit OS X or iOS) or contains the Mach class (64-bit), the kernel flow is diverted to handling Mach traps, rather than BSD system calls. The handler for Mach traps is called `mach_call munger[64]`.

mach_call_munger

Mach traps are processed by `mach_call_munger[64]`, which is implemented (on OS X) in `osfmk/i386/bsd_i386.c`. The term "munging" dates back to the days when function arguments needed to be undergo internal type-casting and alignment from the stack, to a structure of 64-bit integers. Both UNIX and Mach call arguments needed munging, and the 32-bit `unix_syscall` still contains munging code.

Munging is no longer necessary in x86_64, because the AMD-64 ABI uses six registers directly. The only case where munging would required is if a function has more than six arguments (which is seldom, if ever). In the 32-bit version of the handler, a helper function `mach_call_munger32` is called which copies the arguments and aligns them in a `mach_call_args` structure. Listing 8-14 shows the 64-bit version, annotated and noting where 32-bit would differ:

LISTING 8-14: mach_call_munger64, from osfmk/i386/bsd_i386.c

```
void
mach_call_munger64(x86_saved_state_t *state)
{
        int call_number;
        int argc;
        mach_call_t mach_call;
        x86_saved_state64_t     *regs;

        assert(is_saved_state64(state));
        regs = saved_state64(state);

        // In mach_call_munger (the 32-bit version), the call_number is obtained
        // by: call_number = -(regs->eax);
        call_number = (int)(regs->rax & SYSCALL_NUMBER_MASK);


DEBUG_KPRINT_SYSCALL_MACH(
                "mach_call_munger64: code=%d(%s)\n",
                call_number, mach_syscall_name_table[call_number]);

        // Kdebug trace of function entry (see chapter 5)
        KERNEL_DEBUG_CONSTANT(MACHDBG_CODE(DBG_MACH_EXCP_SC,
                                           (call_number)) | DBG_FUNC_START,
                              regs->rdi, regs->rsi,
                              regs->rdx, regs->r10, 0);

        // if this is an obviously invalid call, raise syscall exception
        if (call_number < 0 || call_number >= mach_trap_count) {
                i386_exception(EXC_SYSCALL, regs->rax, 1);
                /* NOTREACHED */
        }
       // Get entry from mach_trap_table. We need the entry to validate the call
       // is a valid one, as well as get the number of arguments
        mach_call = (mach_call_t)mach_trap_table[call_number].mach_trap_function;

       // Quite a few entries in the table are marked as invalid, for deprecated calls.
       // If we stumbled upon one of those, generate an exception

        if (mach_call == (mach_call_t)kern_invalid) {
                i386_exception(EXC_SYSCALL, regs->rax, 1);
                /* NOTREACHED */
        }

        argc = mach_trap_table[call_number].mach_trap_arg_count;
```

```
        // In 32-bit, we would need to prepare the arguments, copying them from
        // the stack to a mach_call_args struct. This is where we would need to
        // call a helper, mach_call_arg_munger32:
        //  if (argc)
        //   retval = mach_call_arg_munger32(regs->uesp, argc, call_number, &args);
        //
        //  In 64-bit, up to six arguments may be directly passed in registers,
        //  so the following code is only necessary for cases of more than 6
         if (argc > 6) {
                int copyin_count;
                copyin_count = (argc - 6) * (int)sizeof(uint64_t);

                if (copyin((user_addr_t)(regs->isf.rsp + sizeof(user_addr_t)), (char
*)&regs->v_arg6, copyin_count)) {
                        regs->rax = KERN_INVALID_ARGUMENT;

                        thread_exception_return();
                        /* NOTREACHED */
                }
        }

                if (retval != KERN_SUCCESS) {
                        regs->eax = retval;

                        DEBUG_KPRINT_SYSCALL_MACH(
                                "mach_call_munger: retval=0x%x\n", retval);

                        thread_exception_return();
                        /* NOTREACHED */
                }
        }
        // Execute the call, collect return value straight into RAX
        regs->rax = (uint64_t)mach_call((void *)(&regs->rdi));

        DEBUG_KPRINT_SYSCALL_MACH( "mach_call_munger64: retval=0x%llx\n", regs->rax);


        // Kdebug trace of function exit (see chapter 5)

        KERNEL_DEBUG_CONSTANT(MACHDBG_CODE(DBG_MACH_EXCP_SC,
                                           (call_number)) | DBG_FUNC_END,
                              regs->rax, 0, 0, 0, 0);

        throttle_lowpri_io(TRUE);

        // return to user mode
        thread_exception_return();
        /* NOTREACHED */
}
```

Note how similar this code is to the disassembly of `fleh_swi` shown earlier in Listing 8-9: even though iOS doesn't use a munger, the sanity checks and Mach trap `kdebug` traces are the same.

mach_trap_table

The `mach_trap_table`, an array of `mach_trap_t` structures, can be found in `osfmk/kern/syscall_sw.c`, where it is followed by the corresponding names, in `mach_syscall_name_table`, as shown in Listing 8-15:

LISTING 8-15: The Mach trap table and syscall_name_table (osfmk/kern/syscall_sw.c)

```
mach_trap_t     mach_trap_table[MACH_TRAP_TABLE_COUNT] = {
/* 0 */         MACH_TRAP(kern_invalid, 0, NULL, NULL),
// many invalid traps…
/* 26 */        MACH_TRAP(mach_reply_port, 0, NULL, NULL),
/* 27 */        MACH_TRAP(thread_self_trap, 0, NULL, NULL),
/* 28 */        MACH_TRAP(task_self_trap, 0, NULL, NULL),
/* 29 */        MACH_TRAP(host_self_trap, 0, NULL, NULL),
// many more traps, most invalid..
/* 127 */       MACH_TRAP(kern_invalid, 0, NULL, NULL),
};

const char * mach_syscall_name_table[MACH_TRAP_TABLE_COUNT] = {
/* 0 */         "kern_invalid",
..
/* 26 */        "mach_reply_port",
/* 27 */        "thread_self_trap",
/* 28 */        "task_self_trap",
/* 29 */        "host_self_trap",
..
/* 127 */       "kern_invalid",
};

int     mach_trap_count = (sizeof(mach_trap_table) / sizeof(mach_trap_table[0]));

kern_return_t kern_invalid(
        __unused struct kern_invalid_args *args)
{
        if (kern_invalid_debug) Debugger("kern_invalid mach trap");
        return(KERN_INVALID_ARGUMENT);
}
```

Most Mach traps are unused, funneled to `kern_invalid()`, which returns `KERN_INVALID_ARGUMENT` to the caller. Those Mach traps that are of some use are discussed in the online appendix. Finding the unexported table in the iOS binary can be accomplished reliably (and just as easily as finding `sysent)` by looking for its distinct signature (a sequence of `kern_invalid` and NULLs), or by following the reference from `fleh_swi`. The joker tool, from the book's companion website, does just that.

Mach traps are not likely to be deprecated any time soon. In fact, Apple seems to be *adding* more traps on occasion. One recent such addition in iOS 5.*x* was the family of `kernelrpc_*` calls (10–23), which will likely make their way into OS X in Mountain Lion. Output 8-1 shows the address of the defined Mach traps on an iOS 5.1 kernel (those not listed are all `kern_invalid`), as displayed by the *joker* tool:

OUTPUT 8-1: Mach traps (and their names) on iOS 5.1

```
 10 _kernelrpc_mach_vm_allocate_trap         800132ac
 11 _kernelrpc_vm_allocate_trap              80013318
 12 _kernelrpc_mach_vm_deallocate_trap       800133b4
 13 _kernelrpc_vm_deallocate_trap            80013374
 14 _kernelrpc_mach_vm_protect_trap          8001343c
 15 _kernelrpc_vm_protect_trap               800133f8
 16 _kernelrpc_mach_port_allocate_trap       80013494
 17 _kernelrpc_mach_port_destroy_trap        800134e4
 18 _kernelrpc_mach_port_deallocate_trap     80013520
 19 _kernelrpc_mach_port_mod_refs_trap       8001355c
 20 _kernelrpc_mach_port_move_member_trap    8001359c
 21 _kernelrpc_mach_port_insert_right_trap   800135e0
 22 _kernelrpc_mach_port_insert_member_trap  8001363c
 23 _kernelrpc_mach_port_extract_member_trap 80013680
 26 mach_reply_port                          800198ac
 27 thread_self_trap                         80019890
 28 task_self_trap                           80019870
 29 host_self_trap                           80017db8
 31 mach_msg_trap                            80013c1c
 32 mach_msg_overwrite_trap                  80013ae4
 33 semaphore_signal_trap                    800252d4
 34 semaphore_signal_all_trap                80025354
 35 semaphore_signal_thread_trap             80025260
 36 semaphore_wait_trap                      800255e8
 37 semaphore_wait_signal_trap               8002578c
 38 semaphore_timedwait_trap                 800256c8
 39 semaphore_timedwait_signal_trap          8002586c
 43 map_fd                                   80025f50
 44 task_name_for_pid                        801e0734
 45 task_for_pid                             801e0598
 46 pid_for_task                             801e054c
 48 macx_swapon                              801e127c
 49 macx_swapoff                             801e14cc
 50 kern_invalid                             80025f50
 51 macx_triggers                            801e1260
 52 macx_backing_store_suspend               801e11f0
 53 macx_backing_store_recovery              801e1198
 58 pfz_exit                                 80025944
 59 swtch_pri                                800259f4
 60 swtch                                    80025948
 61 thread_switch                            80025bb8
 62 clock_sleep_trap                         800160f0
 89 mach_timebase_info_trap                  80015318
 90 mach_wait_until_trap                     80015934
 91 mk_timer_create_trap                     8001d238
 92 mk_timer_destroy_trap                    8001d428
 93 mk_timer_arm_trap                        8001d46c
 94 mk_timer_cancel_trap                     8001d4f0
100 iokit_user_client_trap (probably)        80234aa0
```

A more detailed discussion of the specific traps can be found in the online appendix.

Machine Dependent Calls

Besides Mach traps and UNIX system calls, XNU contains machine dependent calls. As the name implies, these vary by platform. These calls in OS X are open source, but remain undocumented in iOS. Binary inspection confirms that, indeed, these calls exist. True to their machine-specific nature, they mostly offer functionality pertaining to the CPU caches (e.g. invalidating the MMU instruction and data caches).

machdep_call _table

The machine dependent calls have their own dispatch table — `machdep_call_table`, defined in `osfmk/i386/machdep_call.c` in a similar manner to the Mach trap table, and shown in Listing 8-16:

LISTING 8-16: Machine dependent calls, from osfmk/i386/machdep_call.c

```
machdep_call_t          machdep_call_table[] = {
        MACHDEP_CALL_ROUTINE(kern_invalid,0),
        MACHDEP_CALL_ROUTINE(kern_invalid,0),
        MACHDEP_CALL_ROUTINE(kern_invalid,0),
        MACHDEP_CALL_ROUTINE(thread_fast_set_cthread_self,1),
        MACHDEP_CALL_ROUTINE(thread_set_user_ldt,3),
        MACHDEP_BSD_CALL_ROUTINE(i386_set_ldt,3),
        MACHDEP_BSD_CALL_ROUTINE(i386_get_ldt,3),
};
machdep_call_t          machdep_call_table64[] = {
        MACHDEP_CALL_ROUTINE(kern_invalid,0),
        MACHDEP_CALL_ROUTINE(kern_invalid,0),
        MACHDEP_CALL_ROUTINE(kern_invalid,0),
        MACHDEP_CALL_ROUTINE64(thread_fast_set_cthread_self64,1),
        MACHDEP_CALL_ROUTINE(kern_invalid,0),
        MACHDEP_CALL_ROUTINE(kern_invalid,0),
        MACHDEP_CALL_ROUTINE(kern_invalid,0),
};
```

As you can see in the listing, most machine dependent calls are unused in the Intel architecture. In the 32-bit architecture, calls existed to set the LDT and GDT. In 64-bit, only one call — `thread_fast_set_cthread_self64` — remains, used to set the CPU's `MSR_IA32_KERNEL_GS_BASE` to the thread ID. The `set_cthread_self` function also exists on iOS, wherein it sets the processor's control registers `c13,c0`. You can see its source in libc's `arm/pthreads/pthread_set_self.s`, which demonstrates calling machine specific calls on ARM by setting R12 to `0x80000000` and passing the call number in R3.

Diagnostic calls

As if XNU's vast debug facilities are not enough, it contains a fourth class of system calls reserved exclusively for diagnostics. Unlike Mach traps, UNIX system calls, and machine-dependent calls, there is only one diagnostic call defined, appropriately called `diagCall` (or `diagCall64`), and it selects the type of diagnostics required according to its first argument. Also unlike the other types, this call is only active if the kernel's global diagnostic variable, `dgWork.dgFlags` has set the `enaDiagSCS` bit (`#defined` in `osmfk/i386/Diagnostics.h` as `0x00000008`).

During the PPC era, the diagCall was extremely powerful, and could be used for myriad diagnostics, such as controlling and reading physical memory pages. In its Intel incarnation, however, XNU's `diagCall` has been reduced to support only one code: `dgRuptStat` (#25), used to query or reset per-CPU interrupt statistics. You can verify this for yourself by checking `osfmk/i386/Diagnostics.c`, where this call (in both 32-bit and 64-bit versions) is implemented.

The following experiment shows the usage of `diagCall` to create a simple interrupt statistics viewer, similar to Linux's `/proc/interrupts`.

Experiment: Demonstrating OS X's diagCall()

Listing 8-17, if compiled, will demonstrate the power of `diagCall()` by displaying interrupts in your system:

LISTING 8-17: Demonstrating invoking diagCall() by inline assembly

```
int diagCall (int diag, uint32_t *buf)
{
  __asm__ ("movq    %rcx,%r10; movl    $0x04000001, %eax ; syscall ; ");

};

void main(int argc, char **argv)
{
  uint32_t c[1+ 2*8 + 256*8]; // We'll break at 8 processors or cores. Meh.
  uint32_t i = 0;
  int ncpus = 0;
  int d;
  mach_timebase_info_data_t    sTimebaseInfo;
  memset (c, '\0', 1000 * sizeof(uint32_t));

  if (argc ==2 && strcmp(argv[1], "clear")==0)
        { printf("Clearing counters\n");
          printf("diagCall returned %d\n", diagCall(25,0));
          exit(0);

        }

  printf (" diagCall returned %x\n", diagCall(25,c));

  // Can check for failure by diagCall's return code, or by ncpus:
  // The first entry in the buffer should be set to the number of
  // CPUs, and will therefore be non-zero.

  ncpus= c[0];
  if (!ncpus) { fprintf(stderr,"DiagCall() failed\n"); exit(1);}

  printf("#CPUs: %d\n", c[0]);

  printf ("Sample: \t");
  for (i = 0 ; i < ncpus; i++) {
         uint64_t *sample = (uint64_t *) &c[1+256*i];
```

continues

LISTING 8-17 *(continued)*

```
    if ( sTimebaseInfo.denom == 0 ) {
        (void) mach_timebase_info(&sTimebaseInfo);
    }

        printf ("%15ld\t",
        ((*sample /sTimebaseInfo.denom) * sTimebaseInfo.numer) / 1000000000);
         }
  printf ("\n");

  for (i = 0; i<256; i++) {
        int slot =  1+2 + i; // 1 - num cpus. 2 - timestamp (8 bytes)

        if (c[slot] || c[slot+256+2])
            printf ("%10d\t%10d\t%10d\n", i,c[slot], c[slot+2]);
      }

}
```

You'll note the program has inline assembly for the implementation of `diagCall()`, required because Apple has no public wrapper for diagnostic calls. Also, note the assembly is somewhat similar to the Mach traps and system calls discussed in Chapter 2. The difference, however, lies in the system call class being `0x40000000`, rather than the `0x10000000` for UNIX or `0x20000000` for Mach calls.

Assembly aside, the program is a simple one: with no arguments, it will display the interrupt statistics per CPU. Optionally, it can accept a "clear" argument which will reset the statistics counter. But if you try to execute either functionality, you will likely get an error.

To use `diagCall()`, you must first enable the diag boot-argument, and set its value to `0x00000008`, or any other combination which contains that bit (a safe bet is `0xFFFFFFFF`). You can do that by editing the kernel's boot configuration file, `/Library/Preferences/SystemConfiguration/com.apple.Boot.plist`. This file and other boot arguments are discussed in the next chapter, but the modification you need is a simple one: adding the diag argument to the "Kernel Flags" alongside any already defined, as shown in Listing 8-18:

LISTING 8-18: Adding the diag boot argument to enable diagCall

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>Background Color</key>
        <integer>50349</integer>
        <key>Boot Logo</key>
        <string>\System\Library\CoreServices\BootLogo.png</string>
        <key>Kernel Architecture</key>
        <string></string>
        <key>Kernel Flags</key>
        <string>diag=0x00000008</string> <!--There may be other boot args defined !-->
</dict>
</plist>
```

Once the system has been rebooted, the program should work just fine, and provide you with interrupt statistics. You can verify that “clear” indeed resets the counters.

XNU AND HARDWARE ABSTRACTION

Reading through the chapter, you have no doubt noticed that the two architectures — Intel and ARM — abide by the same general concepts of traps, interrupts and “supervisor mode,” yet take a totally different approach in implementing them (with the approach sometimes changing in between processor models!). Likewise, before migrating to Intel the default architecture of OS X was the PowerPC — another processor with its own approach to implementing these ideas.* How, then, can XNU maintain the same code base for such totally different architectures?

One aspect of hardware agnosticism was already discussed in the chapter dealing with the system boot — it is the Platform Expert module, by means of which the kernel can obtain important hardware configuration data. This, however, only addresses some of the issues raised by different hardware implementations. The kernel itself needs to be modified and adapted to address the various CPU related idiosyncrasies.

XNU does not have a full hardware abstraction layer, per se (as did, at one time, Windows). Rather, the approach it adopted follows the Mach tradition, which is very similar to the one in Linux, as well. Throughout the kernel, there are various macros and functions, which hide architecture specific implementations. Linux does so by means of the `arch`/ subdirectory of its kernel sources, wherein the hardware-dependent implementations of kernel functionality are implemented in corresponding assembly. These either add to, or supersede the existing macros in various other subdirectories of the source. Mach has no one convention for architecture specific functions, though most of them are prefixed with `ml` (machine layer, or machine level), and implemented in `osfmk/i386/machine_routines.c` (and, as a little digging shows, `osfmk/arm/machine_routines.c` for iOS, though the arm branch is of course closed source).

For example, consider the rather simple operation, of enabling/disabling interrupts. Intel processors use a bit in the `EFLAGS` register to mark interrupt masking. The `ml_get_interrupts_enabled` is shown in Listing 8-19:

LISTING 8-19: Interrupt checking on Intel architectures

```
_ml_get_interrupts_enabled:
ffffff800022b884        pushq   %rbp            ; standard
ffffff800022b885        movq    %rsp,%rbp       ;   function prolog…
ffffff800022b888        pushf                   ; push EFLAGS on stack
ffffff800022b889        popq    %rax            ; and copy to RAX
ffffff800022b88a        shrq    $0x09,%rax      ; Shift right 9 bits
ffffff800022b88e        andl    $0x01,%eax      ; isolate (return) last bit
ffffff800022b891        leave                   ; undo prolog
ffffff800022b892        ret                     ; return (rax) to caller
```

*Note, that the PowerPC architecture is completely ignored in this book. This is because Apple, with Lion, has removed PPC support from XNU. For an excellent reference on the PPC implementation (up to and including Tiger), refer to Amit Singh’s book.

On ARM, there is no EFLAGS register. Rather, the interrupt state is maintained in the CPSR (Specifically, the 8th bit). The code for the same function thus becomes what is show in Listing 8-20:

LISTING 8-20 Interrupt checking on ARM architectures

```
_ml_get_interrupts_enabled:
8007c26c      mrs     r2, CPSR           ; R2 gets value of CPSR
8007c270      mov     r0, #1  @ 0x1      ; R0 is set to 0x1
8007c274      bic     r0, r0, r2, lsr #7 ; BIt-Clear (AND-NOT) i.e: R0 = R0 &^(R2 <<7)
8007c278      bx      lr                 ; return (R0) to caller
```

On the deprecated PPC (therefore, on kernels up to and including Snow Leopard only), the EE bit (External Interrupt Enable) is bit #15. So the same function becomes what is shown in Listing 8-21:

LISTING 8-21: Interrupt checking on the (now deprecated) PPC architectures

```
_ml_get_interrupts_enabled:
000c3464        mfmsr   r3               ; Move from Machine-Specific-Register to R3
000c3468        rlwinm  r3,r3,17,31,31 ; Rotate Left Word Immediate then aNd with Mask
000c346c        blr                      ; Return
```

Table 8-7 lists some of the ml_ functions in XNU.

TABLE 8-7: ml_ functions in XNU

ML_ FUNCTION	USED FOR
ml_cpu_up/ml_cpu_down	Activate/Deactivate a processor. Null function on Intel.
ml_is64bit ml_thread_is64bit ml_state_is64bit	64 bit mode of CPU, current thread, and saved state. Implemented as CPU Data macros. Currently not applicable on iOS.
ml_io_map	Map I/O space. Intel implementation wraps io_map() from osfmk/i386/io_map.c
ml_phys_[read/write]_[*xxx*][_64]	Functions to read and write physical memory elements (*xxx* can be byte/half/word/double)
ml_static_ptovirt	Physical to Virtual translation. In ARM, this is done using special registers (p15's c7,c8). In Intel, this follows the PTE/PDE mechanism.
ml_[get/set]_interrupts_enabled ml_at_interrupt_context ml_install_interrupt_handler ml_cause_interrupt	Get/set interrupts (discussed above), determine if in interrupt. ml_install_interrupt_handler() is used by IOKit drivers, and actually wraps the platform expert. ml_cause_interrupt is not supported on Intel (and would cause a kernel panic)

It should be noted that while the `ml_` functions are fairly abundant, they do not cover all hardware-specific aspects. As you will see later, many more implementations (e.g. atomic operations, per-CPU data, the "pmap" physical memory abstraction, and more) can be handled in other ways. This is what is meant by the "specific hacks" in the OS X architectural diagram presented throughout this book. Fortunately, porting is not really the developers' problem so much as it is Apple's.

SUMMARY

This chapter discussed the fundamental concepts of operating system architecture. User mode, kernel mode, and the transition mechanisms between them are all supported by the underlying hardware, be it OS X's Intel or iOS's ARM.

The two architectures were compared and contrasted, showing both the theory of each, and then the implementation — in OS X and iOS both — by viewing the low-level assembly. The chapter discussed the implementation of the various system call classes, predominantly UNIX system calls and Mach Traps, and concluded with a discussion of XNU's `ml_*` hardware abstraction primitives.

The next chapter will take you deeper into XNU, introducing you to its source tree, and its boot process. This will enable you to get more comfortable, as the second part of this book ensues, and we delve deeper still into the internals of the kernel common to both OS X and iOS.

REFERENCES

1. The Intel X86_64 Architecture manuals, Volumes 1, 2, 3A, and 3B
2. The ARM Architecture Manuals — online at `http://infocenter.arm.com`
3. Esser, Stefan. "Targeting the iOS Kernel," Syscan 2011, `www.syscan.org`

9

From the Cradle to the Grave — Kernel Boot and Panics

In previous chapters, you have seen how, depending on architecture, the kernel image is found and arguments are passed to it. This chapter picks up where the others have left off and presents a detailed description of how XNU boots — in both OS X and iOS. By going over the kernel sources line by line, you will be able to follow the steps the kernel takes in initializing the system.

This chapter also discusses the premature demise of the kernel, which occurs in cases where an unhandled CPU trap, or other unexpected kernel code path, causes a "panic."

THE XNU SOURCES

To better understand this chapter and this entire part of the book, it is highly recommended that you follow along with the XNU sources. Much like the Linux kernel, XNU sources are freely downloadable. This section details the steps required to obtain and compile XNU.

Getting the Sources

Ever since Apple annexed CMU's Open Source Mach project, it has selectively kept XNU open source. The key word here is "selectively," because Apple only publishes the OS X compiled version. For iOS (i.e. the ARM port of XNU), Apple keeps the XNU source closed. The two used roughly the same kernel version until iOS 4.2, when iOS "took off" and advanced in its kernel version beyond that of OS X. At the time of writing, for example, iOS 5 is at XNU 1878, whereas Lion is lagging still at 1699. This is likely going to change as Mountain Lion takes the lead (with version 2050), unless iOS 6 continues the trend and leaps ahead.

The source code excerpts provided here are from XNU 1699.26.8, which you can download as a tarball from `http://opensource.apple.com/tarballs/xnu/xnu-1699.26.8.tar.gz` and unpack (using `tar zxvf`). This is the version of the kernel Apple provides with Lion 10.7.4, the latest available as this book is frozen in print. It's more than likely that by the time you read these lines, however, a newer kernel version will be available. This version will likely be Mountain Lion's (or later?), and may possibly introduce some changes from the listings in this book. If that is the case, you can either stick to the XNU version cited in this book, or obtain the latest one. In any case, in order to follow along the examples, even outdated open source certainly beats binary disassembly.

> *Take advantage of Apple's XNU source repository at* `http://opensource.apple.com/tarballs/xnu/`*. Examining the same function in different versions of the kernel will enable you to get a firsthand impression of the modifications Apple introduced over time to XNU, following the evolution step by step. You don't even need to download the sources locally: The source tree is available unpacked in* `http://opensource.apple.com/source/xnu/xnu-XXXX.yy.zz/`*, so you can simply append the path of the file you are interested in, and replace the version number of XNU with the kernel you are interested in.*
>
> *Alternatively, check out the book's companion website, which offers an HTML-enabled cross reference, similar to the Linux LXR.*

Making XNU

If you have Apple's developer's tools installed, you are steps away from compiling XNU. This is a fairly straightforward, albeit lengthy, process — but well worth it. Compiling enables you to see first-hand each and every stage of the boot process. You can easily insert debugging and logging messages, as well as selectively comment or `#ifdef` out portions. XNU already has a plethora of debugging information embedded in its code, which you can reveal with a simple `#define DBG` (or `-DDBG`) when making it.

Using the developer tools, you can compile XNU for either Intel 32-bit or 64-bit architecture. The GCC compiler in the developer tools can compile XNU easily, provided that the prerequisites listed in the next section are satisfied.

Prerequisites

To build XNU, you need several development tools:

- **`Cxxfilt`**: Current version: 9. The real name of this package is `C++filt`, but + is an illegal character in DOS filenames.
- **`Dtrace`**: Current version: 7.8. Required for `CTFMerge`.
- **`Kext-tools`**: Current version: 180.2.1.
- **`bootstrap_cmds`**: Current version: 72. Required for `relpath` and other commands.

Fortunately, all these tools are freely available for download from Apple's open-source site. Getting the tarballs is straightforward, although their versions are often updated.

To build `Cxxfilt` and bootstrap commands, a simple `make` usually suffices. Define `RC_OS` to `macos` and `RC_ARCHS` to `i386`, `x86_64`, or both.

`DTrace` and `Kext-tools` build using XCode's command line `xcodebuild`.

To summarize, your command line will resemble the following, as shown as Listing 9-1:

LISTING 9-1: Obtaining and making the prerequisites for building XNU

```
 #
 # Getting C++ filter
 #
$ curl http://opensource.apple.com/tarballs/cxxfilt/cxxfilt-9.tar.gz >
       cxx.tar.gz
$ tar xvf cxx.tar.gz
$ cd cxxfilt-9
$ mkdir -p build obj sym
$ make install RC_ARCHS="i386 x86_64" RC_CFLAGS="-arch i386 -arch x86_64 -pipe" \
 RC_OS=macos RC_RELEASE=Lion SRCROOT=$PWD OBJROOT=$PWD/obj \
 SYMROOT=$PWD/sym DSTROOT=$PWD/build
  #
  # Getting DTrace - This is required for ctfconvert, a kernel build tool
  #
$ curl http://opensource.apple.com/tarballs/dtrace/dtrace-90.tar.gz > dt.tar.gz
$ tar zxvf dt.tar.gz
$ cd dtrace-90
$ mkdir -p obj sym dst
$ xcodebuild install -target ctfconvert -target ctfdump -target ctfmerge \
ARCHS="i386 x86_64" SRCROOT=$PWD OBJROOT=$PWD/obj SYMROOT=$PWD/sym \
DSTROOT=$PWD/dst
  #
  # Getting Kext Tools
  #
$ curl http://opensource.apple.com/tarballs/Kext_tools/Kext_tools-180.2.1.tar.gz \
 > kt.tar.gz
$ tar xvf kt.tar.gz
$ cd Kext_tools-180.2.1
$ mkdir -p obj sym dst
$ xcodebuild install -target Kextsymboltool -target setsegname \
ARCHS="i386 x86_64" SRCROOT=$PWD OBJROOT=$PWD/obj SYMROOT=$PWD/sym \
 DSTROOT=$PWD/dst
  #
  # Getting Bootstrap commands - newer versions are available, but would
  # force xcodebuild
  #
$ curl http://opensource.apple.com/tarballs/bootstrap_cmds/bootstrap_cmds-72.tar.gz \
 > bc.tar.gz
$ tar zxvf bc.tar.gz
$ cd bootstrap_cmds-84
$ mkdir -p obj sym dst
$ make install RC_ARCHS="i386" RC_CFLAGS="-arch i386 -pipe" RC_OS=macos \
 RC_RELEASE=Lion SRCROOT=$PWD OBJROOT=$PWD/obj SYMROOT=$PWD/sym DSTROOT=$PWD/dst
```

Making the Kernel

Once all the prerequisites mentioned in the previous section are satisfied, making the kernel is straightforward, as shown in Listing 9-2:

LISTING 9-2: Making the kernel

```
$ wget http://opensource.apple.com/tarballs/xnu/xnu-1699.26.8.tar.gz # or curl
$ tar xvf xnu-1699.26.8.tar.gz
$ cd xnu-1699.26.8
$ make ARCH_CONFIGS="I386 X86_64" KERNEL_CONFIGS="RELEASE"
MIG clock.h
MIG clock_priv.h
MIG host_priv.h
Generating libkern/version.h from…/1699.26.8/libkern/libkern/version.h.template
MIG host_security.h
...
... (many more lines omitted for brevity)
```

The build will take some time, progressing through each directory. For each file, the build requires one or more of the following actions, shown in Table 9-1:

TABLE 9-1: Build Actions

ACTION	PURPOSE
`AS`	Assemble: Used on `.s` files
`C++`	Compile C++: Used on `.cpp` files (IOKit)
`CC`	Compile: Used on `.c` files
`CTFCONVERT`	Prepare/Process Compact Text Format debugging information
`LDFILELIST`	Link: Used on directories, once all the files in them have been compiled
`MIG`	Mach Interface Generator: Used on `.defs` files, to creates client/server Mach message passing code from stub definitions. The generated files are then compiled (CC)

If the process is successful, the built kernel will be found in `BUILD/obj/RELEASE_I386`, `BUILD/obj/RELEASE_X86_64`, or both. Using the `lipo(1)` tool, you can construct one fat binary to contain both architectures, although that is not strictly necessary.

One Kernel, Multiple Architectures

Apple has adapted XNU to run on no less than four architectures: PowerPC, i386, x86_64, and, in iOS, ARM. In doing so, it drew on its core — Mach — which, by design, was made flexible for any architecture.

Similar to the Linux kernel, which may be compiled for specific architectures, so can Mach. Both kernels follow a similar design. Most of the kernel is architecture-agnostic, and architecture-idiosyncratic parts are implemented in corresponding directories.

In Linux, this is achieved by defining functions as macros and overriding the basic implementations with architecture optimized ones, found in the `arch/` subdirectory of the source tree. In this way, the kernel entry points, low-level thread, and memory management are coded in highly specialized assembly (.s files), while the rest is in C++.

The principle in Mach is almost the same: The `osfmk/` directory, in which the Mach sources reside, has architecture-specific subdirectories. In the open-source XNU, these are `i386/` and `x86_64/`. Older versions of XNU also contain a `ppc/` subdirectory. Strings inside the iOS kernel reveal that a fourth directory, `arm/`, which Apple keeps closed source.

Additionally, XNU relies on a specialized directory, `pexpert` — the so called Platform Expert. This directory is a small, yet highly important one. It contains specialized functions for each architecture. In the open-source version, the only supported architecture is i386/x64 (both under `i386`), but iOS has a similar ARM platform expert, which — again — Apple keeps private (though its symbols, too, occasionally leak in iOS versions).

The i386 Platform Expert is tightly integrated with EFI (from which it obtains configuration parameters) from one end and with IOKit (for which it provides services) from the other. The ARM Platform Expert is similarly integrated with iBoot. Table 9-2 shows the pexpert subdirectory on OS X only. iOS is likely different.

TABLE 9-2: pexpert subdirectory ()

SUBDIRECTORY	CONTAINS
conf	Machine-specific makefiles
gen	Contains the code to handle the boot arguments (`bootargs.c`), device tree (`devicetree.c`) and the output/boot logo (`pe_gen.c`) files
i386	Low-level handlers for interrupts, serial, and machine identification
Pexpert	Contains the header files for all the Platform Expert components the other kernel components use

IOKit, the XNU driver framework, makes extensive use of the Platform Expert. But even the kernel core frequently relies on PE calls. The most commonly called on feature of the Platform Expert is the `_PE_state`, which is a platform dependent singleton structure representing the initial state of the machine, as set up by the boot loader. On an Intel platform, it looks like this:

```
typedef struct PE_state {
        boolean_t       initialized;
        PE_Video        video;
        void            *deviceTreeHead;
        void            *bootArgs;
```

```
} PE_state_t;

PE_state_t  PE_state;
```

With `PE_Video` being the graphics console information, as in the following:

```
struct PE_Video {
  unsigned long   v_baseAddr;  /* Base address of video memory */
  unsigned long   v_rowBytes;  /* Number of bytes per pixel row */
  unsigned long   v_width;     /* Width */
  unsigned long   v_height;    /* Height */
  unsigned long   v_depth;     /* Pixel Depth */
  unsigned long   v_display;   /* Text or Graphics */
  char            v_pixelFormat[64];
  unsigned long   v_offset;    /* offset into video memory to start at */
  unsigned long   v_length;    /* length of video memory (0 for h * w) */
  unsigned char   v_rotate;   /* Rotation: 0:0 1: 90, 2: 180, 3: 270 */
  unsigned char   v_scale;        /* Scale Factor for both X & Y */
  char            reserved1[2];
#ifdef __LP64__
  long            reserved2;
#else
  long            v_baseAddrHigh;
#endif
};
```

A call to `PE_init_platform` (in `pexpert/i386/pe_init.c`) sets up the `PE_state`, most importantly the `bootArgs` pointer. Various kernel components can then access the arguments using `PE_parse_boot_argn()`:

```
boolean_t PE_parse_boot_argn(
        const char      *arg_string,
        void            *arg_ptr,
        int              max_arg);
```

This function allows a caller to specify an `arg_string`, and an `arg_ptr`, a buffer of up to `max_arg` bytes, which will be populated by the function (returning `true`) if the argument was supplied on the kernel command line.

Another commonly used functionality of the Platform Expert is the device tree. This is a rendering of all the devices in the system in a hierarchical tree structure, much like Solaris' `/devices` or Linux's `/sys/devices`. The device tree is initialized by the boot loader (OS X: EFI, iOS: iBoot), and allows the kernel to query which devices are connected. The device tree is detailed in Chapter 6.

The Platform Expert is also used in the low-level handling of CPU, virtual memory, and other hardware. This is why the IOKit makes such frequent use of it. From the user mode perspective, the flow of a system call, (or Mach trap), starts as an architecture agnostic BSD/Mach call, and as it traverses the layers of the kernel, it gets more and more specific. The IOKit also creates a specialized class, IOPlatformExpert, which is used to instantiate a singleton — gIOPlatform — which is then consulted for machine-related information. IOPlatformExpert is defined in an architecture-specific manner, although it does have similar methods across architectures. This will be elaborated on in Chapter 19, which deals exclusively with IOKit.

Configuration Options

XNU has quite a few configuration options, which you can toggle before compiling the kernel. These are #defines, which either set various buffer values, or enable parts of the code and hide others at the preprocessor level, so that the resulting objects are as slim as possible. Most are prefixed with CONFIG, though not always. There are far too many options to list in this book, but the interesting ones include those shown in Table 9-3:

TABLE 9-3: Some of the Configuration Options for Building XNU

OPTION	AFFECTS
CONFIG_AUDIT	Enables the audit subsystem.
CONFIG_DTRACE	Enables DTrace hooks in kernel.
CONFIG_EMBEDDED	Sets embedded device features. Apple sets this for iOS.
CONFIG_MACF	MAC security policy.
CONFIG_NO_PRINTF_STRINGS CONFIG_NO_KPRINTF_STRINGS	Saves 50 K of kernel memory, and makes life a little bit harder for iOS reverse engineers, where it is used.
CONFIG_SCHED_*	Select specific task scheduling algorithm. XNU offers TRADITIONAL, PROTO, GRRR, and FIXED_PRIORITY. Scheduling is discussed in Chapter 12.
SECURE_KERNEL	Kernel security extensions.

Every subdirectory of the kernel source tree (which corresponds to a subsystem) contains a conf/ subdirectory, which controls the options of its subsystem. The options are documented in MASTER files.

The XNU Source Tree

XNU's source tree is considerable — around 50 MB when fully extracted. While it is not as large as the Linux source tree (which is double this figure, even with most drivers excluded), it is still easy to get lost in the source.

A slightly easier way to navigate the source is with the FXR tool, at http://fxr.watson.org/. This tool, (derived from LXR, the Linux Cross Reference tool), explores FreeBSD's source tree, and other code bases, including XNU. The latest version indexed at the time of writing is 1699.24.8 (OS X 10.7.2).

FINDING A SYMBOL OR STRING IN THE SOURCE FILES

If you're looking for a particular function name, variable, or other symbol in the source files, `grep(1)` is your friend. You can use `grep` to enter any regular expression and find it in the `.h` or `.c` files, and — by using `xargs(1)` — extend the command so that the search covers all files in the directory.

For example, if you are looking for `vstart`, you would `cd` to the `xnu` source root directory, and type the following:

```
morpheus@Ergo(../xnu-1699.26.8)$ find . -name "*.c" -print | xargs
grep vstart
./bsd/dev/i386/fbt_x86.c:    "vstart"
./osfmk/i386/i386_init.c: * vstart() is called in the natural mode
(64bit for
./osfmk/i386/i386_init.c:vstart(vm_offset_t boot_args_start)
./osfmk/i386/i386_init.c:   DBG("vstart() NX/XD enabled\n");
./osfmk/ppc/pmap.c: *    kern_return_t pmap_nest(grand, subord,
vstart, size)
... (Other results omitted for brevity) ..
```

The approach is a brute force one, at best, as all instances of your search string will be returned. If the string is a common substring, brace yourself for many results. Still, with a little C, you should be able to sift through the results and find the one or few which are relevant to your search — useful when you don't have access to the HTML cross references.

To make your life easier, nearly all the functions in XNU are implemented so that their name begins the line in which they are implemented. That is, their return value is deliberately stated in the preceding line. This makes it easy to find the implementation of a function you are looking for by using `grep` with the caret (^) sign, which is reserved for the beginning of a line. In the preceding example, using the caret would have given us exactly the result we want:

```
morpheus@Ergo (../xnu-1699.26.8)$ find . -name "*.c" | xargs grep ^vstart
./osfmk/i386/i386_init.c:vstart(vm_offset_t boot_args_start)
```

The regular expression syntax can be further tweaked to filter results, for example by looking for \ at the end of the symbol (denoting where function arguments begin).

XNU's source tree is large, but fairly well organized into several subtrees. These subtrees contain the implementation of the various kernel subsystems, as shown in Table 9-4:

TABLE 9-4: The XNU Subtrees

DIRECTORY	CONTAINS
`bsd`	BSD components of kernel
`config`	Exported symbols for various architectures
`iokit`	The I/O Kit driver runtime subsystem
`libkern`	The kernel main runtime library APIs
`osfmk`	Mach components of kernel
`pexpert`	Platform-specific stuff (PPC, i386)
`security`	The BSD MAC Framework

The BSD layer is further broken down into subcomponents, as you can see in Table 9-5:

TABLE 9-5: BSD Subdirectory

SUBDIRECTORY	CONTAINS
`bsm/security`	Basic Security Module (auditing subsystem)
`conf`	Machine-specific `Makefile`s
`crypto`	Implementations of symmetric algorithms and hashes
`dev`	BSD Devices (`/dev directory entries`)
`hfs`	File system driver (HFS/HFS+) is OS X default
`i386/machine/ppc`	Private kernel headers for Intel/PPC architectures
`kern`	Main kernel code
`libkern`	Kernel runtime exports (CRC, string functions)
`man`	Some actually useful man pages
`net*/netinet*`	Networking subsystem (sockets) and IP stack
`nfs`	NFSv3 stack, for remote file systems
`sys`	Kernel headers
`vfs`	Virtual Filesystem Switch
`vm`	BSD's virtual memory handlers

Likewise, Mach, in the `/osfmk` (Open Software Foundation Mach Kernel) subdirectory has the subdirectories shown in Table 9-6.

TABLE 9-6: OSFMK Subdirectory

SUBDIRECTORY	CONTAINS
`chud`	The Computer Hardware Understanding Development tools. These extremely powerful APIs formed the kernel support for OS X diagnostic tools (known as the CHUD tools), which included the legendary Shark utility, Reggie SE and others. Ever since Leopard (10.5) they have been gradually phased out of OS X, losing ground to DTrace. The code support for them, however, still exists. See the discussion in Chapter 5.
`conf`	Machine-specific `Makefiles`
`console`	Console initialization, serial, boot video and panic UI
`ddb`	Kernel debugger (obsolete)
`default_pager`	VM Pager
`device`	Mach support for I/O Kit and devices
`i386/ppc/x86_64`	CPU-specific implementations (the good stuff)
`ipc`	IPC, ports, and messages
`kdp`	KDP (Debugger) support
`mach, machine`	The Mach generic and machine dependent kernel headers
`man`	The only man pages you'll ever get on Mach calls
`pmc/profiling`	PMC performance monitoring
`UserNotification`	Kernel-User Notification (KUNC)
`vm`	Virtual memory implementation and headers

BOOTING XNU

XNU is a Mach-O object. The boot loader (EFI or iBoot) contain Mach-O parsing code, and can deduce the entry point from the `LC_UNIXTHREAD` command. Using `otool`, you can do so as well.

It is a worthwhile experiment to compile XNU with the various debug settings (`DEBUG`, `CONFIG_DEBUG`, and their ilk) and follow the full debug output, as it will show the flow much like in the following pages. To capture serial output, it is a good idea to run OS X in a Virtual Machine, and define a serial port, redirected to a text file. Even though OS X is technically not supposed to be virtualized, there are many articles and tutorials on how to trick it into running inside a virtual machine, after all.

The boot process is a long and arduous flow, spanning multiple files. Reading this following section in depth will no doubt be tedious. It is recommended that, as a first read, you go over this section in more of a cursory read, not stalling to mull on the aspects which may seem unclear or obscure. Then, after reading the next chapters — wherein the Mach and BSD layers are described in depth — revisit this section, and things will "fall into place."

The Bird's Eye View

The high level view of XNU's boot process is given in Figure 9-1. This is a greatly simplified and somewhat inaccurate view, but it serves as a point of departure for this chapter, as we zoom in with ever-increasing resolution

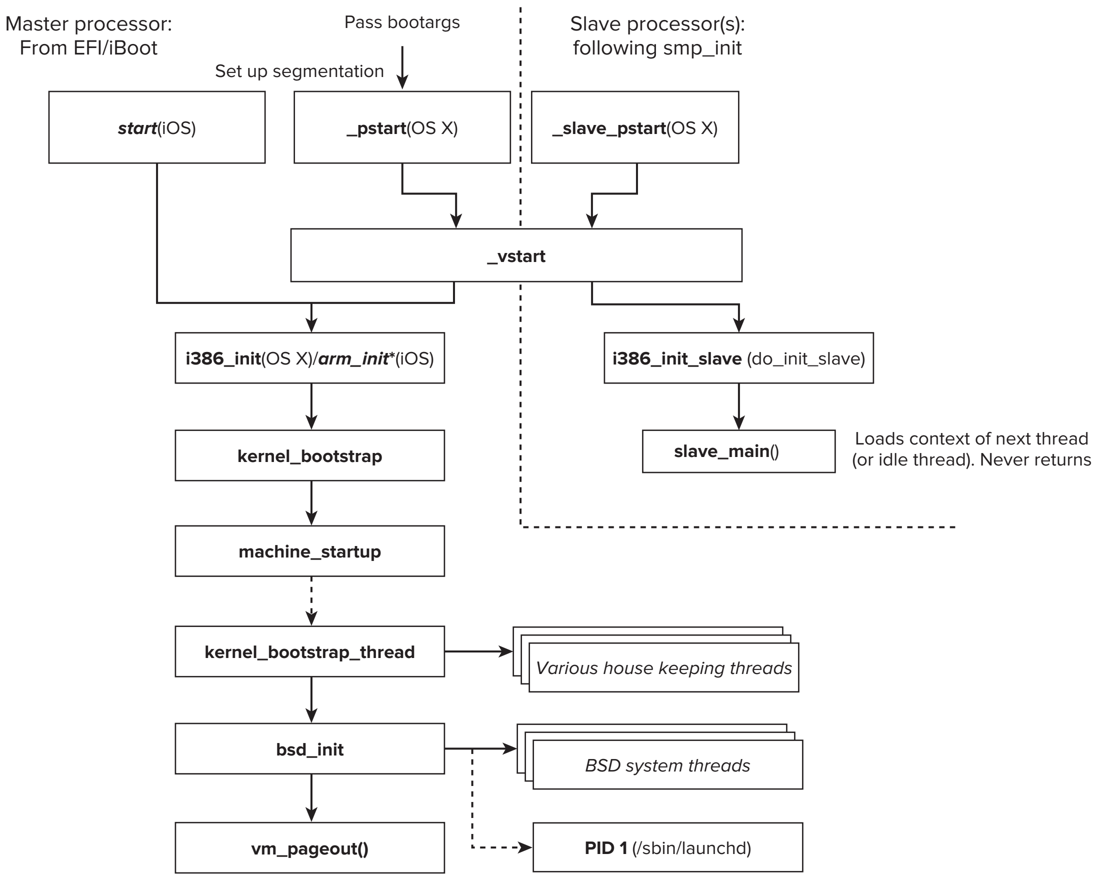


FIGURE 9-1: The high level view of XNU's boot

> *Apple originally left iOS's XNU fully intact with symbols, when the closest thing to a "jailbreak" was an American prime time TV drama with a similar name. Since then, however, iOS has been aggressively and repeatedly stripped, with fewer and fewer symbols remaining with every new release. XNU hasn't changed that dramatically, so a bit of common sense (and other oversight by Apple) allows the reconstruction of symbols. In some cases, however – particularly new code such as SMP (i.e. ARM dual-core support), which was introduced in iOS 4.3 with the iPad 2, the symbols are unknown, and the logic is deduced from educational binary inspection. The iOS picture therefore remains, in some cases, incomplete, and may be subject to change.*

OS X: vstart

`vstart` (`osfmk/i386/i386_init.c`) is the i386/x64 "official" kernel initialization function, and marks the transition from assembly code to C. It is also a special function, in that it executes on the primary (boot) CPU, as well as any slave CPUs (or cores) present in the machine. The slaves can tell themselves apart because the argument to `vstart`, the `boot_args_start` pointer, is NULL for slaves.

The following list depicts the flow of `vstart` on OS X:

- **On Boot (master) CPU**: `vstart` optionally (#if DBG) initializes the serial line by calling `pal_serial_init()`.
- **Enable NX/XD**: On x64 platforms, the NX (No Execute) bit is a processor feature meant to combat code injection. Pages marked as data (commonly the stack and heap) will trigger a page fault if accessed by the Instruction Pointer. This is a hardware enforced mechanism, which defeats a significant part of the code injection techniques, although not all of them; return-oriented programming — the diverting execution to pre-existing library code — will still work.

  The NX/XD bit is set per-processor — master and slaves alike, if `cpuid_extfeatures` (from `osfmk/i386/cpuid.c`) reports this feature is present (`CPUID_EXTFEATURE_XD`).
- **`cpu_desc_init[64]` (`osfmk/i386/mp_desc.c`)**: This initializes the GDT and LDT on the master cpu. This is followed by a call to `cpu_desc_load(64)`, which loads the kernel LDT for use on both master and slaves.
- **`cpu_mode_init()` (in `osfmk/i386/mp_desc.c`)**: This nitializes the CPU's MSRs (used for SYSENTER/SYSCALL), and its physical page map (`pmap`)
- **i386_init/i386_init_slave**: This is called from either the master or slave CPUs.

iOS: start

In iOS most of the boot-related functions have been stripped, yet the `start()` function remains one of the few proudly exported symbols. It will likely remain so, as it is declared in XNU's

LC_UNIXTHREAD command as well. The entry point is in the vicinity of 0x8007c058. In the iPhone 4S, where a XNU decrypted binary is, as yet, unavailable, it resides in 0x8007A0B4.

The entry point has an unusual structure, which helps in its disassembly: Its first three instructions, shown in Listing 9-3, are uncommon enough to allow its detection, and also that of the next step, arm_init. The start() function loads the address of the latter into the link register (R14), so that it effectively returns to it on exit, and then disables interrupts. The entry point for iOS 6 will likely be in the 0x8007*xxx* to 0x8008*xxx* range, though (if Mountain Lion is any indication) kernel ASLR will randomly "slide it" on every boot.

LISTING 9-3: The iOS entry point start code (obtained with the corerupt tool)

```
start:
0x8007A0B4      MOV         R1, #0
0x8007A0B8      LDR         LR, =_arm_init      ; Load next stage as return address
0x8007A0BC      CPSID       IF                  ; Shhh! Disable Interrupts (IRQ/FIQ)
...
0x8007A0D8      MCR         p15, 0, R5,c2,c0, 0  ; Translation table base 0
0x8007A0DC      MCR         p15, 0, R5,c2,c0, 1  ; Translation table base 1
0x8007A0E0      MOV         R5, #2                     ; Boundary size 4K (as page
size)
0x8007A0E4      MCR         p15, 0, R5,c2,c0, 2  ; Translation Table base control
...   ;
0x8007A318      MOV         R5, #0
0x8007A31C      MCR         p15, 0, R5,c8,c7, 0  ; Invalidate I and D TLBs
0x8007A320      DSB         SY
0x8007A324      ISB         SY
0x8007A328      MOV         R7, #0
0x8007A2EC      BX          LR                   ; "returns" to arm_init
```

In the sequence that follows, this function mostly handles low level processor settings, through the ARM control registers, installs the kernel's trap handlers from the ExceptionVectorsBase (discussed in Chapter 8), manipulates more settings, and then jumps to arm_init.

[i386|arm]_init

The platform initialization function — in OS X's case **i386_init()** — initializes the master CPU for use, and readies the kernel boot. A similar functions, in OS X's case — i386_init_slave() — does the same for the slave CPUs. This function is expected to never return. Unlike the next stages, which are largely similar on both platforms, this step is highly specific. This is why the function name contains the architecture name.

In iOS, this function is replaced by arm_init(), which provides very similar functionality, albeit suited for the ARM platform. Its flow is largely the same, give or take a function, such as a call to arm_vm_init() for virtual memory, and a call to ml_io_map(), which the Intel version doesn't have.

The init function is long, but well structured. Like the rest of the functions involved in the boot process, it calls on subroutines to perform the work of initializing each subsystem or component. You can follow the flow in Figure 9-2:

i386_init

pal_i386_init (osfmk/i386/pal_routines.c)	Calls on the Platform Abstraction Layer initialization — in effect simply a call to initialize a lock on the EFI.
PE_Init_Platform (pexpert/i386/pe_init.c)	Initializes the `PE_state` global, which contains a copy of the boot arguments, video arguments, and more. This function calls `pe_identify_platform` (from `pexpert/i386/pe_identify_machine.c`), to set `gPEClockFrequency`.
kernel_early_bootrap (osfmk/kern/startup.c)	This calls lck_mod_init (osfmk/kern/locks.c) and timer_call_initialize (osfmk/kern/timer_call.c) which is used in timer calls.
cpu_init (osfmk/i386/cpu.c)	Sets the current CPU clock timer's deadline to the ominous "EndOfAllTime". Literally, this is no joke. The 64-bit maximum value, is some 677 billion years in our future, long after you, the author, and all humanity perishes, and our universe ceases to exist. After the clock is set to run indefinitely, `cpu_init() calls i386_activate_cpu()(osfmk/i386/mp.c).`
printf_init (osfmk/kern/printf.c)	Called in case a debugger will be attached. Then, kernel printf() messages will be directed to the debugger.
panic_init (libsyscall/mach/panic.c)	Called to redirect any kernel panics so they can be intercepted by an attached kernel debugger.
PE_init_kprintf (pexpert/i386/pe_kprintf.c)	Called to enable kprintf() output to get to the console.
Check for serial console	Check the "serial" boot arg, and switch_to_serial_console() if set.
PE_Init_printf (pexpert/gen/pe_gen.c)	Called to enable printf() output to get to the console.
64-bit processor detection	If CPU features support the `CPUID_EXTFEATURE_EM64T` flag, it will be enabled– unless `"-legacy"` was specified as a command line argument to the kernel.
i386_vm_init (osfmk/i386/i386_init.c)	Takes over virtual memory management from EFI. Also calls `pmap_bootstrap (osfmk/i386/pmap.c)` to initialize kernel physical memory map.
PE_init_platform (pexpert/i386/pe_init.c)	`PE_init_platform` is called again, this time with its first argument set toTRUE, indicating virtual memory has been initialized. It obtains the video information and the device tree from EFI.
PE_create_console (pexpert/i386/pe_init.c)	Starts either the graphics mode console, or the text mode console.
tsc_init (osfmk/i386/tsc.c)	Obtains FSB frequency and other parameters from EFI, and the CPU's Time Stamp Counter (TSC register) frequency from the CPU. It then calculates the conversion factor between the two.
power_management_init (osfmk/i386/pmCPU.c)	Sets up the `pm_init_lock`, which is later used by the kernel extension which manages power.

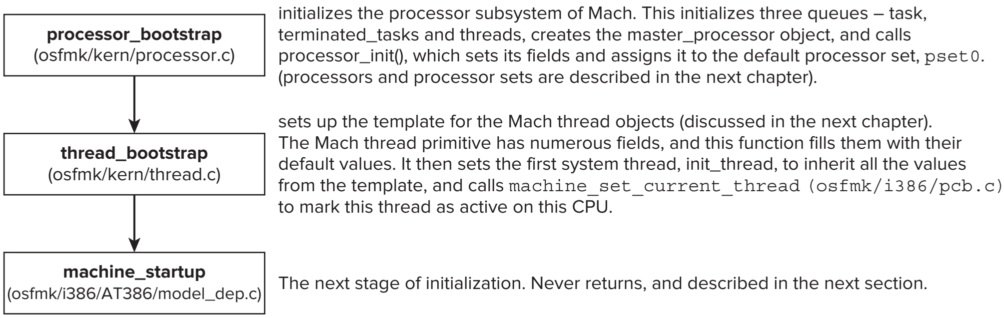


FIGURE 9-2: i386_init flow

A considerable amount of work in the `<platform>_init` function goes to checking for the existence of a console device, initializing it and redirecting the kernel's `printf()`s and `kprintf()`s to it. The console of an OS X device is usually its keyboard and screen, and using the `-v` (verbose) boot argument you can see a verbose boot (alternatively, by pressing Alt+V while rebooting). You can also do so in iOS, if you pass the `–v` argument through redsn0w or other utilities, though the screen often flashes too quickly for any meaningful output to be discerned.

If the `serial` boot argument is specified, the kernel can redirect the console to a serial port, instead. This method comes in handy in iOS to enable kernel debugging. As noted by security researcher Stefan Esser and discussed previously in this book, the iOS serial port may be enabled (though it requires some equipment and minor soldering).

i386_init_slave()

Slave processors' real-mode entry point is set (by `smp_init`, later on), to be `slave_pstart`. This function, in turn, merges with the `start_common`, but leaves the kernel bootargs structure pointer as NULL. The common code calls `vstart`, as shown earlier, but slave processors can then tell themselves apart from the master due to the NULL argument.

`vstart()` behaves slightly differently for the master processor than it does for the slaves, performing the one-time kernel initialization if it detects it is running on the master. Then, the roads diverge; whereas the master processor executes `i386_init()`, the slaves turn to `i386_init_slave()` instead. This function is a call through to `do_init_slave(FALSE)`.

do_init_slave()

The `do_init_slave` function is called when a slave processor wakes up, either for the very first time, or when it awakes from hibernation/sleep. First, the function checks its argument — `fast_restart:` — which may indicate this is a call from `pmCPUHalt` (`osfmk/i386/pmCPU.c`). A fast restart merely wakes up the CPU, whereas a slow, or full start, initializes and then starts the CPU. This, in turn, involves:

- Setting caching and write-through by ensuring the NW and CD flags of CR0 are off

- Configuring the local interrupt controller — `lapic_configure()` — from `osfmk/i386/lapic_native.c`)
- Initializing the FPU (`init_fpu(), osfmk/i386/fpu.c`) in the same manner as `machine_init()`, described later

In either a fast or slow startup, the next step is a call to initialize the CPU (`cpu_init(), osmfk/kern/cpu.c`), as performed by `i386_init` for the main. The function then calls `slave_main` (from `osfmk/kern/startup.c`). This function takes the next available thread for execution from the `current_processor()`'s `next_thread` field. If no runnable threads exist, the idle thread (created by `kernel_bootstrap_thread`) is taken instead. As the thread context is loaded into the processor, this function had better not return (or the kernel will panic).

machine_startup

`machine_startup(osfmk/i386/AT386/model_dep.c)` function, called at the last step of `<platform>_init`, is misleading: although its name and location both seem to imply hardware and model dependency, it is actually less dependent on the underlying hardware than its predecessor, and has the same implementation in OS X and in iOS.

The function mostly parses several command line arguments (using the Platform Expert's `PE_parse_boot_argn`), mostly flags of the debug `boot-arg`, to control boot-time debugging. If `MACH_KDB` is defined, a call to `ddb_init(osfmk/ddb/db_sym.c)` initializes Mach's low-level kernel debugger and halts the kernel boot at this stage, so a debugger may be attached. Otherwise, a few more command line arguments (dealing with scheduling quanta and preemption) are parsed, and then a call to `machine_conf()` sets the `machine_info` structure's `memory_size` field. The full list of arguments can be found later in this chapter.

A call to `ml_thrm_init()` hints at some future plans to initialize CPU thermal reporting on Intel processors, as PPC's XNU had, but `NOTYET`: this is `#ifdef'ed` out on both OS X and iOS. The last step is, therefore, a fall through to `kernel_bootstrap()`, which also never returns, and performs the bulk of the low level Mach initialization.

kernel_bootstrap

The `kernel_bootstrap(osfmk/kern/startup.c)` function continues to setup and initialize the core subsystems of the Mach kernel, erecting the necessary foundations upon which the BSD is overlaid. From this stage onward, initialization is largely the same in OS X and iOS, with a few minor differences that relate to low-level initialization of machine-dependent aspects (such as the physical map abstraction), or to specific features, most of which are new to iOS.

Aside from virtual memory (without which there is nothing), `kernel_bootstrap` also initializes the key abstractions of Mach:

- **IPC**: Mach is based around message passing, and this requires significant resources, such as memory, synchronization objects, and the Mach Interface Generator (MIG).
- **Clock**: The clock abstractions enable alarms (the system clock) and time-telling (the "calendar").

- **Ledgers**: Ledgers are part of Mach's system enabling accounting. This has recently been revamped in iOS 5 and Mountain Lion.
- **Tasks**: Tasks are Mach's containers, akin to BSD's processes (in fact, a 1:1 mapping exists between the two).
- **Threads**: Threads are the actual units of execution. A task is merely a resource container, but it is the thread which gets scheduled and executed.

The `kernel_bootstrap` function doesn't return. Instead, it assumes the context of the `kernel_bootstrap_thread`, which is the system's first active thread. As this thread, it carries on with initialization, dealing with subsystems of increasing complexity.

The flow of `kernel_bootstrap` is annotated in Figure 9-3.

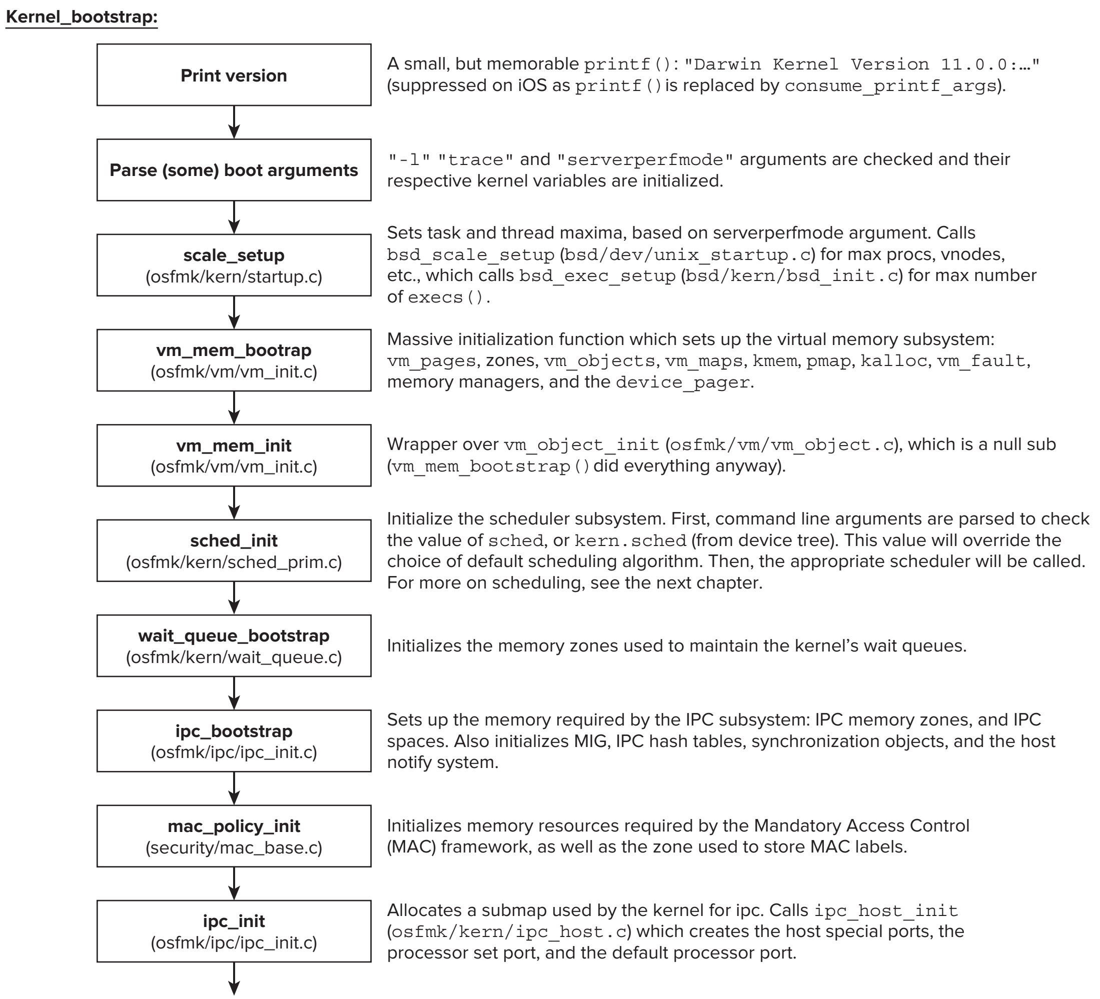


FIGURE 9-3: The flow of kernel_bootstrap (from osmfk/kern/startup.c)

continues

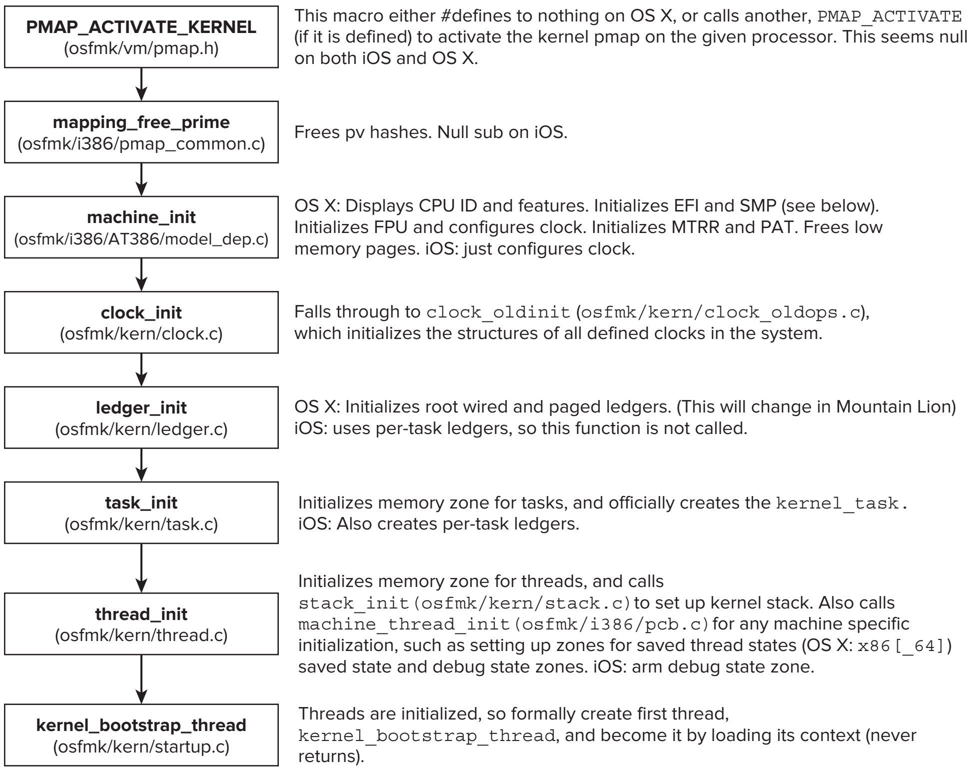


FIGURE 9-3: The flow of kernel_bootstrap (from osmfk/kern/startup.c) *(continued)*

machine_init

Just before the Mach primitives are initialized, `kernel_bootstrap` calls `machine_init(osfmk/i386/AT386/model_dep.c)`, for machine specific aspects. On ARM, this call doesn't do much, aside from configure the clock. In OS X, however, this call is of paramount importance, especially in SMP (which Mac hardware is by default). Its flow is shown in Figure 9-4:

The function responsible for the SMP initialization is `smp_init`. This function is responsible for two main tasks:

- **Initialize the LAPIC**: In SMP architectures, each processor (or core) has a Local Advanced Programmable Interrupt Controller. This is responsible, at the hardware level, for interrupt delivery to the core.
- **Set the slave CPU's entry point**: This is done using a physical memory copy through `install_real_mode_bootstrap()`, because Intel CPUs and cores wake up with paging disabled. The entry point is set to `slave_pstart()`, as discussed previously.

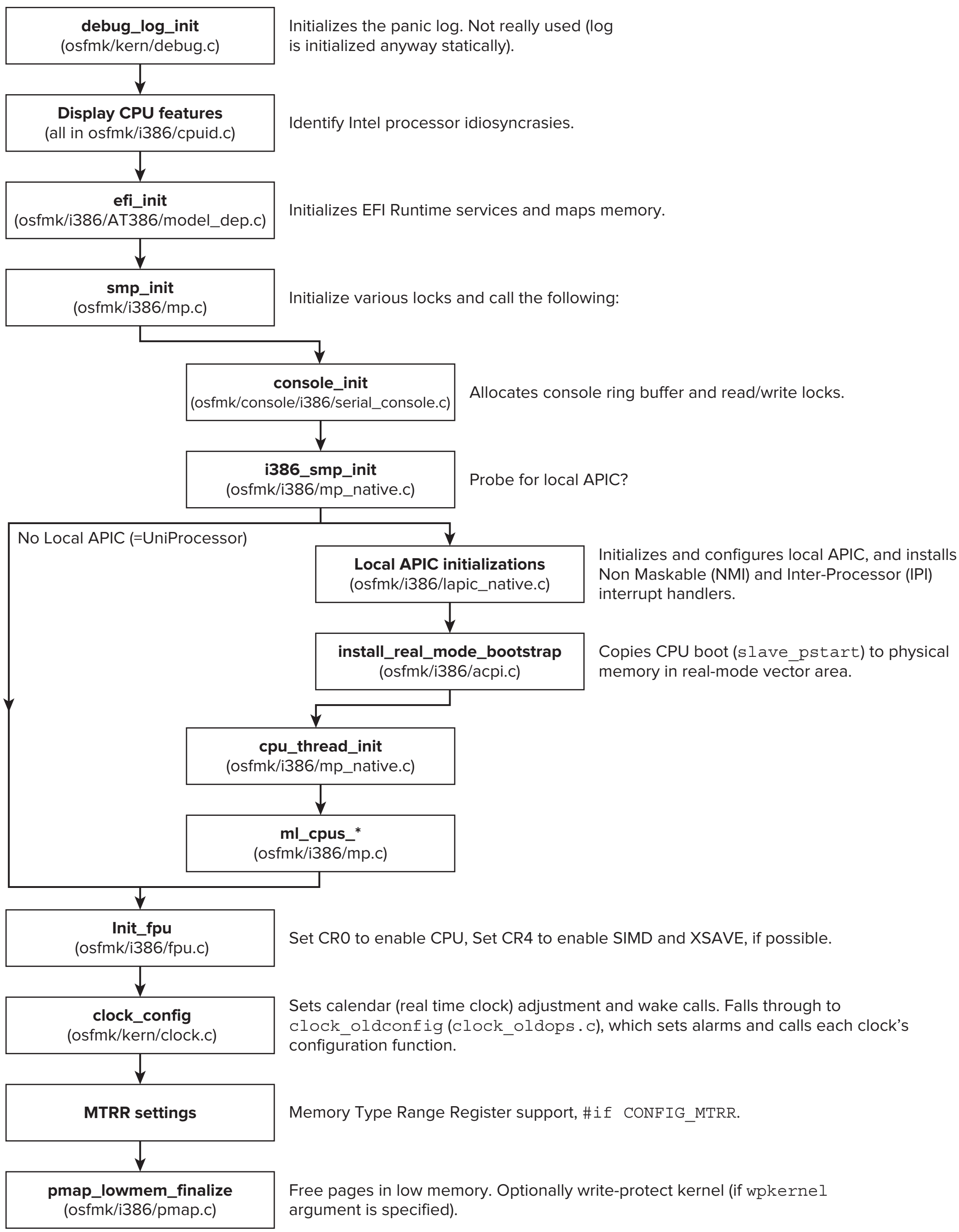


FIGURE 9-4: The flow of machine_init() on OS X

kernel_bootstrap_thread

In its new persona as the `kernel_bootstrap_thread` the main thread keeps on with its task of initializing the various subsystems, whose foundations were established in the last stage.

Now that thread support has been enabled, the `kern_bootstrap_thread` can call on `kernel_create_thread()` to spawn helper threads. Indeed, it does just that, with the very first thread created being the idle thread. This thread is necessary so that the system cores or CPUs will always have something to execute when all other threads are blocking.

Following the idle thread, the next thread started is the scheduler itself. The scheduler is described in depth later in Chapter 11. The scheduler is the task which will, at specified intervals and after interrupts, get to decide which thread gets to execute next.

After spawning a few system threads to handle thread maintenance, OS X's XNU starts a `mapping_replenish()` thread. Similar functionality is achieved on iOS by spawning a `zone_refill_thread`, though only a little bit later.

If the kernel is configured with `SERIAL_KDP` (as both OS X and iOS are), a call to `init_kdp()` next initializes the debugger. It's rather odd that Apple left KDP support in iOS: Though i-Devices come with no official serial port, their (single) connection can be made into a serial port[1], and KDP support is instrumental in letting hackers obtain a view of memory.

The next important step carried out is initializing IOKit, which is XNU's device driver framework. This is key, because without IOKit, XNU can't directly access devices: It simply has no code of its own to access even the most basic devices of the disk, display, and network.

Once IOKit is initialized, interrupts may be enabled. This is done by a call to `spllo()`, which `#defines` to `ml_enable_interrupts()`. As shown in the previous chapter, this function adapts to the underlying interrupt mechanism (Intel's IF EFLAG or ARM's Interrupt bit in CPSR).

The next module to initialize is the shared region module, which is used by clients such as `dyld(1)` when loading shared libraries, and the kernel itself in what is known as the *commpage*. The commpage is a single page that is mapped from the kernel directly to *all* processes, and contains various exported data, as well as functions. This page always resides in the same address and is accessible to all processes, as described in Chapter 4.

If the kernel is compiled with Mandatory Access Control (`CONFIG_MACF`), as both OS X and iOS are, a call to `mac_policy_initmach()` follows, which enables the policy modules to start their work as early as possible. This is crucial for maintaining system security, as otherwise various race conditions could allow attackers to attempt operations before policies come into full effect.

Once MAC is enabled, the BSD subsystem can be initialized. This is a massive function, `bsd_init()`, worthy of its own section and is detailed later. This function eventually spawns the init task, which executes `/sbin/launchd`, the progenitor of all user mode processes.

Following BSD's initialization, if the kernel was configured with the `serial` boot argument, a serial console is enabled by spawning a dedicated console listener thread. By this time, user mode processes (spawned after the BSD subsystem completes its initialization) may access the console by opening its tty. Again, somewhat surprisingly, this is enabled in iOS.

On an SMP system, the penultimate step is to enable the local page queue for each CPU. On a uniprocessor, this is skipped. Finally, with nothing else left to do, the main thread assumes a new personality for the last time — that of `vm_pageout()`, which will manage swapping for the system and is covered in Chapter 12, dealing with the Mach VM subsystem. (See Figure 9-5.)

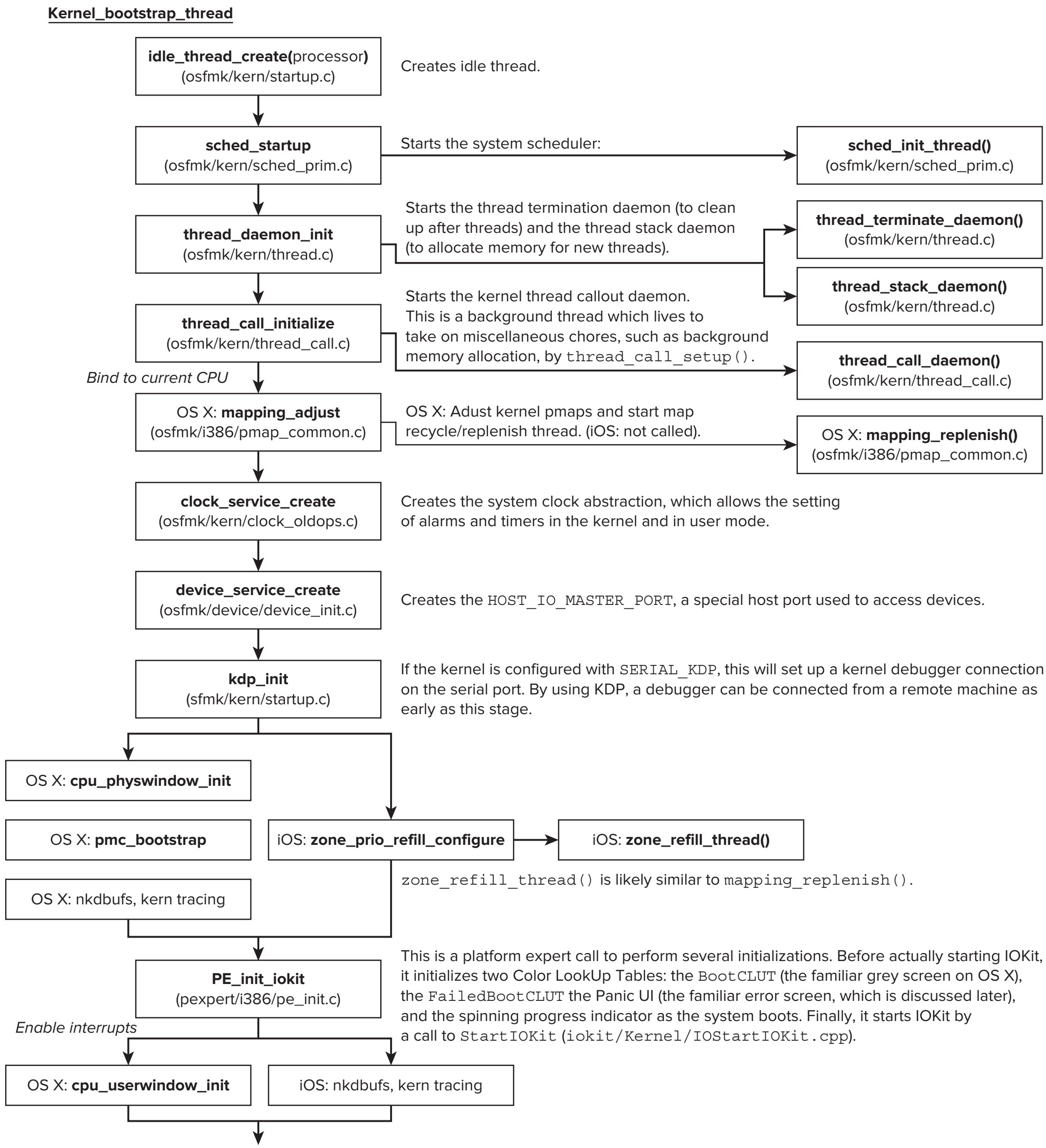


FIGURE 9-5: Flow of kernel_bootstrap_thread

continues

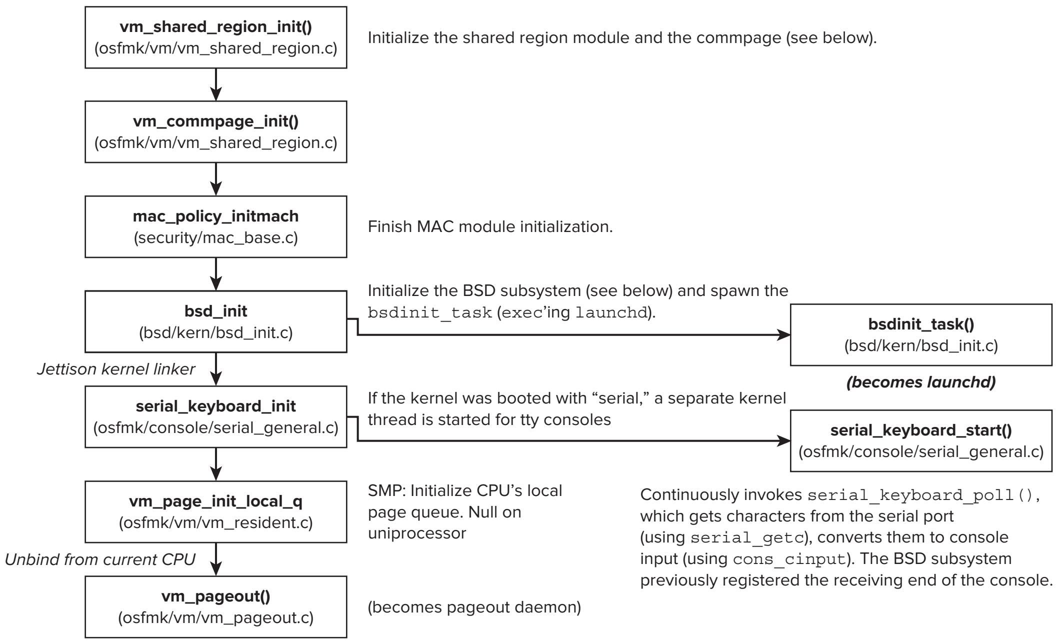


FIGURE 9-5: Flow of kernel_bootstrap_thread *(continued)*

bsd_init

The entire setup of the BSD layer of XNU is performed by a single function called (unsurprisingly) `bsd_init()`, in the similarly named `bsd/kern/bsd_init.c`. This function call is enclosed in an `#ifdef MACH_BSD`, which demonstrates just how decoupled the Mach part of XNU can be made from its BSD. In XNU, however, the two are intricately intertwined following this call.

There is a significant amount of work which follows. Most of it is performed by self-contained `*_init()` functions, to initialize the various subsystems, each in turn. Most of the functions take no arguments. This (and a panic or two) makes it relatively easy to pick out of iOS's long disassembly. Because this function is the fulcrum of all of the BSD subsystem, the rest of the disassembly falls like a string of dominoes, as shown in Listing 9-4, which has been partially annotated:

LISTING 9-4: Partial disassembly of bsd_init() of an iPhone 4S memory image

```
                ...
0x802B710E   LDR   R0, "bsd_init: Failed to create execve"...
0x802B7110   BL    _panic
0x802B7114   B     802B711A     ; Normal boot obviously skips over the panic
0x802B7116   BL    _bsd_bufferinit
0x802B711A   BL    sub_802040AC ; IOKitInitializeTime
0x802B711E   MOVS  R6, #0
0x802B7120   BL    sub_802B7D7C ; ubc_init
0x802B7124   BL    sub_801E2070 ; devsw_init
0x802B7128   BL    sub_802B5DE4 ; vfsinit
0x802B712C   BL    sub_801AF7F4 ; mcache_init
0x802B7130   BL    sub_801BE110 ; mbinit
```

```
0x802B7134   BL    sub_800D858C ; net_str_id_init
0x802B7138   BL    sub_802B7740 ; knote_init
0x802B713C   BL    sub_802B74E8 ; aio_init
0x802B7140   BL    sub_801B5320 ; pipeinit
0x802B7144   BL    sub_801D24D4 ; pshm_lock_init
0x802B7148   BL    sub_801D1AB0 ; psem_lock_init
0x802B714C   BL    sub_801DBC0C ; pthread_init
0x802B7150   BL    sub_802B8174 ; pshm_cache_init
0x802B7154   BL    sub_802B814C ; psem_cache_init
0x802B7158   BL    sub_802B7D28 ; time_zone_slock_init
0x802B715C   BL    sub_801B2410 ; select_wait_queue_init
0x802B7160   BL    sub_802B74B8 ; stackshot_lock_init
0x802B7164   BL    sub_801ABEAC ; sysctl_register_fixed
0x802B7168   BL    sub_802B7B84 ; sysctl_mib_init
0x802B716C   BL    sub_800C8A04 ; dlil_init
0x802B7170   BL    sub_802B63A8 ; protocol_kpi_init
0x802B7174   BL    sub_802B7FFC ; socketinit
0x802B7178   BL    sub_802B7EB8 ; domaininit
0x802B717C   BL    sub_800FC040 ; iptap_init
```

You can follow the flow along in Figure 9-6. Note that, unlike the previous figure, this does not point out the threads spawned by the functions, even though quite a few do so.

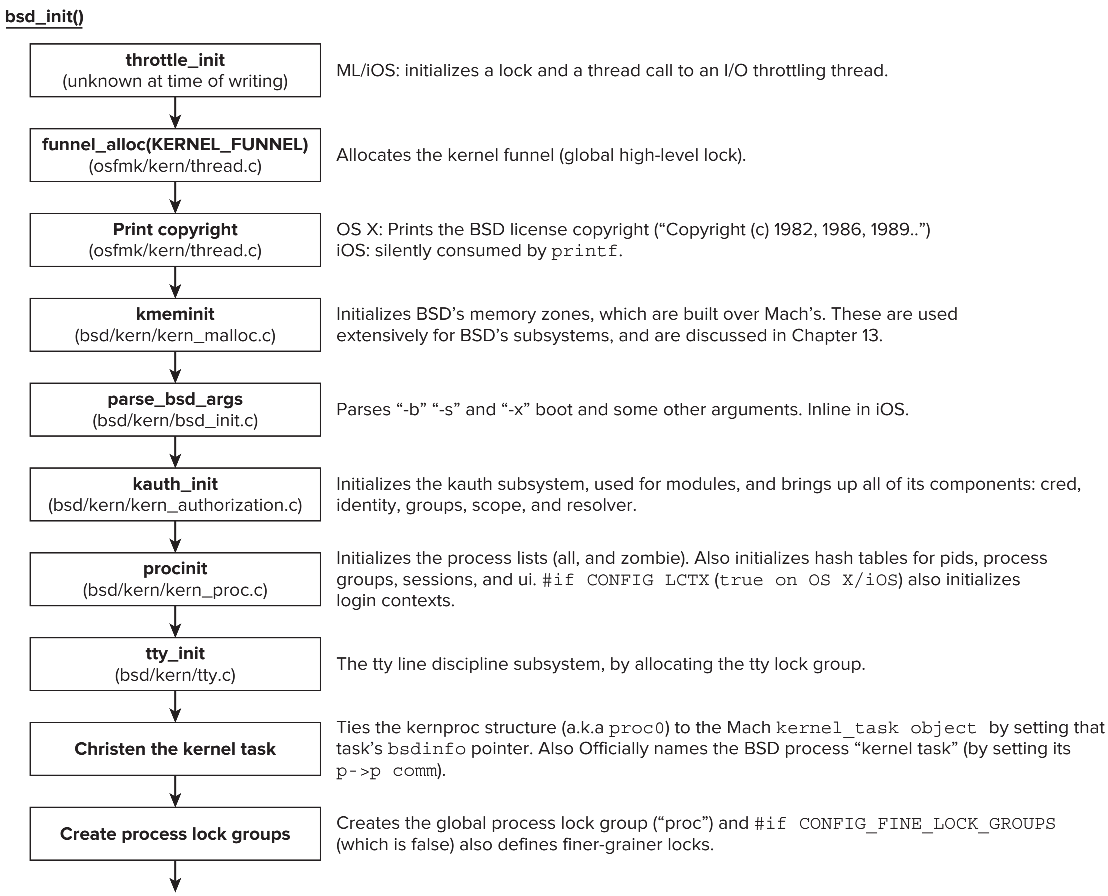


FIGURE 9-6: The flow of bsd_init()

continues

Set execarg limits	Sets limits on `exec()` args, and the maximum size of the execargs cache as a function of how many simultaneous `exec()` calls are allowed.
...	
mac_policy_initbsd (security/mac_base.c)	Enables the BSD portion of the MAC framework policies and auto-exempt the kernel process from it. On Intel, a call to `check policy init()` validates the policy was initialized correctly.
Create Process 0	Turns the kernel process into a full BSD process, with PID 0. Initializes various fields of kernproc to reflect the code signing validity and other settings of the process.
Kauth credentials	Allocates credentials for the kernel using `kauth_cred_create()`, and then calls `kauth_cred_ref()` to increment the reference count. Both functions are from `bsd/kern/kern_credential.c`.
file_lock_init (bsd/kern/kern_descrip.c)	Initialize the "file" lock group, which contains the uipc lock and `file_flist_lock`.
MAC label assignment	Creates the process resource limits table (used by `ulimit(1)` and `get/setrlimit(2)`). This is inherited later by all subprocesses.
Associate rlimit table	`mac_cred_label_associate_kernel()` and `mac_task_label_update_cred()` to update the kauth credentials previously created and tie them to the Mandatory Access Control framework.
Associate file descriptor table	Ties `fileproc0`, the master file descriptor table, to kernproc's `p_fd`, and initialize some of its fields.
Associate sigacts and stats	Ties `pstats0` and `sigacts0` to `p_stats` and `p_sigacts` fields of `kernproc`.
chgproccnt(0, 1) (bsd/kern/kern_proc.c)	"Charges" root's process quota for two processes (0, the kernproc, and 1, the `bsdinit_task` to launch, once called `mach_init`, and nowadays called `launchd`).
Allocate execve submap	Allocates a kernel page able submap which can be used during `execve()`. This is required because an `execve()` will be needed soon to spawn PID 1.
bsd_bufferinit (bsd/dev/unix_startup.c)	Allocates buffers for most BSD subsystems, such as `vnodes`, network protocols. Called `bsd_startupearly(bsd/dev/unix_startup.c)` and ends with `bufinit (bsd/vfs/vfs_bio.c)`, which also initializes lists and hashes
IOKitInitializeTime (iokit/Kernel/IOStartIOKit.cpp)	Waits until IOKit's IORTC (real time clock) arrives (and, on OS X, also IONVRAM, for NVRAM support), and initializes the system time (or calendar, in Mach parlance) to support `gettimeofday()` functions.
ubc_init (bsd/kern/ubc_subr.c)	Initializes the Unified Buffer Cache, which is the BSD layer's block buffering mechanism, and is used to speed up file and block device I/O.
devsw_init (bsd/kern/bsd_stubs.c)	Initializes the BSD device switch lock group.
vfsinit (bsd/vfs/vfs_init.c)	Initializes the Virtual Filesystem Switch, which is the BSD layer's unified interface for file systems (Chapter 14).

FIGURE 9-6: The flow of bsd_init() *(continued)*

mcache_init (bsd/kern/mcache.c)	Initializes the BSD mcache mechanism, which is an efficient allocator with individual CPU cache optimizations (discussed in chapter 13).
mbinit (bsd/kern/uipc_mbuf.c)	Initializes MBufs, which are the data buffers used by the network stack.
net_str_id_init (bsd/net/net_str_id.c)	Allocates a lock for `net_str`, which is used in MBuf tag allocation and looking up strings associated with network kernel extensions (NKEs).
OS X: **audit_init** (bsd/security/audit/audit.c)	`#if CONFIG_AUDIT` (which is true in OS X, but not iOS), this brings up the audit subsystem. Auditing is discussed in Chapter 3.
knote_init (bsd/kern/kern_event.c)	Allocates the memory zone ("knote zone") to be used by up to 8192 kernel events, and the required lock groups for them. Sets up the kqueue lock, and timer filter lock.
aio_init (bsd/kern/kern_aio.c)	Initializes the asynchronous I/O locks: `aio_proc`, `aio_entry`, and `aio_queue`. Also initializes `AIO_NUM_WORK_QUEUES` (currently, 1) work queue, and `CONFIG_AIO_THREAD_COUNT` worker threads.
pipeinit (bsd/kern/sys_pipe.c)	Allocates a lock group for PThreads. If PSYNCH is defined, also a workgroup cleanup thread callout, and a zone for psynch (discussed in Chapter 13).
Sys V Sem/Shm/Msg lock Init	Depending on `SYSV_SHM`, `SYSV_MEM`, and `SYSV_MSG`, the locks for the System V APIs are initialized here. This holds in OS X, but not in iOS.
POSIX Sem/Shm lock Init	POSIX Semaphores and Shared memory locks, by contrast supported on both OS X and iOS, are initialized here, as a prerequisites for POSIX threads.
pthread_init (bsd/kern/pthread_synch.c)	Allocates a lock group for PThreads. If PSYNCH is defined, also a workgroup cleanup thread callout, and a zone for psynch (discussed in Chapter 13).
POSIXSem/Shm cache Init (bsd/kern/posix_[sem|shm].c)	Initialize hash tables for POSIX semaphores and shared memory.
time_zone_slock_init (bsd/kern/kern_time.c)	Allocates the time zone spin lock.
select_wait_queue_init (bsd/kern/sys_generic.c)	Falls through to `wait_queue_init (&select_conflict_queue, SYNC_POLICY_FIFO);`.
stackshot_lock_init (bsd/kern/kdebug.c)	Obtaining a stackshot (discussed in Chapter 5) requires a lock over the processes, so they don't get modified during the process of the stackshot.
Sysctl registration (...c)	A call to `sysctl_register_fixed (bsd/kern/kern_newsysctl.c)` registers the top level `sysctl(8)` namespaces, followed by a call to `sysctl_mib_init(bsd/kern/kern_mib.c)` to populate the `hw.*` MIBs.
dlil_init (bsd/net/dlil.c)	Initializes the Data Link Interface Layer (DLIL), which is the support for layer II protocols, such as Ethernet. Parses several network related boot args, initializes various zones, spawns the dlil `input_thread` for the loopback interface, and potentially initializes PF, if defined.
proto_kpi_init (bsd/net/kpi_protocol.c)	Allocates the locks used by the kernel programming interface which enables access to registered network protocols from within the kernel.

continues

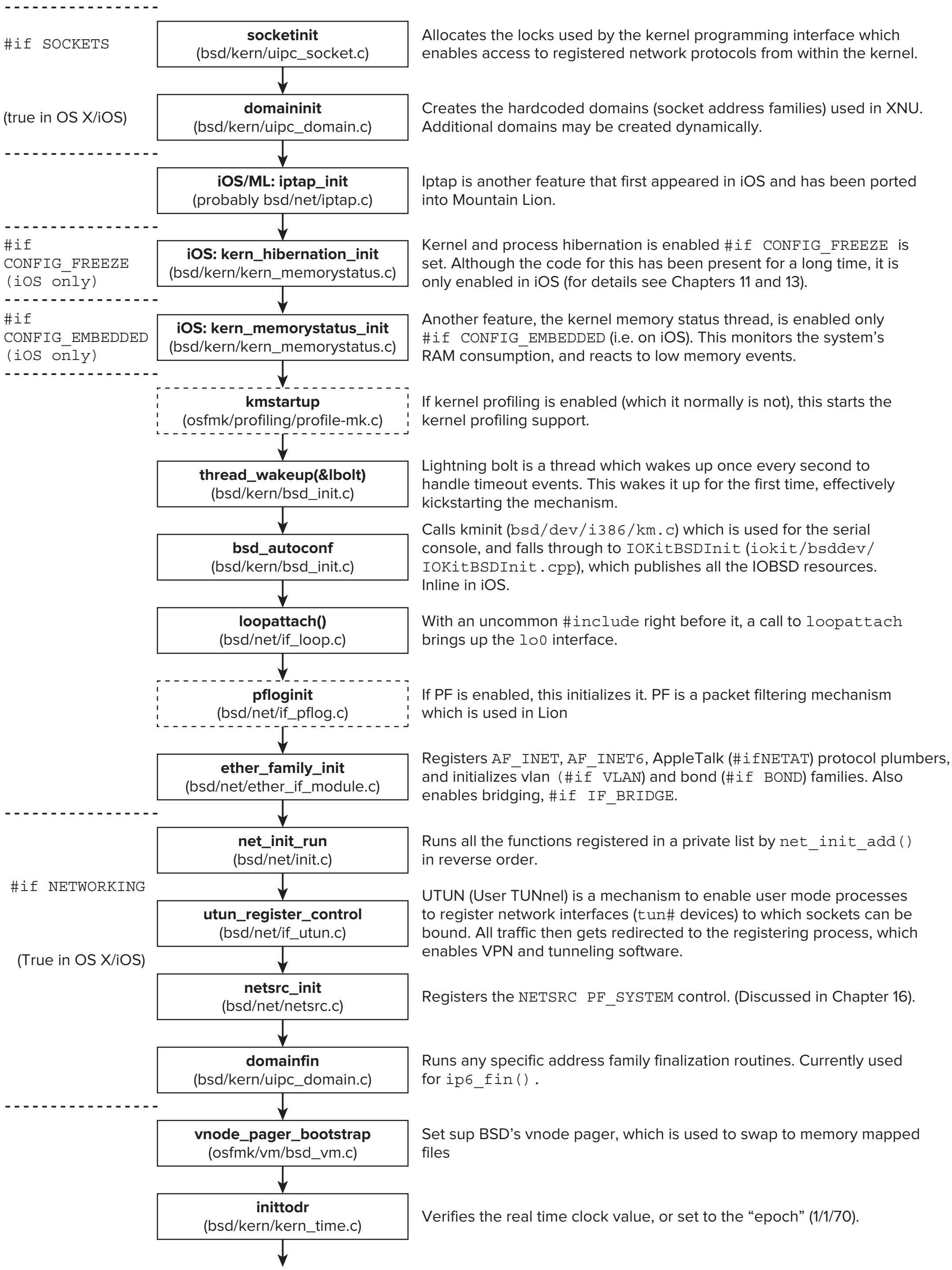


FIGURE 9-6: The flow of bsd_init() *(continued)*

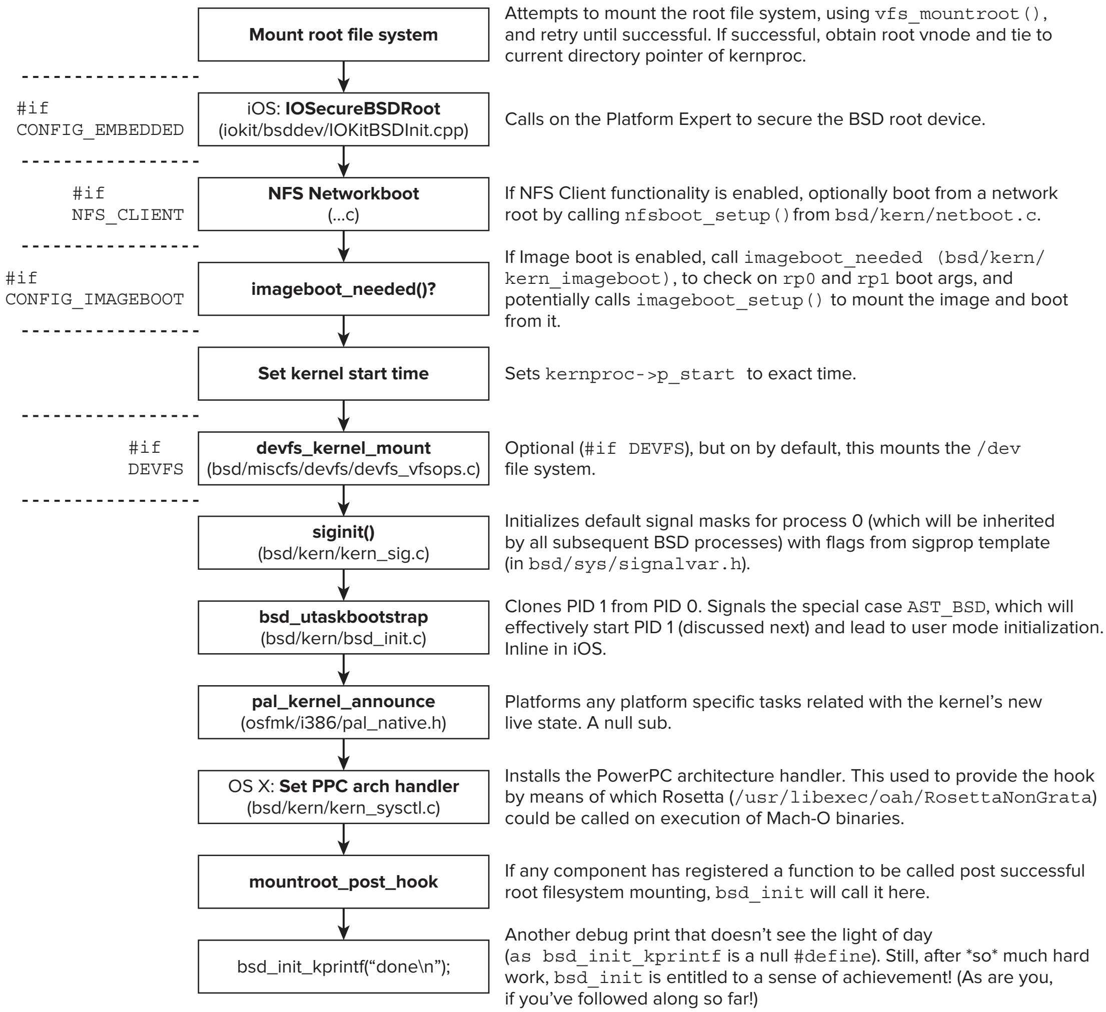


FIGURE 9-6: The flow of bsd_init()

bsdinit_task

Towards the end of its execution, `bsd_init()` makes a call to `bsd_utaskbootstrap()`. This function is responsible indirectly for starting PID 1, which is the first task to emerge into user mode. To do so, it first makes a call to `cloneproc()`, which creates a new Mach task. But from here to user mode the road is long.

To actually spin off the new task, `utaskbootstrap()` generates an asynchronous system trap (AST) by calling `act_set_astbsd()` on the newly created thread. ASTs are covered in Chapter 11, dealing with Mach scheduling, but in the interim suffice it to say that they are scheduling events, which in this case will result in the init task executing: The call followed by a call to `thread_resume()`

on it, and then `utaskbootstrap()` returns to `bsd_init()`. When the AST is processed, the Mach AST handler will specifically handle this special case, by calling `bsd_ast()` (from `bsd/kern/kern_sig.c`), which in turn calls `bsdinit_task()`. This function is shown in Listing 9-5:

LISTING 9-5: bsdinit_task() (from bsd/kern/bsd_init.c)

```
bsdinit_task(void)
{
        proc_t p = current_proc();
        struct uthread *ut;
        thread_t thread;

        process_name("init", p);

        ux_handler_init();

        thread = current_thread();
        (void) host_set_exception_ports(host_priv_self(),
                         EXC_MASK_ALL & ~(EXC_MASK_RPC_ALERT),//pilotfish (shark) ..
                        (mach_port_t) ux_exception_port,
                                      EXCEPTION_DEFAULT| MACH_EXCEPTION_CODES,
                                      0);

        ut = (uthread_t)get_bsdthread_info(thread);

        bsd_init_task = get_threadtask(thread);
        init_task_failure_data[0] = 0;

#if CONFIG_MACF
        mac_cred_label_associate_user(p->p_ucred);
        mac_task_label_update_cred (p->p_ucred, (struct task *) p->task);
#endif
        load_init_program(p);
        lock_trace = 1;
}
```

The `bsdinit_task()` sets the initial process name to `init`, true to its UNIX origins. This is nothing more than a simple memcpy to the `proc_t`'s `comm` field. Next, a call to `ux_handler_init()`. This creates a separate kernel thread, `ux_handler`, which is responsible for handling UNIX exceptions — i.e. receiving messages on a global `ux_exception_port`. What follows is a registration of the init thread's exception port, to register this global port as its own. This, as is discussed in Chapter 12 (under "Exceptions"), ensures that all UNIX exceptions of init — and therefore all UNIX processes (its descendants) — are handled by this thread. Finally, it calls `load_init_program()`.

`load_init_program()` (shown in Listing 9-6) is responsible for turning PID 1 into the well-known `launchd`. To do so, it first manually sets up `argv[]`, in user memory. The `argv[0]` is set to `init_program_name`, a 128-byte array hardcoded to `/sbin/launchd`. Optionally, if the kernel was booted with `-s` (which results in the `boothowto` global variable flagging `RB_SINGLE`), the same `-s` is propagated to `launchd`.

Once `argv[]` is set up, `launchd` is started by a standard call to `execve()`. Since this call is expected to never return, if it does, the exec has failed. The code that follows it, therefore, is a kernel panic. With this, the path this thread takes is all in user mode, and is discussed in Chapter 5.

LISTING 9-6: load_init_program (from bsd/kern/kern_exec.c)

```
// Note that launchd's path is hard-coded right into the kernel.
// This was "/mach_init" up to OS X 10.3

static char             init_program_name[128] = "/sbin/launchd";
struct execve_args      init_exec_args;

/*
 * load_init_program
 *
 * Description: Load the "init" program; in most cases, this will be "launchd"
 *
 * Parameters:  p                   Process to call execve() to create
 *                                  the "init" program
 *
 * Returns:     (void)
 *
 * Notes:     The process that is passed in is the first manufactured
 *            process on the system, and gets here via bsd_ast() firing
 *            for the first time.  This is done to ensure that bsd_init()
 *            has run to completion.
 */

void load_init_program(proc_t p)
{
     vm_offset_t     init_addr;
     int             argc = 0;
     uint32_t argv[3];
     int                     error;
     int             retval[2];

     /*
      * Copy out program name.
      */

     init_addr = VM_MIN_ADDRESS;
     (void)vm_allocate(current_map(),&init_addr,PAGE_SIZE,VM_FLAGS_ANYWHERE);
      if (init_addr == 0)
            init_addr++;

     (void) copyout((caddr_t) init_program_name, CAST_USER_ADDR_T(init_addr),
                        (unsigned) sizeof(init_program_name)+1);

     argv[argc++] = (uint32_t)init_addr;
     init_addr += sizeof(init_program_name);
     init_addr = (vm_offset_t)ROUND_PTR(char, init_addr);

     /*
```

continues

LISTING 9-6 *(continued)*

```
        * Put out first (and only) argument, similarly.
        * Assumes everything fits in a page as allocated
        * above.
        */
        if (boothowto & RB_SINGLE) {
               const char *init_args = "-s";

               copyout(init_args, CAST_USER_ADDR_T(init_addr),
                           strlen(init_args));

               argv[argc++] = (uint32_t)init_addr;
               init_addr += strlen(init_args);
               init_addr = (vm_offset_t)ROUND_PTR(char, init_addr);

        }
        /*
         * Null-end the argument list
         */
        argv[argc] = 0;
        /*
         * Copy out the argument list.
         */

        (void) copyout((caddr_t) argv, CAST_USER_ADDR_T(init_addr),
                        (unsigned) sizeof(argv));

        /*
         * Set up argument block for fake call to execve.
         */

        init_exec_args.fname = CAST_USER_ADDR_T(argv[0]);
        init_exec_args.argp = CAST_USER_ADDR_T((char **)init_addr);
        init_exec_args.envp = CAST_USER_ADDR_T(0);

        /*
         * So that mach_init task is set with uid,gid 0 token
         */
        set_security_token(p);

        error = execve(p,&init_exec_args,retval);
        if (error)
                panic("Process 1 exec of %s failed, errno %d",
                      init_program_name, error);
}
```

Sleeping and Waking Up

Any laptop owner no doubt appreciates OS X's ability to sleep. This ability is even more important for i-Devices, wherein power consumption must be minimized, while at the same time maintaining the "always-on" experience.

The iOS sleeping and hibernation mechanisms are, at the time of writing, not entirely figured out: Most of the work there, as in OS X, is done by an external kernel extension (OS X's AppleACPI).

In OS X, XNU's portion of the sleep and hibernation code is open source, but the Kext's part isn't. The kernel can be put to sleep by a call from the Kext by `acpi_sleep_kernel()`. The `AppleACPIPlatform.Kext` uses this call. It proceeds as follows:

- All CPUs but the current one are halted. This is done by calling `pmCPUExitHaltToOff()`, which is a wrapper over a corresponding function from a dispatch table. The kernel does not have an implementation for this, and relies on a specialized Kext (`AppleIntelCPUPowerManagement.Kext`) to call `pmKextRegister` with the dispatch table (defined as a `pmDispatch_t` in `osfmk/i386/pmCPU.h`).
- The local APIC is shut down, in preparation for sleep.
- A kdebug message is output.
- CR3 is saved on x86_64.
- A call to `acpi_sleep_cpu` (in `osfmk/x86_64/start.s`) puts the CPU to sleep. This saves all the registers, and calls a caller supplied callback function (from the calling Kext) to put CPU to sleep. In case of hibernation, `acpi_hibernate` is called instead, which first writes the memory image to disk.
- Control is passed back to the firmware.

`AppleACPIPlatform.Kext` can also request the installation of a wake handler. This is done by a call to `acpi_install_wake_handler` (also in `osfmk/i386/acpi.c`), which uses `install_real_mode_handler` (encountered previously in the discussion of slave processors). The wake handler is `acpi_wake_prot`, an assembly function from `osfmk/x86_64/start.s`. `acpi_wake_prot`, which performs the following actions:

- Switches back to 64-bit mode
- Restores kernel GDT, CR0 , LDT and IDT, and task register
- Restores all saved registers (by `acpi_sleep_cpu()`)

When the function returns, it does so into `sleep_kernel()`,right after the call `acpi_sleep_cpu()`. Think of it as one really long function call, but it eventually does return. The rest of `sleep_kernel()` basically undoes all of the sleep steps, in reverse order. Finally, it calls `install_real_mode_bootstrap()`, to once again set `slave_pstart()` as the slave CPUs' activation function.

BOOT ARGUMENTS

XNU has quite a few boot arguments, but Apple really doesn't bother documenting them. Nor is there any particular naming convention - some use a hyphen (-), whereas others do not.

There are generally two ways to pass arguments to the kernel:

- Via the NVRAM using the `boot-args` variable (which can be set using the `nvram` command.

- Via `/Library/Preferences/SystemConfiguration/com.apple.Boot.plist`. This is a standard Property List file, in which you can specify arguments in a `kernel_flags` element.

> *In iOS, iBoot has long been modified so as to not pass boot arguments to XNU. Jailbreaking utilities (such as redsn0w) enable passing argument strings to the kernel, but only in a tethered boot.*

Table 9-7 lists some useful kernel boot arguments of Mac OS X, sorted by a rough alphabetical order:

TABLE 9-7: XNU Boot Arguments

ARGUMENT	HANDLED BY	USED FOR
`-l`	`kernel_bootstrap`	Leaking logging
`-s`	`parse_bsd_args` `bsd/kern/bsd_init.c`	Single user mode (`boothowto |= RB_SINGLE`)
`-b`	`parse_bsd_args` `bsd/kern/bsd_init.c`	Bypassing the boot RC (`boothowto |= RB_NOBOOTRC`)
`-x`	`parse_bsd_args` `bsd/kern/bsd_init.c`	Safe booting (`boothowto |= RB_SAFEBOOT`)
`-disable_aslr`	`parse_bsd_args` `bsd/kern/bsd_init.c`	Randomizing address space layout. May only be disabled if `DEVELOPMENT` or `DEBUG` are `#defined`
`-no_shared_cr3`	`pmap_bootstrap` `(osfmk/x86_64/pmap.c)`	Forcing a kernel to reside in its own address space and not piggybacked on processes. Useful only for some minor debugging
`-no64exec`	`parse_bsd_args` `bsd/kern/bsd_init.c`	Forcing 32-bit mode `Bootarg_no64_exec = 1`
`-kernel_text_ps_4K`	`pmap_lowmem_finalize`	Kernel to be allocated with 4 KB, rather than 2 MB pages
`-zc` `-zp` `-zinfop` `zlog` `zrecs`	`zone_init` `osfmk/kern/zalloc.c`	Mach zone debugging. Described in more detail in Chapter 12
`cpus`	`i386_init` `osfmk/i386/i386_init.c`	Artificially limiting how many CPUs to use

ARGUMENT	HANDLED BY	USED FOR
debug	machine_startup (osfmk/i386/AT386/model_dep.c)	Debug mode. See "Kernel Debugging" later in this chapter
diag	osfmk/i386/i386_init.c	dgWork.dgFlags global variable for enabling diagnostic system calls
himemory_mode	osfmk/i386/i386_init.c	Toggling High memory mode — debugging on systems with more than 4 GB of physical memory
io iotrace	StartIOKit (iokit/Kernel/IOStartIOKit.cpp)	Setting the gIOKitDebug and gIOKitTrace flags, respectively (and gIOKitTrace actually imports flags from gIOKitDebug)
kextlog	OSKext::initialize (libkern/c++/OSKext.cpp)	Setting the sKernelLogFilter mask, which is used for kext logging. Discussed in Chapter 18
kmem	parse_bsd_args	Enabling /dev/kmem. Not available if SECURE_KERNEL is #defined. Naturally, not available on iOS
maxmem	i386_init (osfmk/i386/i386_init.c)	Artificially limiting how much physical memory to use, in MB
msgbuf	parse_bsd_args	Adjusting the size of kernel ring buffer (shown by dmesg(1) command)
novfscache	parse_bsd_args	Disabling the VFS cache
policy_check	parse_bsd_args	Setting policy check flags if CONFIG_MACF is defined.
serial	i386_init osfmk/i386/i386_init.c	Setting serial mode — serial keyboard/console. Depending on this argument, serialbaud (in pexpert/i386/pe_serial.c) can set the serial baud rate
serverperf-mode	kernel_bootstrap	Setting server performance mode
wpkernel	pmap_lowmem_finalize	Writing protect kernel region

Additional arguments can be defined by kext subsystems, such as the Kernel Debugger Protocol (KDP), and the virtual memory zone allocator (osfmk/kern/zalloc.c) discussed in Chapter 12. Kexts can likewise parse the argument string (by calling PE_parse_boot_argn) to obtain private arguments. A good example for this is iOS's AppleMobileFileIntegrity — a key component trusted with code signing entitlements, whose arguments are discussed in Chapter 14.

KERNEL DEBUGGING

The kernel allows remote debugging using the KDP protocol. This is a simple protocol, carried over UDP, which is used by XNU for debugging and/or core dump generation. The client is the debugged system, and the server is some other (hopefully more stable) system. Table 9-8 shows the boot arguments used by KDP:

TABLE 9-8: Arguments Parsed by kdp_register_send_receive() in osfmk/kdp/kdp_udp.c

ARGUMENT	TOGGLES/ENABLES
`debug`	Bit-flags specifying debugging options. See Table 9-9.
`_panicd_ip`	IP address of remote `PanicD`.
`_router_ip`	IP address of router.
`_panicd_port`	UDP port number of remote `PanicD`.
`_panicd_corename`	Core file on remote `PanicD`.

The arguments in the preceding table are used in conjunction with `kdp_match_name` (which can be set to `serial`, `en0`, `en1`, and so on) to set up the kernel debug protocol.

In order to trace kernel extensions (kexts) and their debug/log messages, the `Kextlog` boot-arg can be used. This is a bitmask argument, which controls the kernel's built-in filtering mechanisms, much like Windows' `DebugPrintFilter` does for its `DbgPrint`. The argument can also be changed at run-time, via `sysctl(8)` as `debug.Kextlog`. This is discussed in great detail under "Kext Logging," in Chapter 18, which is devoted exclusively to kexts.

To enable full kernel debugging, the system must be booted with `debug`. The kernel debug flags are specified in TN2118[2] ("Kernel Core Dumps") and in the Kernel Programming Guide[3], as shown in Table 9-9.

TABLE 9-9: Flag Values of the debug Boot Aargument and Their Meanings

FLAG	VALUE	MEANING
`DB_HALT`	`0x01`	Halt boot, waiting for debugger to attach.
`DB_PRT`	`0x02`	Redirect `printf()`s in kernel to console.
`DB_NMI`	`0x04`	Allow dropping immediately into the kernel debugger on the command-power key sequence, or by holding together Command+Option+Ctrl+Shift+Esc.
`DB_KPRT`	`0x08`	Redirect `kprintf()`s in kernel to serial port, if defined.
`DB_KDB`	`0x10`	Sets KDB as the current debugger.

FLAG	VALUE	MEANING
`DB_SLOG`	`0x20`	Outputs diagnostics to system log.
`DB_ARP`	`0x40`	Allows ARP in KDP.
`DB_LOG_PI_SCRN`	`0x0100`	Disables Panic dialog. This is useful when core dumps are generated, as it will show instead the progress of sending the core.
`DB_KERN_DUMP_ON_PANIC`	`0x0400`	Core dumps on panic — handled by `kdp_panic_dump()` in `kdp.c`.
`DB_KERN_DUMP_ON_NMI`	`0x0800`	Core dumps on an NMI, but not crash. If `DB_DBG_POST_CORE (0x1000)` is additionally set, kernel will wait for debugger attachment.
`DB_PANICLOG_DUMP`	`0x2000`	Only shows panic log on dump, not full core.

Heisenberg's Uncertainty Principle makes live kernel debugging on the same machine impossible. The debugger is, therefore, a different machine than the debuggee and normally requires a serial port, Ethernet, or FireWire connection. In OS X, the `fwkpfv(1)` command may be used to direct `kprintf()`s over FireWire. Another tool, `fwkdp(1)`, may be used to enable KDP over FireWire.

VMWare makes debugging immeasurably easier, by enabling the debuggee to be in a virtual machine (OS X is not VM-friendly, but can be cajoled — or coerced, on non-Apple architectures — into it). The host debugger can attach using the kdp-reattach macro from the Kernel Debug Kit's `kgmacros`. This requires setting up a static ARP entry for the debuggee's IP, but is a fairly straightforward process. If the VM is booted with DB_HALT (`nvram boot-args="debug=0x01"`), it will halt until the debugger attaches. VMWare has its own built-in support, and the process of using it, or KDP, is well documented[4].

"Don't Panic"

As Mac users know, every now and then the operating system itself may unexpectedly halt, due to an instability in the kernel mode. Linux simply dumps everything in black and white on the console, Windows favors EGA blue, while Mac OS X prefers grey alpha-blending. This "Gray Screen of Death" is the all-too-familiar result of the kernel calling the internal `panic()` routine. This routine, which displays the unexpected shutdown message and halts the CPU, does so very rarely, and only in cases where a system halt is the least worst option, preferable to possible serious data corruption. This generally happens in two cases:

- The kernel code path reaches some unexpected location, like the `default:` clause of a `switch()` statement that otherwise handled all known conditions. For example, the HFS+ code (in `bsd/hfs`) contains calls to `panic()` on every possible file system data structure inconsistency.

- An unhandled exception or trap occurs in kernel mode, causing the kernel trap handler (`kernel_trap in osfmk/i386/trap.c`) to be invoked for a kernel mode thread and reach an unhandled code path. The kernel trap handler then, for lack of any other option, calls `panic_trap()`. This function `kprintf()`s a message, and calls `panic()` from `kern/debug.c`. It, in turn, calls `Debugger()` (from `i386/AT386/model_dep.c`), which draws the familiar dialog using a call to `draw_panic_dialog()`.

Panics shouldn't happen, period. The kernel, as the underlying foundation of the entire operating system, must be solid and reliable. When panics do occur, usually they can be traced to a faulty driver (i.e. a kext). Very rarely, however, they arise from a bug in the kernel itself. These bugs are, one hopes, fixed as future versions of the kernel are released.

Manually Triggering a Panic

Whether for testing purposes or for debugging, OS X has several options for manually triggering a panic:

- Triggering a panic with `DTrace`: `dtrace -w -n "BEGIN{ panic();}"`. The "-w" (destructive probes) switch of DTrace is required, as a panic is certainly considered destructive.
- A kernel extension to automatically trigger a panic, downloadable as part of TN2118 ("Kernel Core Dumps").
- A "fake" panic, by calling `sysctl`.

The safest option for simulating panics is the third — merely testing the panic UI, by means of a `sysctl`. This is shown in the experiment — Viewing the Panic UI — later in this chapter.

Implementation of Panic

The kernel code to generate a panic is in the Mach core, in `osfmk/console`. Table 9-10 lists the files dealing with panics.

TABLE 9-10: Files in osfmk/ Related to Panics

FILE	CONTAINS
`panic_dialog.c`	Main file for panic dialog generation
`panic_image.c`	The pixel map containing the familiar image displayed on panic
`panic_ui/genimage.c`	A C image generator — converts from raw bitmap to C struct `panicimage`
`panic_ui/qtif2raw.c`	Converts image from QuickTime 256-color to raw bitmap
`panic_ui/setupdialog.c`	Alternate binary to perform both `genimage` and `qtif2raw`

The functions in these files are not exported to user mode for obvious reasons, but there is also a way to simulate a panic, as the following experiment shows.

Experiment: Viewing the Panic UI

The code in `bsd/kern/kern_panicinfo.c` defines the following:

```
#define KERN_PANICINFO_TEST     (KERN_PANICINFO_IMAGE+2)
  /* Allow the panic UI to be tested by root without causing a panic */

static int sysctl_dopanicinfo SYSCTL_HANDLER_ARGS
{
  ..
case KERN_PANICINFO_TEST:

                panic_dialog_test();
                break;

}
```

The `panic_dialog_test` is implemented in `osfmk/console/panic_dialog.c`:, as shown in Listing 9-7:

LISTING 9-7: panic_dialog_test, from osfmk/console/panic_dialog.c

```
void panic_dialog_test( void )
{
        boolean_t o_panicDialogDrawn = panicDialogDrawn;
        boolean_t o_panicDialogDesired = panicDialogDesired;
        unsigned int o_logPanicDataToScreen = logPanicDataToScreen;
        unsigned long o_panic_caller = panic_caller;
        unsigned int o_panicDebugging = panicDebugging;

        panicDebugging = TRUE;
        panic_caller = (unsigned long)(char *)__builtin_return_address(0);
        logPanicDataToScreen = FALSE;
        panicDialogDesired = TRUE;
        panicDialogDrawn = FALSE;

        draw_panic_dialog();

        panicDebugging = o_panicDebugging;
        panic_caller = o_panic_caller;
        logPanicDataToScreen = o_logPanicDataToScreen;
        panicDialogDesired = o_panicDialogDesired;
        panicDialogDrawn = o_panicDialogDrawn;
}
```

To show the panic dialog test, the simple code snippet shown in Listing 9-8, run as root, would do:

LISTING 9-8: Testing a panic image (OS X only)

```
size_t len = 0;
int name[3] = { CTL_KERN, KERN_PANICINFO, KERN_PANICINFO_IMAGE + 2 };
sysctl(name, 3, NULL, (void *)&len, NULL, 0);
```

The is required because the actual constant you would be using, `KERN_PANICINFO_TEST`, is not exported from the kernel headers . If you are feeling especially adventurous, you can use the `KERN_PANICINFO sysctl` with the following:

```
int name[3] = { CTL_KERN, KERN_PANICINFO, KERN_PANICINFO_IMAGE };
```

…which will enable you to *set* a panic kernel image by using the following code snippet:

```
int len;
char *buf = /* image in kraw format */
int bufsize = /* size of the above image */
int name[3] = { CTL_KERN, KERN_PANICINFO, KERN_PANICINFO_IMAGE };
sysctl(name, 3, NULL, (void *)&len, buf, bufsize);
```

Panic Reports

When a panic occurs, there is nothing more to do but force a halt and save the data so the cause might be determined post mortem. Since the halt will likely force a power cycle (read: cold reboot), however, the data will be lost if just saved to RAM. The filesystem logic might be in a non-consistent state (and might also be the cause of the panic). This leaves the machine's NVRAM as a last resort.

The Platform Expert (specifically, `PESavePanicInfo()`) calls on the NVRAM handler to write the data to an NVRam variable — `aapl,panic-info` (defined as `kIODTNVRAMPanicInfoKey` in `iokit/IOKit/IOKitKeys.h`). The log is saved in packed form (using `packA()`, a simple algorithm in `osfmk/kern/debug.c`), which writes the 7-bit ASCII characters in the log consecutively into 8-bit bytes. This, however, requires full 8-bit values to be escaped as `%XX`, similar to URI escaping, which somewhat defeats the purpose of packing.

When the system boots next, a specialized `launchDaemon, /System/Library/CoreServices/DumpPanic`, is invoked by `launchd` (from `/System/Library/LaunchDaemons/com.apple.DumpPanic.plist`). This daemon checks the panic data in the NVRAM variable, unpacks the data, and moves it to `/Library/Logs/DiagnosticReports`. These logs are then saved using the following naming convention:

```
Kernel_YYYY-MM-DD-HHDDSS_computer_name.panic
```

The actual report is generated using a private (and, thus, undocumented) framework called `CrashReporterSupport`. In Lion, the daemon also depends on a library, `libDiagnosticMessagesClient.dylib`.

Apple's TN2063[5] details how to decipher panic logs, using `gdb` and the Kernel Debug Kit. Alternatively, you can follow the examples shown here, which rely on `otool(1)` instead. The method shown here has the advantage of being applicable on any system, without additional downloads, but would not work for panics generated by kernel extensions (kexts) without their symbols.

> *Apple's Kernel Debug Kit (available through the Mac OS X Developer Program or elsewhere on the Internet) isn't really a "kit" so much as the collection of GDB macros and a debug build of the kernel. Nonetheless, it is very useful, especially for live kernel debugging (over serial port or VM). While it greatly simplifies the process shown in the following example, it's important to understand the manual process of tracing through a panic, for times wherein the debug kit may not be available. The process described is also advantageous in that it doesn't require GDB.*

Example: 32-Bit Crash Log of an Unhandled Trap

Crashes are like snowflakes. No two are exactly the same. This is because, at the time of the crash, the internal state of the kernel is dependent on many factors. Depending on which kernel extensions have been loaded and unloaded, and which threads are active, the resulting crash dump can vary greatly. In this example, we consider an actual crash log, one of too many which occurred as this book was written. (See Output 9-1.) The next time you encounter a crash (or, if you still have a panic log in your `DiagnosticReports/` directory), you can follow along the steps described next. The output will be different, naturally, but the process is generally the same.

OUTPUT 9-1: A crash dump log

```
Sun Jul 04 08:50:33 2011
panic(cpu 1 caller 0x2aab59): Kernel trap at 0x00f9a983, type 14=page fault, registers:
CR0: 0x8001003b, CR2: 0x00000000, CR3: 0x00100000, CR4: 0x00000660
EAX: 0x00000001, EBX: 0x0c267b00, ECX: 0x01000000, EDX: 0x00000001
CR2: 0x00000000, EBP: 0x6d513bd8, ESI: 0x00000001, EDI: 0x00000000
EFL: 0x00010202, EIP: 0x00f9a983, CS:  0x00000008, DS:  0x0c260010
Error code: 0x00000000

Backtrace (CPU 1), Frame : Return Address (4 potential args on stack)
0x6d5139d8 : 0x21b510 (0x5d9514 0x6d513a0c 0x223978 0x0)
0x6d513a28 : 0x2aab59 (0x59aeec 0xf9a983 0xe 0x59b0b6)
0x6d513b08 : 0x2a09b8 (0x6d513b20 0xd4fb480 0x6d513bd8 0xf9a983)
0x6d513b18 : 0xf9a983 (0xe 0x48 0xd4f0010 0x10)
0x6d513bd8 : 0xf9e909 (0xc267b00 0x0 0x0 0x0)
0x6d513c78 : 0xf9ea1c (0xc267b00 0xe0000100 0x0 0x0)
0x6d513c98 : 0x53e815 (0xc267b00 0xa75df80 0x0 0xf9d146)
0x6d513cd8 : 0xfa60fa (0xc267b00 0xa75df80 0x0 0x3)
0x6d513d88 : 0x30aaba (0xe000004 0x20006415 0x6d513ed0 0x1)
0x6d513dc8 : 0x2fdf34 (0x6d513de8 0x3 0x6d513e18 0x5874e3)
0x6d513e18 : 0x2f29ac (0xa0bea04 0x20006415 0x6d513ed0 0x1)
```

continues

OUTPUT 9-1 *(continued)*

```
0x6d513e78 : 0x470ed0 (0x82b36a0 0x20006415 0x6d513ed0 0x6d513f50)
0x6d513e98 : 0x49cc02 (0x82b36a0 0x20006415 0x6d513ed0 0x6d513f50)
0x6d513f78 : 0x4f6075 (0x86a5d20 0x7f6dfc8 0x812acd4 0x0)
0x6d513fc8 : 0x2a144d (0x7f6dfc4 0x0 0x0 0x8d6da64)

      Kernel Extensions in backtrace (with dependencies):
         com.apple.iokit.IOStorageFamily(1.6.2)@0xf97000->0xfaefff

BSD process name corresponding to current thread: diskarbitrationd

Mac OS version:
10J869

Kernel version:
Darwin Kernel Version 10.7.0: Sat Jan 29 15:17:16 PST 2011;
 root:xnu-1504.9.37~1/RELEASE_I386
System model name: MacBookAir3,2 (Mac-2410XXXXXXxxxxx)

System uptime in nanoseconds: 218120590760858
unloaded Kexts:
com.apple.iokit.SCSITaskUserClient      2.6.5
    (addr 0x586e7000, size 0x28672) - last unloaded 212106050855061
loaded Kexts:
...
com.apple.driver.AppleMikeyHIDDriver    1.2.0
com.apple.driver.AppleHDA       1.9.9f12
com.apple.driver.AGPM   100.12.19
...
com.apple.driver.AppleMikeyDriver       1.9.9f12
```

How does one approach a panic log? In this case, because the panic is generated from an unhandled trap, the first line contains the trap number.

```
panic(cpu 1 caller 0x2aab59): Kernel trap at 0x00f9a983, type 14=page fault,...
```

The code at `0x00f9a983` generated a page fault. The panic code displays the culprit: The `com.apple.iokit.IOStorageFamily` kext, version 1.6.2, which was loaded from address `0xf97000` through `0xfaefff`. This automatically singles the problematic portion:

```
..
0x6d513b18 : 0xf9a983 (0xe 0x48 0xd4f0010 0x10)
0x6d513bd8 : 0xf9e909 (0xc267b00 0x0 0x0 0x0)
0x6d513c78 : 0xf9ea1c (0xc267b00 0xe0000100 0x0 0x0)
0x6d513c98 : 0x53e815 (0xc267b00 0xa75df80 0x0 0xf9d146)
0x6d513cd8 : 0xfa60fa (0xc267b00 0xa75df80 0x0 0x3)
..
```

Note the `0x53e815` in the preceding output. This address is in the kernel proper, not in the kext. The address is a 32-bit one, and the kernel version line identifies it as an i386 kernel. Using `otool -tV`, you can disassemble the kernel and find the line that led to the calls following it. Because this is a return address, the instruction before it should be a call instruction. Using `grep –B 1` (to show the line before the match) reveals:

```
morpheus@Ergo $ otool -tV -arch i386 /mach_kernel | grep -B 1 53e815
0053e80f     call     *0x000002e4(%eax)
0053e815     movl     0x28(%esi),%ebx
```

The closest symbol to this address is `__ZN9IOService5closeEPS_m`. The I/O Kit runtime and various drivers are C++, not C, so their names are mangled. In this case, demangling would yield `IOService::close(IOService*, unsigned long)`. We can craft a rather crude shell script to find all the symbols by employing `grep –B 1` on each address, as shown in Output 9-2:

OUTPUT 9-2: Finding and symbolicating the addresses of a panic

```
# Load all the addresses from the crash dump into a variable, say $ADDRS

$ ADDRS=`cat /Library/Logs/DiagnosticReports/\
         Kernel_2011-07-16-085033_Mes-MacBook-Air.panic  |
        grep ^0x  |
        cut -d : -f2 | cut -d' ' -f2 | cut -dx -f2`

# Next, for each address, symbolify. The line before the address is the
# corresponding call instruction, so we use grep –B 1 to retrieve it

$ for addr in $ADDRS;
      do otool -tV -arch i386 /mach_kernel | grep -B 1 $addr | head -1;
  done
0021b50b     calll     _Debugger                  ; panic() calls _Debugger()
002aab54     calll     0x0021b353                 ; calls _panic
002a09b3     calll     _kernel_trap               ; nearest symbol is lo_alltraps
..  ( return to IOKit Driver)
0053e80f     call     *0x000002e4(%eax)           ; __ZN9IOService5closeEPS_m
.. ( call to IOKit Driver)
0030aab4     call     *0x0083b690(%edx)           ; nearest symbol is _spec_ioctl
002fdf31     call     *(%eax,%edx,4)              ; inside VNOP_IOCTL
002f29a7     calll     _VNOP_IOCTL                ; unnamed function @002f2860
00470ecd     call     *0x08(%edx)                 ; nearest symbol is _fo_ioctl
0049cbfd     calll     0x00470e91                 ; nearest symbol is ioctl
004f6072     call     *0x04(%edi)                 ; Calling from syscall table
002a1448     calll     _unix_syscall64            ; In _lo64_unix_scall
```

What do we do about the IOKit Driver? The dump identified it as `com.apple.iokit.IOStorageFamily` kext. The binary resides in `/System/Library/Extensions` (`IOStorageFamily.Kext/Contents/MacOS/IOStorageFamily`). To make sure we have the right version, use `grep` on the `Info.plist` file, as shown in Output 9-3:

OUTPUT 9-3: Verifying the kernel extension version

```
$ cat /System/Library/Extensions/IOStorageFamily.Kext/Contents/Info.plist |
         grep -B 1 1.6.2
     <key>CFBundleShortVersionString</key>
     <string>1.6.2</string>
--
     <key>CFBundleVersion</key>
     <string>1.6.2</string>
```

This is, as expected, 1.6.2. We can then try `otool(1)` on it. But, because a kext is a relocatable file, the addresses displayed by `otool(1)` will be wrong — based at `0x00000000`. Turning to the panic log again, note the address range: `0xf97000` through `0xfaefff`. It then becomes trivial to find the symbols. For example, to find `0xfa60fa`, we would have to look for the difference between `0xfa60fa` to `0xf97000` — i.e., `0xf0fa`.

We can now reconstruct the chain of events (written in order), as shown in Output 9-4. Finding the kext addresses is left as an exercise for the reader, and is done in a similar manner to the one described here.

OUTPUT 9-4: Reconstructed chain of events.

```
002a1448     calll     _unix_syscall64      ; Entry from user mode: syscall64
004f6072     call     *0x04(%edi)           ; Dispatch to syscall table
0049cbfd     calll     0x00470e91           ; nearest symbol is ioctl
00470ecd     call     *0x08(%edx)           ; nearest symbol is _fo_ioctl
002f29a7     calll     _VNOP_IOCTL          ; (*fp->f_ops->fo_ioctl)
002fdf31     call     *(%eax,%edx,4)        ; inside VNOP_IOCTL
0030aab4     call     *0x0083b690(%edx)     ; nearest symbol is _spec_ioctl
0xfa60fa (0xc267b00 0xa75df80 0x0 0x3)      ; IOPartitionScheme::handleClose
0053e80f     call     *0x000002e4(%eax)     ; IOService::close (provider)
0xf9ea1c (0xc267b00 0xe0000100 0x0 0x0)     ; driver::close(this, e0001000 are kIO bits)
0xf9e909 (0xc267b00 0x0 0x0 0x0)            ; . . .
0xf9a983 (0xe 0x48 0xd4f0010 0x10)
         << Page fault occurs and control passes to lo_alltraps >>
002a09b3     calll     _kernel_trap        ; nearest symbol is lo_alltraps
002aab54     calll     0x0021b353          ; i.e call _panic
0021b50b     calll     _Debugger
```

Because this is a 32-bit kernel, the arguments are all on the stack. You could thus dive even deeper, as the panic log specifies the four positions on the stack frame next to the return address — i.e. what would be up to four arguments. On a 64-bit system, you won't be so lucky and neither would you be on iOS. Both Intel 64-bit and ARM use the registers for parameter passing, using the stack only for those rare cases of more than 4-6 arguments. Reconstructing function arguments on those architectures is next to impossible.

SUMMARY

This chapter described the two most important phases of the kernel lifecycle — birth and death. The kernel is "born" when it is instantiated by the boot loader (in x86 - EFI's `boot.efi`, and in iOS - `iBoot`), and loads all the various subsystems and kernel threads before the first process, launched, emerges in user mode. The chapter followed the kernel startup, up to the beginning of the first BSD task — `launchd`. User mode boot is discussed in Chapter 7.

A kernel panic, which is the premature death of the kernel, isn't all too frequent an occurrence, but when it does happen, it is a serious incident. The kernel dumps whatever information it can, and then halts the CPU to prevent any damage to the system. This chapter explained panics, and described the means to diagnose them.

The next chapters will take you deeper into the kernel, by delving into the architectural components of XNU.

REFERENCES

1. iOS Kernel Exploitation, BlackHat 2011: https://media.blackhat.com/bh-us-11/Esser/BH_US_11_Esser_Exploiting_The_iOS_Kernel_Slides.pdf
2. TN2118. Kernel Core Dumps — http://developer.apple.com/library/mac/#technotes/tn2004/tn2118.html
3. Apple Developer Kernel Programming Guide — https://developer.apple.com/library/mac/#documentation/Darwin/Conceptual/KernelProgramming/
4. VMWare Debugging. Hardware Debugging — http://ho.ax/posts/2012/02/vmware-hardware-debugging/
5. TN2063. Understanding and Debugging Kernel Panics — http://developer.apple.com/library/mac/technotes/tn2063/

10

The Medium Is the Message: Mach Primitives

At the heart of XNU lies the Mach microkernel, which Apple assimilated from NeXTSTEP. Mach is the very core of the kernel in both OS X and iOS, although it is somewhat modified from its original version, which is Carnegie Mellon University's open source microkernel.

Even though the Mach core is wrapped by the BSD layer and the main kernel interface is in the standardized POSIX system calls, the core works with its own particular set of APIs and primitives. It is these constructs that this chapter discusses.

Mach may be a microkernel by design, but is a pretty complex system. This chapter therefore focuses on its core building blocks, as follows:

- **Introducing: Mach:** Presents the Mach design philosophy and goals.
- **Message Passing Primitives:** Discusses messages and ports, the basic of Mach IPC.
- **Synchronization Primitives:** Details the various kernel objects — locks and semaphores, which are used to ensure safety in concurrency.
- **IPC in depth:** Discusses what happens behind the scenes when Mach messages are passed, and discusses the Mach Interface Generator (MIG) tool, which is used throughout the kernel.
- **Machine Primitives:** Details the Mach host, clock processor, processor, and processor_set abstractions. These abstractions provide an architecture-independent way to access system information and functions.

The next chapters will cover specific domains in Mach — scheduling and virtual memory management.

INTRODUCING: MACH

Much has been written about the process that led to Apple adopting Mach in Mac OS X, but the history is of less significance to this book, which focuses primarily on the technical aspects. Suffice it to say that Apple's flagship at the time, the ailing Mac OS 9, was heading for the reefs: As a less-than-efficient operating system, based on cooperative multitasking and highly proprietary, its performance was limited and not up to par with its peers. Apple realized that sooner or later it would have to re-engineer its entire kernel. With the acquisition of NeXT, the opportunity presented itself to take its already proven (although somewhat avant-garde) kernel design, and use it in Mac OS.

Mach is the collaboration of many people, but arguably none have contributed to it as much as one — Avadis Tevanian, Jr. His fingerprints (in the form of the file main comments) are still present in much of the code. Tevanian was part of Mach since its inception at CMU, and later evolved it — first at NeXT, then at Apple, where he worked until 2006.

The Mach Design Philosophy

Mach started its life as academic research into operating system infrastructure. Contrary to the monolithic philosophy, which implements a full-blown, complicated kernel, Mach boasts a highly minimalist concept: a thin, minimal core, supporting an object-oriented model wherein individual, well-defined components (in effect, subsystems) communicate with one another by means of messages. Unlike other operating systems, which present a complete model on top of which user mode processes may be implemented, Mach provides a bare-bones model, on top of which the operating system itself may be implemented. OS X's XNU is one specific implementation of UNIX (specifically, BSD 4.4) over Mach, although in theory any operating system may use the same architecture. Indeed, Windows borrows some design concepts from Mach as well, albeit with a vastly different implementation.

In Mach, everything is implemented as its own object. Processes (which Mach calls *tasks*), threads, and virtual memory are objects, each with its own properties. This, in itself, is not anything noteworthy. Other operating systems also use objects (effectively, C structures with function pointers) to implement their underlying primitives.

What makes Mach different is its choice of implementing object-to-object communication by means of *message passing*. Unlike other architectures, in which one object can access another as the need arises through a well-known interface, Mach objects cannot directly invoke or call on one another. Rather, they are required to pass messages. The source object sends a message, which is queued by the target object until it can be processed and handled. Similarly, the message processing may produce a reply, which is sent back by means of a separate message. Messages are delivered reliably (if a message is sent, it is guaranteed to be received) in a FIFO manner (received in the same order they are sent). The content of the message is entirely up to the sender and the receiver to negotiate.

As a minimalist architecture, Mach does not concern itself with higher-level concepts. Once the basic primitives of a process and a thread are defined, everything else may be handled by separate threads. Files and file systems, for example, are left for a higher level to implement. Likewise, device drivers are a higher-level concept that is left undefined at the Mach layer.

The Mach kernel thus becomes a low-level foundation, concerning itself with only the bare minimum required for driving the operating system. Everything else may be implemented by some higher layer of an operating system, which then draws on the Mach primitives and manipulate them in whatever way it sees fit.

It's important to emphasize that while Mach calls are visible from user mode, they implement a deep core, on top of which a larger kernel may be implemented. Mach is, essentially, a *kernel-within-a-kernel*. The "official" API of XNU is that of the BSD POSIX layer, and Apple keeps Mach to the absolute bare minimum. The average developer knows nothing of Mach, thanks to the far richer enveloping Cocoa APIs. Mach calls, however, remain a fundamental part of the architecture.

Although XNU is open source, Apple (probably intentionally) does not provide much documentation about Mach, whereas other components of XNU are well documented. To exacerbate the issue, the documentation that is provided — in XNU's `osfmk/man` directory — is a collection of antiquated, and sometimes inaccurate, man2html pages. Some documentation may be found in CMU's original documents[1,2], but it too, is quite venerable and sometimes irrelevant.

> *While XNU relies on Mach 3.0, there are some considerable differences between the Mach implementation of XNU and that of CMU Mach, or GNU's. Apple has removed support for several Mach APIs that were previously supported — for example,* `task_set_emulation()` *calls, which were used for system call emulation (and in XNU return* `KERN_NOT_SUPPORTED`*). Likewise, thread tracing is no longer supported, nor is Mach's Event Trace Analysis Package (ETAP), although these features were present in older incarnations of XNU.*
>
> *On the other hand, XNU has made some significant additions, including adding custom virtual memory handlers. Even different versions of XNU sometimes contain noticeable differences in Mach. The rest of this chapter explores those Mach features that are present in XNU.*

Mach Design Goals

The design document of Mach (which is still freely available from the Open Source Foundation[3]) lists several design goals, first and foremost of which is moving all functionality out of the kernel and into user mode, leaving the kernel with the bare minima, i.e:

- Management of "points of control" or execution units (threads).
- Allocation of resources to individual threads or groups (tasks).
- Virtual memory allocation and management.
- Allocation of low-level physical resources — namely, the CPU, memory, and any physical devices.

Remember, that Mach only provides for the low-level arbitration primitives. That is, Mach will provide a means to enforce a policy, but not the policy itself. Mach does not recognize any security features, priority, or preferences — all of which need be defined by the higher-level implementation.

A powerful advantage of the Mach design is, that — unlike other operating systems — it has taken into account aspects of multi-processing. Much of the kernel functionality is implemented by separate, distinct components, which pass well-defined messages between them, with no global scope. As such, there is no real requirement that all the components execute on the same processor, or even the same machine. Theoretically, Mach could be extended to an operating system for computer clusters just as easily.

MACH MESSAGES

The most fundamental concept in Mach is that of a *message*, which is exchanged between two endpoints, or *ports*. The message is the core building block of Mach's IPC, and is designed to be suitable for passing between any two ports — whether local to the same machine, or on some remote host. Issues such as parameter serialization, alignment, padding and byte-ordering are all taken into consideration and hidden by the implementation.

Simple Messages

A message, like a network packet, is defined as an opaque blob encapsulated by a fixed header. In Mach's case, this is defined in `<mach/message.h>` simply as:

```
typedef struct
{
        mach_msg_header_t       header;
        mach_msg_body_t         body;
} mach_msg_base_t;
```

The message header is mandatory, and defines the required meta data about the message, namely:

```
typedef struct
{
  mach_msg_bits_t       msgh_bits;          // header bits—optional flags
  mach_msg_size_t       msgh_size;          // Size, in bytes
  mach_port_t           msgh_remote_port;   // Dst (outgoing) or src (incoming)
  mach_port_t           msgh_local_port;    // Src (outgoing) or dst (incoming)
  mach_msg_size_t       msgh_reserved;      // …
  mach_msg_id_t         msgh_id;            // Unique ID
} mach_msg_header_t;
```

Simply put, a message is a blob of `size msgh_size`, sent from one port to another, with some optional flags.

A message may optionally have a trailer, specified as a `mach_msg_trailer_type_t` (really just an `unsigned int`):

```
typedef struct
{
       mach_msg_trailer_type_t     msgh_trailer_type;
       mach_msg_trailer_size_t     msgh_trailer_size;
} mach_msg_trailer_t;
```

Each type further defines a particular trailer format. These are left extensible for future implementation, although the following trailers, listed in Table 10-1, are already defined:

TABLE 10-1: Mach Trailers

TRAILER	USED FOR
`mach_msg_trailer_t`	Empty trailer
`mach_msg_security_trailer_t`	Sender security token
`mach_msg_seqno_trailer_t`	Sequential numbering
`mach_msg_audit_trailer_t` `mach_msg_context_trailer_t`	Auditing token (for BSM)
`mach_msg_mac_trailer_t`	Mandatory Access Control policy label

Replies and kernel-based messages use the trailer option, which may be specified with a reserved flag, as shown later in Table 10-3.

Complex messages

The Mach message structures described so far are fairly simply simple, as one could expect. Some messages, however, require additional fields and structure. These messages, aptly titled "complex," are indicated by the presence of the `MACH_MSGH_BITS_COMPLEX` bit in their header flags, and are structured differently: The header is followed by a descriptor count field, and serialized descriptors back to back (though possibly of different sizes). The currently defined descriptors are shown in Table 10-2:

TABLE 10-2: Complex message descriptors

TRAILER	USED FOR
`MACH_MSG_PORT_DESCRIPTOR`	Passing around a port right
`MACH_MSG_OOL_DESCRIPTOR`	Passing out-of-line data
`MACH_MSG_OOL_PORTS_DESCRIPTOR`	Passing out-of-line ports
`MACH_MSG_OOL_VOLATILE_DESCRIPTOR`	Passing out-of-line data which may be subject to change (volatile)

As you can see in Table 10-2, most descriptors involve "out-of-line" data. This is an important feature of Mach messages, which allows the addition of scattered pointers to various data, in a manner somewhat akin to adding an attachment to an e-mail. This is defined in `<mach/message.h>` for a 64-bit structure as follows (32-bits defined similarly):

```
typedef struct
{
  uint64_t                    address;          // pointer to data
  boolean_t                   deallocate: 8;    // deallocate after send?
  mach_msg_copy_options_t     copy: 8;          // copy instructions
  unsigned int                pad1: 8;          // reserved
  mach_msg_descriptor_type_t  type: 8;          // MACH_MSG_OOL_DESCRIPTOR
```

```
    mach_msg_size_t              size;          // size of the data at address
  } mach_msg_ool_descriptor64_t;
```

Simply put, the OOL descriptor specifies the address and size of the data to be attached, and instructions as to how to deal with it: whether it can be deallocated, and copy options (e.g. physical/virtual copy). OOL-data descriptors are commonly used to pass large chunks of data, alleviating the need for a costly copy operation.

Sending Messages

Mach messages are sent and received with the same API function, `mach_msg()`. The function has implementations in both user and kernel mode, and has the following prototype:

```
mach_msg_return_t   mach_msg
                    (mach_msg_header_t               msg,
                     mach_msg_option_t            option,
                     mach_msg_size_t           send_size,
                     mach_msg_size_t       receive_limit,
                     mach_port_t            receive_name,
                     mach_msg_timeout_t          timeout,
                     mach_port_t                  notify);
```

The function takes a message buffer, which is an in pointer for a send operation, and an out pointer for a receive operation. A sister function, `mach_msg_overwrite`, lets the caller specify two more arguments — a `mach_msg_header_t *` to a receive buffer and the `mach_msg_size_t` buffer size.

In both cases, the actual operation — send or receive — can be determined and tweaked using any bitwise combination of the options shown in Table 10-3.

TABLE 10-3: mach_msg() Send Options

OPTION FLAG	USED TO
`MACH_RCV_MSG`	Receive a message into the `msg` buffer.
`MACH_RCV_LARGE`	Leave large messages queued and fail with `MACH_RCV_TOO_LARGE` if the receive buffer is too small. In this case, only the message header (which specifies the message size) will be returned, so the caller can allocate more memory.
`MACH_RCV_TIMEOUT`	Pay attention to the timeout field for receive operation and fail with a `MACH_RCV_TIMED_OUT` after `timeout` milliseconds if no message received. The `timeout` value may also be 0.
`MACH_RCV_NOTIFY`	Receive notification.
`MACH_RCV_INTERRUPT`	Allow operation to be interrupted (and return `MACH_RCV_INTERUPTED`), rather than retrying operation.
`MACH_RCV_OVERWRITE`	In `mach_msg_overwrite`, specifies the extra parameter — the receive buffer — is in/out.

MACH_SEND_MSG	Send the message in the msg buffer.
MACH_SEND_INTERRUPT	Allow send operation to be interrupted (and return MACH_SEND_INTERUPTED), rather than retrying operation.
MACH_SEND_TIMEOUT	Pay attention to the timeout field for send operation — and fail after timeout milliseconds with a MACH_SEND_TIMED_OUT.
MACH_SEND_NOTIFY	Notify message delivery to notify port.
MACH_SEND_ALWAYS	Used internally.
MACH_SEND_TRAILER	Specifies one of the known Mach trailers lies at offset size of the message (i.e. immediately after the message buffer).
MACH_SEND_CANCEL	(Removed in Lion) Cancel a message.

Originally, Mach messages were designed for a true micro-kernel architecture. That is, the mach_msg() function had to copy the memory backing the message between the sender and receiver. While this is true to the microkernel paradigm, the performance impediment of frequent memory copy operations proved unbearable. XNU, therefore, "cheats" by being monolithic: All kernel components share the same address space, so message passing can simply pass the pointer to the message, thereby saving a costly memory copy operation.

To actually send or receive messages, the mach_msg() function invokes a Mach trap. This is, essentially, the Mach equivalent of a system call, which was discussed in Chapter 8, which deals with kernel architectures. Calling mach_msg_trap() from user mode will use the trap mechanism to switch to kernel mode, wherein the kernel implementation of mach_msg() will do the work.

Ports

Messages are passed between end points, or *ports*. These are really nothing more than 32-bit integer identifiers, although they are not used as such, but as opaque objects. Messages are sent from some port to some other port. Each port may receive messages from any number of senders but has only one designated receiver, and sending a message to a port queues the message until it can be handled by the receiver.

All Mach primitive objects are accessed through corresponding ports. That is, by seeking a handle on an object, one really requests a handle to its port. Access to a port is by means of *port rights*, defined in <mach/port.h>, as shown in Table 10-4:

TABLE 10-4: Mach Port Rights

MACH_PORT_RIGHT_	MEANING
SEND	Send (enqueue) messages to this port. Multiple senders are allowed.
RECEIVE	Read (dequeue) messages from this port. Effectively, this is ownership of the port.

continues

TABLE 10-4 *(continued)*

MACH_PORT_RIGHT_	MEANING
SEND_ONCE	Send only one message. The right immediately revoked afterwards, into DEAD_NAME.
PORT_SET	Receive rights to multiple ports simultaneously.
DEAD_NAME	Port right after SEND_ONCE is exhausted.

The key rights are, as one can imagine, SEND and RECEIVE. SEND_ONCE is the same as SEND, but allows for only one message (that is, it is revoked by the system after its first use). The holder of the MACH_PORT_RIGHT_RECEIVE right is, in effect, the owner of the port, and the only entity allowed to dequeue messages from the port.

The functions in <mach/mach_port.h> can be used to manipulate task ports, even from outside the task. In particular, the mach_port_names routine can be used to dump the port namespace of a given task. Listing 10-1 reproduces the functionality of GDB's info mach-ports command.

LISTING 10-1: A simple Mach port dumper

```
kern_return_t lsPorts(task_t TargetTask)
{
    kern_return_t              kr;
    mach_port_name_array_t     portNames           = NULL;
    mach_msg_type_number_t     portNamesCount;
    mach_port_type_array_t     portRightTypes      = NULL;
    mach_msg_type_number_t     portRightTypesCount;
    mach_port_right_t          portRight;
    unsigned int               p;

    // Get all of task's ports
    kr = mach_port_names(TargetTask,
                         &portNames,
                         &portNamesCount,
                         &portRightTypes,
                         &portRightTypesCount);
    if (kr != KERN_SUCCESS)
    { fprintf (stderr,"Error getting mach_port_names.. %d\n", kr);return (kr); }
    // Ports will be dumped in hex, like GDB, which is somewhat limited. This can be
    // extended to recognize the well known global ports (left as an exercise for the
    // reader)
    for (p = 0; p < portNamesCount; p++) {
         printf( "0x%x 0x%x\n",  portNames[p], portRightTypes[p]);
      } // end for
} // end lsPorts

int main(int argc, char * argv[])
{
    task_t               targetTask;
    kern_return_t        kr;
```

```
    int pid = atoi (argv[1]);
    // task_for_pid() is required to obtain a task port from a given
    // BSD PID. This is discussed in the next chapter
    kr  = task_for_pid(mach_task_self(),pid, &targetTask);
    lsPorts (targetTask);
    // Not strictly necessary, but be nice
    kr = mach_port_deallocate(mach_task_self(), targetTask);
}
```

A more complete example can be found in Apple Developer's sample code for `MachPortDump`[4].

Passing Ports Between Tasks

Ports and rights may be passed from one entity to another. Indeed, it is not uncommon to see complex Mach messages containing ports delivered from one task to another. This is a very powerful feature in IPC design, somewhat akin to mainstream UNIX's domain sockets, which allow the passing of file descriptors between processes.

Lion enables the conversion of UNIX file descriptors into Mach ports, and vice versa. These objects, appropriately called *fileports*, are primarily used by the notification system.

Port Registration and the Bootstrap Server

Mach allows ports to be registered globally — that is, on a system-wide level, with a port naming server. In XNU, this "bootstrap server" is none other than `launchd(8)` — PID 1 — which, at the Mach task level, registers the bootstrap service port. (recall the discussion in Chapter 7, which explained this in detail under `launchd's` role of `mach_init`). Because every other process (and therefore Mach task) on the system is a descendant of `launchd`, it inherits this port upon birth. The APIs in Chapter 7 can then be used to locate service ports.

The Mach Interface Generator (MIG)

Mach's model of message passing is one implementation of Remote Procedure Call (RPC). In a perfect world, the programmer need not bother with the implementation of message passing, since these are performed at a lower-level, and are largely independent of the message contents. The underlying support code can therefore be automatically generated: The programmer need only write the interface specification, using a higher level Interface Definition Language (IDL), from which a specialized pre-processor tool can generate the code required to construct the actual messages, and send them (In higher level languages this is sometimes referred to as serialization, or marshaling). To enable RPC to be architecture-independent and agnostic to byte-ordering, a network data representation is often adopted.

Classic UN*X has SUN-RPC, which is still widely used (as an integral part of NFS). In it, a portmapper (running on TCP or UDP port 111) is responsible for maintaining registered programs. The programs themselves make use of the *rpcgen* compiler to generate code from the IDL. Data is converted into an external data representation (XDR), which is in network byte ordering. Mach does not use a dedicated port mapper (though `launchd(8)` handles some of the logic), but has a component very similar to *rpcgen*, called the Mach Interface Generator, commonly referred to as MIG.[5]

If you look at the `/usr/include/mach` directory, you will see (alongside the miscellaneous header files), `.defs` files. These files contain the IDL definition files for the various Mach "subsystems," as shown in Table 10-5:

TABLE 10-5: Mach subsystem interface definition files in <mach/*>

BASE	SUBSYSTEM	USE
123	`audit_triggers`	Audit logging facility. Contains a single routine — `audit_triggers`
1000	`Clock`	Clock and alarm routines
1200	`clock_priv`	Kernel clock privileged interface definitions
3125107	`clock_reply`	Contains reply to `clock_alarm` request
2401 2405	`exc` `mach_exc`	Mach exception handling
950	`host_notify_reply`	Contains a single routine, `host_calendar_changed`
400	`host_priv`	Host privileged operations, such as reboot, kernel modules, and physical memory
600	`host_security`	Contains definitions for task tokens
5000	`ledger`	Contains definitions for the resource book-keeping subsystem. This was part of the Mach specification, but was inactive in XNU up until iOS 5.0 and Mountain Lion
617000	`lock_set`	Lock set subsystem (detailed in the previous section)
200	`mach_host`	Mach host abstraction routines (detailed in this chapter)
3200	`mach_port`	Mach port handling functions
–	`mach_types`	Data type definitions for kernel objects
4800	`mach_vm`	Miscellaneous virtual memory handling functions. Supersedes vm (detailed in Chapter 12)
64	`notify`	Port notification routines
3000	`processor`	Processor control (detailed in this chapter)
4000	`processor_set`	Processor set control (detailed in this chapter)
5200	`security`	Security and Mandatory Access Control interfaces
–	`std_types`	Data type definitions
3400	`task`	Task operations (detailed in Chapter 11)

27000	`task_access`	OS X/iOS enhancement to support access checks on task handles and code signature checks (detailed in Chapter 10)
3600	`thread_act`	Thread operations (detailed in Chapter 11)
3800	`vm_map`	Miscellaneous virtual memory handling functions. Superseded by `mach_vm` (detailed in Chapter 12)

The *subsystem*s are collections of *operations* that are grouped together. The operations will be serialized in Mach messages. User programs can declare and use additional subsystems, as `launchd(8)` does (e.g. `protocol_vproc`, subsystem #400, by means of which `launchctl(1)` can communicate with it). There is also no need for global uniqueness (the abovementioned `protocol_vproc` overlaps with `host_priv`), so long as the destination of the message knows which subsystem is relevant.

An *operation* can be one of several types. The MIG specification lists the following types shown in Table 10-6.

TABLE 10-6: MIG Operation types

OPERATION TYPE	PURPOSE
Routine Simpleroutine	Sends a message to the server. A routine blocks until a reply is received, and returns a `kern_return_t`. A simpleroutine does not block to receive a reply, but returns immediately with the return code from `msg_send()` .
Procedure Simpleprocedure	As routines, but do not return a `kern_return_t` .
Function	Returns a value from the server function.

In practice, XNU only uses routines and simpleroutines. The various operations are numbered sequentially, starting with the subsystem's base number. The keyword "skip" may be used to reserve numbers for deprecated or obsolete operations.

The `mig(1)` command line tool acts as the pre-processor for the `.defs` files, and creates the `.h` and `.c` files for the client and the server (the latter are actually created by `migcom(1)`, a utility used internally). This command is not normally part of OS X or XCode, but is part of the `bootstrap_cmds` package which can be readily downloaded from `http://opensource.apple.com`.

For each operation, `mig(1)` generates a substantial portion of code, for both the client and the server, along with a C-style header file. The operation is converted into a C function which encapsulates the message passing code (i.e. the call to `mach_msg()` with `MACH_SEND_MSG` and `MACH_RCV_MSG` flags). The generated code handles all the message house-keeping, such as validation of types, lengths, and return values. A significant chunk of the code also handles Network Data Representation (NDR, akin to SUNRPC's XDR, eXternal Data Representation), which is largely empty conversion macros, as XNU does not support network-borne Mach messaging.

The following experiment illustrates how the Mach Interface Generator is used to automatically generate code.

Experiment: Using mig(1) to Generate Files Automatically

The `mig(1)` utility operates on `.defs` files in a similar manner. To show this, pick an arbitrary file in `/usr/include/mach` — in this example, `mach_host.defs`. Looking at the file, you should be able to see the definitions of routines, as shown in Listing 10-2:

LISTING 10-2: mach_host.defs and the host MIG subsystem

```
...
subsystem
#if     KERNEL_SERVER
          KernelServer
#endif  /* KERNEL_SERVER */
                     mach_host 200;          Message Base

/*
 *      Basic types
 */

#include <mach/std_types.defs>
#include <mach/mach_types.defs>
#include <mach/clock_types.defs>
#include <mach_debug/mach_debug_types.defs>

...

routine host_info(
                host            : host_t;     Message #200 (Base + 0)
                flavor          : host_flavor_t;
        out     host_info_out   : host_info_t, CountInOut);
...

routine host_kernel_version(                  Message #201 (Base + 1)
                host            : host_t;
                out     kernel_version  : kernel_version_t);
...

skip; /* was enable_bluebox */     // was message 211
skip; /* was disable_bluebox */    // was message 212
```

Copy the file into an empty directory, and run the `mig(1)` utility on the file. You should see the following files as in Output 10-1:

OUTPUT 10-1: Output of running mig(1) on mach_host.defs

```
morpheus@Ergo (/tmp/scratch)$ ls -l
total 792
-r--r--r--  1 morpheus  wheel    6975 Mar 26 11:34 mach_host.defs
-rw-r--r--  1 morpheus  wheel   20334 Mar 26 11:34 mach_host.h
-rw-r--r--  1 morpheus  wheel  164125 Mar 26 11:34 mach_hostServer.c
-rw-r--r--  1 morpheus  wheel  207442 Mar 26 11:34 mach_hostUser.c
```

The resulting `mach_host.h` file is the `#include` file readily usable by C programs, and should be nearly or entirely identical to the `<mach/mach_host.h>`. Looking at the client file, you will notice the considerable amount of automatically generated code. Looking specifically at the `host_info` message, you should see something like listing 10-3, which has been further annotated for readability:

LISTING 10-3: The mach_hostUser.c file generated by mig(1) from mach_host.defs

```
...
/* Routine host_info */

// prototype generated directly from defs
mig_external kern_return_t host_info
(
        host_t host,
        host_flavor_t flavor,
        host_info_t host_info_out,
        mach_msg_type_number_t *host_info_outCnt
)
{

// MIG defines the request and reply structures next.

#ifdef  __MigPackStructs
#pragma pack(4)
#endif
        typedef struct {
                mach_msg_header_t Head;
                NDR_record_t NDR;                    // Network data representation
                                                     // information
                host_flavor_t flavor;
                mach_msg_type_number_t host_info_outCnt;
        } Request;
#ifdef  __MigPackStructs
#pragma pack()
#endif

#ifdef  __MigPackStructs
#pragma pack(4)
#endif
        typedef struct {
                mach_msg_header_t Head;
                NDR_record_t NDR;                    // Network data representation
                                                     // information
                kern_return_t RetCode;
                mach_msg_type_number_t host_info_outCnt;
                integer_t host_info_out[15];
                mach_msg_trailer_t trailer;
        } Reply;
#ifdef  __MigPackStructs
#pragma pack()
#endif

    union {
```

```
routine host_info(
     host            : host_t;
     flavor          : host_flavor_t;
 out host_info_ out  : host_info_t,
                       CountInOut);
```

continues

LISTING 10-3 *(continued)*

```
                Request In;
                Reply Out;
        } Mess;

        Request *InP = &Mess.In;
        Reply *Out0P = &Mess.Out;

        mach_msg_return_t msg_result;

#ifdef  __MIG_check__Reply__host_info_t__defined
        kern_return_t check_result;
#endif  /* __MIG_check__Reply__host_info_t__defined */

        DeclareSendRpc(200, "host_info")

        InP->NDR = NDR_record;

        InP->flavor = flavor;

        // somewhat crude sanity check on argument length. "15" is the hard-coded limit
        if (*host_info_outCnt < 15)
                InP->host_info_outCnt = *host_info_outCnt;
        else
                InP->host_info_outCnt = 15;

        // Prepare message header
        InP->Head.msgh_bits =
                MACH_MSGH_BITS(19, MACH_MSG_TYPE_MAKE_SEND_ONCE);
        /* msgh_size passed as argument */
        InP->Head.msgh_request_port = host;
        InP->Head.msgh_reply_port = mig_get_reply_port();
        InP->Head.msgh_id = 200;

   __BeforeSendRpc(200, "host_info")                 Message #200 (Base + 0)

      // this is the heart of host_info and, indeed, most MIG generated code: A call to
      // mach_msg.

        msg_result = mach_msg(&InP->Head, MACH_SEND_MSG|MACH_RCV_MSG|
        MACH_MSG_OPTION_NONE, (mach_msg_size_t)sizeof(Request),
       (mach_msg_size_t)sizeof(Reply), InP->Head.msgh_reply_port, MACH_MSG_TIMEOUT_NONE,
        MACH_PORT_NULL);
        __AfterSendRpc(200, "host_info")

      // If the message sending fails, we have nothing more to seek here. Abort.
        if (msg_result != MACH_MSG_SUCCESS) {
                __MachMsgErrorWithoutTimeout(msg_result);
                { return msg_result; }
        }

      // MIG can optionally define reply checking logic. It is easier for it to generate
      // the code anyway, #ifdef'd, so as to generate uniform code in all cases.

#if     defined(__MIG_check__Reply__host_info_t__defined)
```

```
        check_result = __MIG_check__Reply__host_info_t((__Reply__host_info_t *)Out0P);
        if (check_result != MACH_MSG_SUCCESS)
                { return check_result; }
#endif  /* defined(__MIG_check__Reply__host_info_t__defined) */

        // If output is within specified buffer bounds, copy what we can to caller, and
        // fail
        if (Out0P->host_info_outCnt > *host_info_outCnt) {
                (void)memcpy((char *) host_info_out, (const char *)
                Out0P->host_info_out,4 *  *host_info_outCnt);
                *host_info_outCnt = Out0P->host_info_outCnt;
                { return MIG_ARRAY_TOO_LARGE; }
        }
        // Otherwise, it is safe to copy all the output to the caller.
        (void)memcpy((char *) host_info_out, (const char *) Out0P->host_info_out, 4 *
         Out0P->host_info_outCnt);

        // Set buffer count
        *host_info_outCnt = Out0P->host_info_outCnt;

        // And.. we're done!
        return KERN_SUCCESS;
}
```

Replies, by convention, are numbered at 100 over their respective requests. This means that the reply to `host_info` (#200), for example, will be 300, as you can indeed verify by looking at the code generated for `__MIG_check__Reply__host_info_t`, in the same file.

IPC, IN DEPTH

So far, we have covered the basic primitives required for IPC: the messages, the ports they are sent from and received on, and the semaphores and locks required to enable safe concurrency. But we have given little attention to the underlying implementation of these primitives, in particular the port objects themselves. This section goes into more detail.

Every Mach task (the high-level abstraction somewhat corresponding to a process, as you will see in the next chapter) contains a pointer to its own IPC namespace, which holds its own ports. Additionally, a task can obtain the system-wide ports, such as the host port, the privileged ports, and others.

The port object exported to user space (the `mach_port_t` previously shown) is really a handle to the "real" port object, which is an `ipc_port_t`. This is defined in `osfmk/ipc/ipc_port.h` as shown in Listing 10-4.

LISTING 10-4: The structure behind a Mach port

```
struct ipc_port {

        /*
         * Initial sub-structure in common with ipc_pset
         * First element is an ipc_object second is a
```

continues

LISTING 10-4 *(continued)*

```
         * message queue
         */
        struct ipc_object ip_object;
```

```
   struct ipc_object
      {
        ipc_object_bits_t io_bits;
        ipc_object_refs_t io_references;
        decl_lck_mtx_data(,     io_lock_data)
       };
```

```
        struct ipc_mqueue ip_messages;
```

```
 typedef struct ipc_mqueue {
   union {
     struct {
        struct  wait_queue      wait_queue;
        struct ipc_kmsg_queue   messages;
        mach_port_msgcount_t    msgcount;
        mach_port_msgcount_t    qlimit;
        mach_port_seqno_t       seqno;
        mach_port_name_t        receiver_name;
        boolean_t               fullwaiters;
            } port;
     struct {
            struct wait_queue_set   set_queue;
            mach_port_name_t        local_name;
            } pset;
         } data;
 } *ipc_mqueue_t;
```

```
        union {
                struct ipc_space *receiver;     // pointer to receiver's IPC space
                struct ipc_port  *destination;  // or pointer to global port
                ipc_port_timestamp_t timestamp;
        } data;

        ipc_kobject_t ip_kobject;       // Type of object behind this port (IKOT_*
                                        // constant from osfmk/kern/ipc_kobject.h)

        mach_port_mscount_t ip_mscount;
        mach_port_rights_t ip_srights;
        mach_port_rights_t ip_sorights;

        struct ipc_port *ip_nsrequest;
        struct ipc_port *ip_pdrequest;
        struct ipc_port_request *ip_requests;
        boolean_t ip_sprequests;

        unsigned int ip_pset_count;
        struct ipc_kmsg *ip_premsg;
        mach_vm_address_t ip_context;

...

#if CONFIG_MACF_MACH
        struct label    ip_label;     // used to enforce BSD's Mandatory Access Control
                                      // Framework
#endif
};
```

To gain a better understanding, it helps to look at the implementations of the two most important IPC functions: `mach_msg_send()` and `mach_msg_receive()`.

Behind the Scenes of Message Passing

Mach messages in user mode use the `mach_msg()` function, described earlier, which calls its corresponding kernel function `mach_msg_trap()` through the kernel's Mach trap mechanism (discussed in Chapter 8). The `mach_msg_trap()` falls through to `mach_msg_overwrite_trap()`, which determines a send or receive operation by testing `MACH_SEND_MSG` or `MACH_RCV_MSG` flag, respectively.

Sending Messages

Mach message-sending logic is implemented in two places in the kernel: `mach_msg_overwrite_trap()`, and `mach_msg_send()`. The latter is used only for kernel-mode message passing, and is not visible from user mode.

In both cases, the logic is similar, and proceeds according to the following:

- Obtain current IPC space by a call to `current_space()`.
- Obtain current VM space (`vm_map`) by a call to `current_map()`.
- Sanity check on size of message.
- Compute `msg` size to allocate: This is taken from the `send_size` argument, plus a hard coded `MAX_TRAILER_SIZE`.
- Allocate the message using `ipc_kmsg_alloc`.
- Copy the message (`send_size` bytes of it), and set `msgh_size` in header.
- Copy the port rights associated with the message, and any out-of-line memory into the current `vm_map` by calling `ipc_kmsg_copyin`. This function calls `ipc_kmsg_copyin_header` and `ipc_kmsg_copyin_body`, respectively.
- Call `ipc_kmsg_send()` to actually send the message:
    - First, a reference to `msgh_remote_port` is obtained, and locked.
    - If the port is a kernel port (i.e. the port `ip_receiver` is the kernel IPC space), the message is processed using `ipc_kobject_server()` (`from osfmk/kern/ipc_kobject.c`). This will find the corresponding function in the kernel to execute on the message (or call `ipc_kobject_notify()` to do so) and should also generate a reply to the message.
    - In any case — that is, if the port is not in kernel space, or due to a reply returned from `ipc_kobject_server()` — the function falls through to deliver the message (or the reply to it) by calling `ipc_mqueue_send()`, which copies the message directly to the port's `ip_messages` queue and wakes up any waiting thread.

Receiving Messages

Similar to the message sending case, the Mach message-sending logic is implemented in two places in the kernel. As before, the `mach_msg_overwrite_trap()` is used to serve requesters from user mode, whereas `mach_msg_receive()` is reserved for kernel-mode ones.

- Obtain current IPC space by a call to `current_space()`.
- Obtain current VM space (`vm_map`) by a call to `current_map()`.

- No sanity check is performed on the size of the message. This is unnecessary, as messages have been validated during sending.
- The IPC queue is obtained by a call to `ipc_mqueue_copyin()`
- A reference is held on the current thread. Using a reference on the current thread makes it suitable for Mach's continuation model, which alleviates the need to maintain the full thread stack. This model is described in more detail in the Mach scheduling chapter.
- The `ipc_mqueue_receive()` is called to dequeue the message.
- Finally, `mach_msg_receive_results()` is called. This function could also be called from a continuation.

SYNCHRONIZATION PRIMITIVES

Message-passing is just one component of the Mach IPC architecture. The second is *synchronization*, which enables two or more concurrent operations to determine access to shared resources.

Synchronization relies on the ability to exclude access to a resource while another is using it. The most basic primitive, therefore, is a *mutual exclusion* object, or *mutex*. Mutexes are nothing more than ordinary variables in kernel memory, usually integers up of machine size, with one special requirement — the hardware must enforce atomic operations on them: "Atomic," in the sense that an operation on a mutex cannot be disrupted — not even by a hardware interrupt. In SMP systems, a second requirement of physical mutual exclusion is required, which is usually implemented by some type of memory fence or barrier.

The following section describes Mach's synchronization primitives. There are quite a few of those, and each is aimed at a particular purpose. As a quick guide, consult Table 10-7:

TABLE 10-7: Mach Synchronization Primitives

OBJECT	IMPLEMENTED IN	OWNER	VISIBILITY	WAIT
Mutex (`lck_mtx_t`)	`i386/i386_locks.c`	One	Kernel	Idle*
Semaphore (`semaphore_t`)	`kern/sync_sema.c`	Many	User	Idle
Spinlock (`hw_lock_t, ..`)	`i386/i386_lock.s`	One	Kernel	Busy
Lock sets (`lock_set_t`)	`kern/sync_lock.c`	One	User	Idle (as mutex)

Like most of the primitives discussed in this chapter, Mach provides lock by putting together two layers:

- **The hardware specific layer:** Relies on processor idiosyncrasies and specific assembly instructions to provide the atomicity and exclusion
- **The hardware agnostic layer:** Wraps the specifics with a uniform API. The API makes the layers on top of Mach (or the user API) totally oblivious to the implementation specifics. This is usually achieved with a simple set of macros.

Lock Group Objects

Most Mach synchronization objects do not exist by their own right. Rather, they belong to a `lck_grp_t` object. The lock groups are defined in `osfmk/kern/locks.h` as shown in Listing 10-5:

LISTING 10-5: The lck_grp_t, from osfmk/kern/locks.h

```
typedef struct _lck_grp_ {
        queue_chain_t           lck_grp_link;
        uint32_t                lck_grp_refcnt;
        uint32_t                lck_grp_spincnt;
        uint32_t                lck_grp_mtxcnt;
        uint32_t                lck_grp_rwcnt;
        uint32_t                lck_grp_attr;
        char                    lck_grp_name[LCK_GRP_MAX_NAME];
        lck_grp_stat_t          lck_grp_stat;
} lck_grp_t;
```

Simply put, the `lck_grp_t` is simply a member in a linked list, with a given name, and up to three lock types: spinlocks, mutexes, and read/write locks. A lock group also has statistics (the `lck_grp_stat_t`), which can be used for debugging synchronization related issues. The attributes are largely unused, though `LCK_ATTR_DEBUG` can be set. Table 10-8 lists the APIs for creating and destroying lock groups:

TABLE 10-8: Mach lock group API functions

MACH MUTEX API	USED TO
`lck_grp_t` **`*lck_grp_alloc_init`** `(const char* grp_name, lck_grp_attr_t *attr);`	Create a new lock group. The group is identified by *grp_name*, and possesses the attributes specified in *attr*. In most cases, the attributes are default, as set by `lck_grp_attr_alloc_init();`
`void` **`lck_grp_free`** `(lck_grp_t *grp);`	Deallocate lock group *grp*.

Virtually every subsystem of Mach, as well as most of BSD, creates and utilizes a lock group for itself during initialization.

Mutex Object

The most commonly used lock object is the mutex. Mutexes are defined as `lck_mtx_t` objects. The mutex objects are largely architecture agnostic. A mutex must belong to a lock group and are defined in `osfmk/kern/locks.h` with the operations in Table 10-9:

TABLE 10-9: Mach mutex API functions

MACH MUTEX API	USED TO
`lck_mtx_t *lck_mtx_alloc_init( lck_grp_t *grp, lck_attr_t *attr);`	Allocate a new mutex object, belonging to group `grp`, with the attributes specified by `attr`.
`lck_mtx_init( lck_mtx_t *lck, lck_grp_t *grp, lck_attr_t *attr);`	As `lck_mtx_alloc_init`, but initializes an already allocated mutex `lck`.
`lck_mtx_lock(lck_mtx_t *lck)` `lck_mtx_try_lock(lck_mtx_t *l)`	Lock the mutex *lck*. This will block indefinitely. The try variant doesn't block, but may fail.
`lck_mtx_unlock(lck_mtx_t*lck);`	Unlock the mutex *lck*.
`lck_mtx_destroy(lck_mtx_t *lck, lck_grp_t *grp);`	Mark *lck* as destroyed and no longer usable. The mutex is still allocated, however (and may be reinitialized)
`lck_mtx_free(lck_mtx_t *lck, lck_grp_t *grp);`	Mark *lck* as destroyed, and deallocate it.
`wait_result_t lck_mtx_sleep (lck_mtx_t *lck, lck_sleep_action_t action, event_t event, wait_interrupt_t inter);`	Make current thread sleep until *lck* becomes available.
`wait_result_t lck_mtx_sleep_deadline (lck_mtx_t *lck, lck_sleep_action_t action, event_t event, wait_interrupt_t inter, uint64_t deadline);`	Make current thread sleep until *lck* becomes available, or until deadline has been met.

The implementation of the mutex operation is architecture-dependent, and in the open source XNU is split between `osfmk/kern/locks.c` and `osfmk/i386/locks_i386.c`, with optimized assembly primitives in `osfmk/i386/i386_lock.s`. There are additionally `lck_mtx_lock__[try]_spin_*` functions, which on Intel architectures can convert mutexes to spinlocks (discussed later).

Read-Write Lock Object

Mutexes have a major drawback, which is that only one thread can hold them at a given time. In many scenarios, multiple threads may require read-only access to a resource. In those cases, using a mutex would prevent concurrent access, even though the threads would not interfere with one another.

Enter: The read-write lock. This is a "smarter" mutex, which distinguishes between read and write access. Multiple readers ("consumers") can hold the lock at any given time, but only one writer ("producer") can hold the lock. When a writer holds the lock, all other threads are blocked. The API for read-write locks is largely identical to that of mutexes, save for the locking functions, which accept a second argument specifying the lock type.

TABLE 10-10: Mach rwlock API functions

MACH RWLOCK API	USED TO
`lck_rw_t` **`*lck_rw_alloc_init`** `(lck_grp_t *`*`grp`*`, lck_attr_t *`*`attr`*`);`	Allocate a new rwlock object, belonging to group `grp`, with the attributes specified by `attr`.
`lck_rw_init``(lck_rw_t *`*`lck`*`, lck_grp_t *grp, lck_attr_t *`*`attr`*`);`	As `lck_rw_alloc_init`, but initializes an already allocated `rw lck`.
`lck_rw_lock``(lck_rw_t *lck, lck_rw_type_t `*`read_or_write`*`);`	Lock the mutex `lck` for `read_or_write` access. Readers: This call will block only if a writer holds the lock. Writers: This call will block until all other threads give up the lock. This call is a wrapper of `lck_rw_lock_shared` and `lck_rw_lock_exclusive`.
`lck_rw_unlock``(lck_mtx_t *`*`lck`*`, lck_rw_type_t `*`read_or_write`*`);`	Unlock the mutex `lck`. This call is a wrapper of `lck_rw_unlock_shared` and `lck_rw_unlock_exclusive`.
`lck_rw_destroy``(lck_mtx_t *`*`lck`*`, lck_grp_t*`*`grp`*`);`	Mark `lck` as destroyed and no longer usable. The mutex is still allocated, however (and may be reinitialized).
`lck_mtx_free``(lck_mtx_t *lck, lck_grp_t *grp);`	Mark `lck` as destroyed, and deallocate it.
`wait_result_t` **`lck_rw_sleep`** `(lck_mtx_t *lck, lck_sleep_action_t `*`action`*`, event_t `*`event`*`, wait_interrupt_t `*`inter`*`);`	Make current thread sleep until *`lck`* becomes available. The *`action`* can specify `LCK_SLEEP_SHARED` or `LCK_SLEEP_EXCLUSIVE`.

Spinlock Object

Both mutexes and semaphores are idle-wait objects. This means that if the lock object is held by some other owner, the thread requesting access is added to a wait queue, and is blocked. Blocking a thread involves giving up its time slice and yielding the processor to whichever thread the scheduler decrees should be next. When the lock is made available, the scheduler will be notified and — at its discretion — dequeue the thread and reschedule it. This, however, could severely impact performance, since often times the object is only held for a few cycles, whereas the cost of two or more context switches is orders of magnitude greater. In these cases, it may be advisable to not yield the processor, and — instead — continue to try to access the lock object repeatedly, in what is called a busy-wait. If, indeed, the current owner of the lock object relinquishes it anyway in a matter of a few cycles, it saves at least two context switches.

This "if," however, is a really big "if." A spinning thread does so in what may end up being an endless loop: The current owner may not give up the spinlock so quickly, and could in fact hold it indefinitely while waiting for some other resource. This leads to the much-dreaded busy deadlock scenario, in which the entire system may grind to a halt.

The basic spinlock type is the hardware-specific `hw_lock_t`. On top of it are implemented the other lock types: the `lck_spin_t` (a thin wrapper), the `simple_lock_t`, and the `usimple_lock_t`. The locks may have different implementations, though in practice the simple lock is usually just `#define`d over the usimple one.

The APIs for all three spinlock types resemble those of the other objects. A detailed example of locking at the hardware level (the `hw_lock_t`), contrasting ARM and Intel as well as UP and SMP, can be found in the appendix in this book.

Semaphore Object

Mach offers semaphores, which are generalizations of mutex objects. A semaphore is a mutex object whose value can be other than 0 or 1 — up to some positive number, which is the count of concurrent semaphore holders. To put it another way, a mutex can be considered as a special case of a binary semaphore. Semaphores, however, are visible in user mode, whereas mutexes aren't.

> *Mach semaphores are not the same as POSIX semaphores. The API presented here is different, and not POSIX compliant. The underlying implementation of POSIX semaphores, however, is over Mach semaphores (e.g. POSIX's* `sem_open()` *calls on Mach's* `semaphore_create()`)

The API for semaphores, listed in Table 10-11 is straightforward to use:

TABLE 10-11: Mach Semaphore API functions

MACH SEMAPHORE API	USED TO
`semaphore_create(task_t t, semaphore_t *sem, int policy, int value);`	Create a new semaphore in `sem` for task `t`, with initial count `value`. The `policy` indicates how blocking threads will be awakened, as per the same values of lock policies.
`semaphore_destroy (task_t t, semaphore_t semaphore);`	Destroy a `semaphore` port semaphore in `t`.
`semaphore_signal (semaphore_t semaphore);`	Increment count of a semaphore. If the count becomes greater than or equal to zero, a blocking thread is awakened, according to the policy.
`semaphore_signal_all (semaphore_t semaphore);`	Set count of semaphore to zero, thereby waking all threads.
`semaphore_wait (semaphore_t semaphore);`	Decrement count on semaphore, and block until count becomes non-negative again.

The semaphore itself is not a lockable object. It is a small struct, containing the reference to the owner and its port. Additionally, it contains a `wait_queue_t`, which is a linked list of threads waiting on it. It is that `wait_queue_t` which gets locked, by means of a hardware lock. This is shown in Listing 10-6:

LISTING 10-6: THE SEMAPHORE OBJECT, FROM osfmk/kern/sync_sema.h

```
typedef struct semaphore {
        queue_chain_t     task_link;  /* chain of semaphores owned by a task */
        struct wait_queue wait_queue; /* queue of blocked threads & lock     */
        task_t            owner;      /* task that owns semaphore            */
        ipc_port_t        port;       /* semaphore port                      */
        uint32_t          ref_count;  /* reference count                     */
        int               count;      /* current count value                 */
        boolean_t         active;     /* active status                       */
} Semaphore;

#define semaphore_lock(semaphore)   wait_queue_lock(&(semaphore)->wait_queue)
#define semaphore_unlock(semaphore) wait_queue_unlock(&(semaphore)->wait_queue)
```

Semaphores also have one other interesting property — they may be converted to and from ports. The functions in `osfmk/kern/ipc_sync.c` allow this. This functionality, however, is not exposed to user mode, and is not used in the kernel proper.

Lock Set Object

Tasks can utilize lock sets at the user mode level. These are conceptually arrays of locks (actually, mutexes), which can be acquired by a given lock ID. The locks can also be given — handed off — to other threads. Handing off will block the handing thread and wake up the receiving thread.

The lock sets are essentially wrappers over the kernel's mutexes, `lck_mtx_t`'s, as shown in the Figure 10-1:

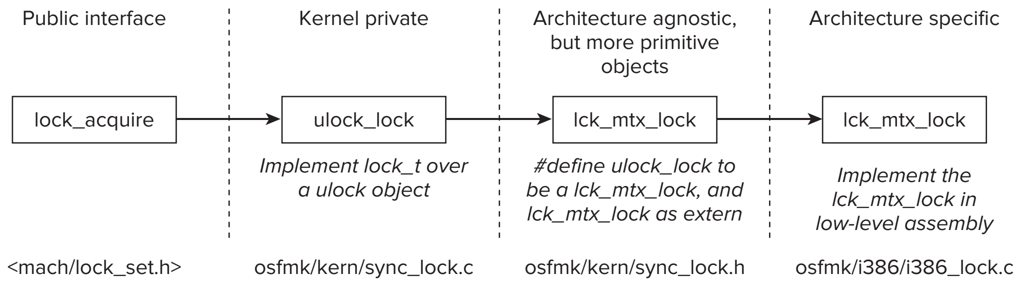


FIGURE 10-1: Lock set implementation over mutexes

The APIs are listed in Table 10-12:

TABLE 10-12: Lock Set APIs (visible in user mode)

MACH LOCK SET API	USED TO
`lock_set_create``(task_t` *`t`*`,` `lock_set_t` *`lock_set`*`,` `int` *`count`*`,` `int` *`policy`*`);`	Create a lock set `lock_set` for task `t`, with up to count locks. Wake up threads obtaining lock in set according to policy: `SYNC_POLICY_FIFO`: queued `SYS_POLICY_FIXED_PRIORITY`: by priority
`lock_set_destroy``(task_t` *`t`*`,` `lock_set_t` *`lock_set`*`);`	Destroy a lock set and any locks it may contain.
`lock_acquire` `(lock_set_t` *`lock_set`*`,` `int` *`lock_id`*`);`	Acquire lock `lock_id` in lock set `lock_set`. This function may block indefinitely.
`lock_release` `(lock_set_t` *`lock_set`*`,` `int` *`lock_id`*`);`	Release lock `lock_id` in lock set `lock_set`, if held.

lock_try (lock_set_t *lock_set*, int *lock_id*);	Try to acquire, but fail if lock is already held with KERN_LOCK_OWNED, rather than block until available.
lock_make_stable (lock_set_t *lock_set*, int *lock_id*);	Make a lock, which was acquired and returned KERN_LOCK_UNSTABLE, once again stable.
lock_handoff (lock_set_t *lock_set*, int *lock_id*);	Give a lock (which is currently owned) to another thread.
lock_handoff_accept (lock_set_t *lock_set*, int *lock_id*);	Accept a lock which was previously given with lock_handoff_accept.

The interesting aspect of locksets is that they allow the *handoff* of locks. This is the act of passing a lock from one task to another. Mach also uses handoff in the context of scheduling, allowing one thread to yield the processor but specify which thread to run in its stead.

MACHINE PRIMITIVES

Mach abstracts the machine it is operating on by several so called "machine primitives," which include the *host* (physical machine abstraction), clock (time keeping), processor (CPU), and *processor set* (logical groupings of CPUs). These are described next.

Host Object

Mach's most fundamental object is the "host," which represents the machine itself. The host object is a simple construct, defined in <osmfk/kern/host.h> as shown in Listing 10-7:

LISTING 10-7: Host abstraction definition from osfmk/kern/host.h

```
struct  host {
        decl_lck_mtx_data(,lock)              /* lock to protect exceptions */
        ipc_port_t special[HOST_MAX_SPECIAL_PORT + 1]; // ports such as priv, I/O,
        pager, struct exception_action exc_actions[EXC_TYPES_COUNT];
};
typedef struct host     host_data_t;
```

The host is really nothing more than a collection of "special ports," which are used to send the host various messages, and a collection of exception handlers (which are described later in this chapter). A lock is defined over the host to avoid concurrent access during exception processing.

The host structure serves three basic functions:

- **Provides machine information:** Mach provides a surprisingly rich set of API calls to query machine information, and all require obtaining the host port in order to function.
- **Provide access to subsystems:** Through the host abstraction, an application can request access to any of several "special" ports used by subsystems. Additionally, it is possible to gain access to all the other machine abstractions (notably, the `processor` and `processor_set`).
- **Provides default exception handling:** As shown later, exceptions are escalated from the thread level to the process (task) level, and — if not handled — to the host level for generic handling.

The important aspect of the host APIs is that they provide information that is virtually unobtainable in other ways. The Mach APIs provide the most straightforward way to get information about kernel modules, memory tables, and other aspects, which POSIX (and, therefore, the BSD layer) does not offer. Table 10-13 lists these APIs:

TABLE 10-13: Mach host APIs

MACH HOST API	USED TO
`host_info` `(host_t host,` ` host_flavor_t flavor,` ` host_info_t host_info_out,` ` mach_msg_type_number_t` ` *host_info_outCnt`	Get various system information, according to `flavor`: `HOST_BASIC_INFO`: Basic information on the host — `host_info_out` is a `host_basic_info`. `HOST_SCHED_INFO`: `host_info_out` is a `host_sched_info` specifying scheduling information.
`host_processor_info` `(host_t host,` `processor_flavor_t flavor,` `natural_t *processorCount,` `processor_info_array_t *info,` `mach_msg_type_number_t *count);`	Get detail on the host processors: *`processorCount`* will hold the number of processors, and information (according to `flavor`) will be returned in `info`, an array of `infoCnt` bytes.
`host_get_clock_service` `(host_t host,` ` clock_id_t clock_id,` ` clock_serv_t *clock_serv);`	Get a pointer to the host's clock service (discussed later).
`kmod_get_info` `(host_t host,` `kmod_args_t *modules,` `mach_msg_type_number_t *modulesCnt);`	Get a list of kernel modules on the host — deprecated in Snow Leopard, and unsupported in Lion and iOS.

host_virtual_physical_table_info (host_t *host*, hash_info_bucket_array_t **info*, mach_msg_type_number_t **infoCnt*);	Virtual to physical address mapping tables. Only supported on debug kernels (#if MACH_VM_DEBUG).
host_statistics (host_t *host_priv*, host_flavor_t *flavor*, host_info_t *host_info_out*, mach_msg_type_number_t *hioCnt*);	Obtain various statistics about host. A host_statistics64 function also exists.
host_lockgroup_info (host_t host, lockgroup_info_array_t *lockgroup_info, mach_msg_type_number_t **lgiCnt*);	Obtain information about kernel lock groups (internal lock objects in kernel).

OS X and jailbroken iOS contain a hostinfo(1) command, which displays the mach_host_info_t structure information in user-friendly form as shown in Listings 10-8a through 10-8c:

LISTING 10-8A: hostinfo(1) on the author's MacBook Air

```
root@Ergo (/)# hostinfo
Mach kernel version:
     Darwin Kernel Version 10.8.0: Tue Jun  7 16:33:36 PDT 2011; root:xnu-1504.15.3~1/
     RELEASE_I386
Kernel configured for up to 2 processors.
2 processors are physically available.
2 processors are logically available.
Processor type: i486 (Intel 80486)
Processors active: 0 1
Primary memory available: 4.00 gigabytes
Default processor set: 74 tasks, 337 threads, 2 processors
Load average: 1.29, Mach factor: 1.14
```

LISTING 10-8B: hostinfo(1) on an iPod Touch

```
Podicum:~ root# hostinfo
Mach kernel version:
       Darwin Kernel Version 11.0.0: Thu Sep 15 23:34:16 PDT 2011; root:xnu-1878.4.43~2/
       RELEASE_ARM_S5L8930X
Kernel configured for a single processor only.
1 processor is physically available.
1 processor is logically available.
```

continues

LISTING 10-8B *(continued)*

```
Processor type: armv7 (arm v7)
Processor active: 0
Primary memory available: 248.95 megabytes
Default processor set: 33 tasks, 233 threads, 1 processors
Load average: 0.46, Mach factor: 0.58
```

LISTING 10-8C: hostinfo(1) on the iPad 2 (Note two processors = 2 cores)

```
Padishah:~ root# hostinfo
Mach kernel version:
        Darwin Kernel Version 11.0.0: Wed Mar 30 18:52:42 PDT 2011; root:xnu-1735.46~10/
        RELEASE_ARM_S5L8940X
Kernel configured for up to 2 processors.
2 processors are physically available.
2 processors are logically available.
Processor type: armv7 (arm v7)
Processors active: 0 1
Primary memory available: 502.00 megabytes
Default processor set: 34 tasks, 281 threads, 2 processors
Load average: 0.07, Mach factor: 1.92
```

These commands are a straightforward dump of the `host_basic_info` struct defined in `osfmk/mach/host_info.h` (and `<mach/host_info.h>`). If the “i486” processor type is somewhat surprising, it is because the APIs have not been updated in a long, long time.

Experiment: Using Host Functions to Obtain Information

Listing 10-9 shows how you can create a `hostinfo(1)` like utility using a few lines of code:

LISTING 10-9: The source of a hostinfo(1) like utility.C

```
#include <mach/mach.h>
#include <stdio.h>

// A quick & dirty hostinfo(1) like utility

int main(int argc, char **argv)
{

        mach_port_t     self = host_self();
        kern_return_t   rc;
        char            buf[1024]; // suffices. Better code would sizeof(..info)
        host_basic_info_t hi;
        int len = 1024;

    // Getting the host info is simply a matter of calling host_info
    // on the host_self(). We do not need the privileged host port for
    // this..
        rc = host_info (self,              // host_t host,
                            HOST_BASIC_INFO,   // host_flavor_t flavor,
                            (host_info_t) buf, // host_info_t host_info_out,
```

```
                &len); // mach_msg_type_number_t *host_info_outCnt

        if (rc != 0) { fprintf(stderr,"Nope\n"); return(1);}


    hi = (host_basic_info_t) buf; // type cast, so we can print fields

  // and print fields..
        printf ("CPUs:\t\t %d/%d\n",  hi->avail_cpus, hi->max_cpus);
        printf ("Physical CPUs:\t %d/%d\n",  hi->physical_cpu, hi->physical_cpu_max);
        printf ("Logical CPUs:\t %d/%d\n",   hi->logical_cpu,  hi->logical_cpu_max);
        printf ("CPU type:\t %d/%d, Threadtype: %d\n", hi->cpu_type,
                                                   hi->cpu_subtype, hi->cpu_threadtype);

    // Note memory_size is a signed 32-bit! Max value is 2GB, then it flips to negative
    printf ("Memory size:\t %d/%ld\n", hi->memory_size, hi->max_mem);


        return(0);
}
```

This listing will compile cleanly on OS X and iOS. The "physical/logical" distinction between the CPUs doesn't really work, as Mach can't tell the difference. The reader is encouraged to add other `_info` like utilities as an exercise.

Host Special Ports

The Mach host object also contains "special" ports. These, as you can see in Listing 10-7, are maintained in an internal array — so merely having the host port is insufficient to obtain access to them. A call to `host_get_special_port` must be made and, as most specific ports are well known, macros exist to obtain each of them, as shown in Listing 10-10:

LISTING 10-10: Host special ports and the macros to get them (osfmk/mach/host_special_ports.h)

```
/*
 * Always provided by kernel (cannot be set from user-space).
 */
#define HOST_PORT                        1
#define HOST_PRIV_PORT                   2
#define HOST_IO_MASTER_PORT              3 // used by IOKit (see chapter 13)
#define HOST_MAX_SPECIAL_KERNEL_PORT     7 /* room to grow */

/*
 * Not provided by kernel
 */

#define HOST_DYNAMIC_PAGER_PORT         (1 + HOST_MAX_SPECIAL_KERNEL_PORT)
#define HOST_AUDIT_CONTROL_PORT         (2 + HOST_MAX_SPECIAL_KERNEL_PORT)
#define HOST_USER_NOTIFICATION_PORT     (3 + HOST_MAX_SPECIAL_KERNEL_PORT)
#define HOST_AUTOMOUNTD_PORT            (4 + HOST_MAX_SPECIAL_KERNEL_PORT)
#define HOST_LOCKD_PORT                 (5 + HOST_MAX_SPECIAL_KERNEL_PORT)
#define HOST_SEATBELT_PORT              (7 + HOST_MAX_SPECIAL_KERNEL_PORT)
#define HOST_KEXTD_PORT                 (8 + HOST_MAX_SPECIAL_KERNEL_PORT)
```

continues

LISTING 10-10 *(continued)*

```
#define HOST_CHUD_PORT                 (9 + HOST_MAX_SPECIAL_KERNEL_PORT)
#define HOST_UNFREED_PORT              (10 + HOST_MAX_SPECIAL_KERNEL_PORT)
#define HOST_AMFID_PORT                (11 + HOST_MAX_SPECIAL_KERNEL_PORT)
#define HOST_GSSD_PORT                 (12 + HOST_MAX_SPECIAL_KERNEL_PORT) // Lion
#define HOST_MAX_SPECIAL_PORT          (13 + HOST_MAX_SPECIAL_KERNEL_PORT)
                                       /* room to grow here as well */

/*
 * Special node identifier to always represent the local node.
 */
#define HOST_LOCAL_NODE                 -1

/*
 * Definitions for ease of use.
 *
 * In the get call, the host parameter can be any host, but will generally
 * be the local node host port. In the set call, the host must the per-node
 * host port for the node being affected.
 */

#define host_get_host_port(host, port)  \
        (host_get_special_port((host),  \
        HOST_LOCAL_NODE, HOST_PORT, (port)))
#define host_set_host_port(host, port) (KERN_INVALID_ARGUMENT)

#define host_get_host_priv_port(host, port)     \
        (host_get_special_port((host),          \
        HOST_LOCAL_NODE, HOST_PRIV_PORT, (port)))
#define host_set_host_priv_port(host, port) (KERN_INVALID_ARGUMENT)

#define host_get_io_master_port(host, port)     \
        (host_get_special_port((host),          \
        HOST_LOCAL_NODE, HOST_IO_MASTER_PORT, (port)))
#define host_set_io_master_port(host, port) (KERN_INVALID_ARGUMENT)

... (others defined similarly)…
```

Not all the special ports are necessarily kernel ones. In fact, most of those `#define`'d in Listing 10-10 are in user mode, owned by specific daemon processes. These user-mode special ports are listed in Table 10-15:

TABLE 10-15: Host special ports claimed by user mode processes

CONSTANT	USED FOR
`HOST_DYNAMIC_PAGER_PORT(8)`	OS X: Used by dynamic_pager. Serves swap file resizing requests (described in Chapter 11).
`HOST_AUDIT_CONTROL(9)`	OS X: used by `auditd` (described in Chapter 3).
`HOST_USER_NOTIFICATION_PORT(10)`	OS X: Used by the `kuncd`, Kernel/User Notification Center daemon. This is a daemon which receives requests from kernel mode and displays dialogs to the user.

HOST_AUTOMOUNTD_PORT(11)	OS X: used by the file system automount daemon.
HOST_LOCKD_PORT(12)	OS X: used by the RPC lockd.
HOST_SEATBELT_PORT(14)	Seatbelt — the former name of the Sandbox API. Used by the sandboxd.
HOST_KEXTD_PORT(15)	OS X: The Kernel Extension Daemon — Responsible for centralizing kernel extension load requests from user mode, and assisting the kernel when loading multiple kexts. Unused in iOS.
HOST_CHUD_PORT(16)	The Computer Hardware Understanding Port, reserved for CHUD programs, for low-level profiling and diagnostics. Used by appleprofilepolicyd.
HOST_UNFREED_PORT(17)	iOS: Used by fairplayd, Apple's DRM enforcer.
HOST_AMFID_PORT(18)	iOS: Used by amfid and AppleMobileFileIntegrity, which enforces code signatures and entitlements.
HOST_GSSD_PORT(19)	As of Lion: Used by GSS. Before Lion, this was a task-level special port (#8). Unused in iOS.

The special ports can be requested from launchd, in the MachServices key, by specifying the HostSpecialPort key. Listing 10-11 shows the sandboxd requesting the HOST_SEATBELT_PORT on OS X or iOS:

LISTING 10-11: Requesting HOST_SEATBELT_PORT (#14) in com.apple.sandboxd.plist

```
...
     <key>MachServices</key>
     <dict>
           <key>com.apple.sandboxd</key>
           <dict>
                   <key>HostSpecialPort</key>
                   <integer>14</integer>
           </dict>
     </dict>
...
```

Whether they are kernel-provided or external, the same function can be used to retrieve special ports, however. This function is host_get_special_port(), which is defined in osfmk/kern/host.c, and shown in Listing 10-12:

LISTING 10-12: host_get_special_port(), as defined in osfmk/kern/host.c

```
host_get_special_port(
        host_priv_t     host_priv,
        __unused int    node,
```

continues

LISTING 10-12 *(continued)*

```
        int             id,
        ipc_port_t      *portp)
{
        ipc_port_t      port;

        if (host_priv == HOST_PRIV_NULL ||
            id == HOST_SECURITY_PORT || id > HOST_MAX_SPECIAL_PORT || id < 0)
                return KERN_INVALID_ARGUMENT;

        host_lock(host_priv);
        port = realhost.special[id];
        *portp = ipc_port_copy_send(port);
        host_unlock(host_priv);

        return KERN_SUCCESS;
}
```

Host Privileged Operations

The most important special host port is the host's privileged port. It is a prerequisite to quite a few operations, which are deemed "privileged" and require accessing special ports. While anyone is able to get the host port by means of `mach_host_self()`, discussed previously, only privileged users can get the privileged port by calling `host_get_host_priv_port()`, shown in Listing 10-8. Once the port is obtained, it can be used in any of the calls shown in Table 10-16, defined in <mach/host_priv.h>:

TABLE 10-16: Functions in <mach/host_priv.h>

MACH HOST_PRIV API	USED FOR
host_get_boot_info (host_priv_t *host_priv*, kernel_boot_info_t *info*)	Return boot information in `info`. Actual implementation is machine-specific. OS X's (in `osfmk/i386/AT386/model_dep.c`) returns an empty string.
host_reboot (host_priv_t *hp*, int *options*);	Reboot host, according to `options`. Currently defined are `HOST_REBOOT_DEBUGGER` (to invoke the kernel debugger) and `HOST_REBOOT_UPSDELAY`. This function calls on the Platform Expert to do the actual work of halting/restarting.
host_priv_statistics (host_priv_t *host_priv*, host_flavor_t *flavor*, host_info_t *host_info_out*, mach_msg_type_number_t **hioCnt*);	In OS X and iOS, same as `host_statistics`.

host_default_memory_manager `(host_priv_t` *`host_priv`*`,` `memory_object_default_t *`*`def`*`,` `memory_object_cluster_size_t cluster_size);`	Register default pager task (discussed in Chapter 12).
[mach]_vm_wire `(host_priv_t` *`host_priv`*`,` `vm_map_t task,` `vm_address_t address,` `vm_size_t size,` `vm_prot_t desired);`	Change residency of memory range (*`address–address+size`*) resident in VM map of `task` according to `desired`. This is very similar to `mlock(2)`. To unwire (`munlock(2)`), specify `VM_PROT_NONE` in flags. Note, that while BSD treats `mlock(2)` as a per-process API, in Mach this is a host level call, as it affects the entire machine's physical memory. This calls `mach_vm_wire()` internally.
vm_allocate_cpm `(host_priv_t host_priv,` `vm_map_t task,` `vm_address_t *address,` `vm_size_t size,` `int flags);`	Experimental API meant to offer a contiguous physical memory allocator.
host_processors `(host_t host_priv,` `processor_port_array_t pl,` `mach_msg_type_number_t *count);`	Populate array of *`count`* processors ports *`pl`* on the system.
host_get_clock_control `(host_priv host_priv,` `clock_id_t id,` `clock_ctrl_t control);`	Set `control` to be a handle (send right) to the clock specified by `id`.
kmod_create`(...);` **kmod_destroy**`(...);` **kmod_control**`(...);`	Mach kernel module support. No longer supported in either OS X or iOS.
host_get_special_port `(host_priv_t` *`host_priv`*`,` `int` *`node`*`,` `int` *`which`*`,` `mach_port_t *`*`port`*`);`	Get or set any of the host's special ports (discussed in the last section).

continues

TABLE 10-16 *(continued)*

MACH HOST_PRIV API	USED FOR
host_set_special_port (host_priv_t *host_priv*, int which, mach_port_t *port*);	
host_set_exception_ports (host_priv_t *host_priv*, exception_mask_t *exc_mask*, exception_mask_array_t *masks*, mach_msg_type_number_t **mCnt*, exception_handler_array_t *old*, exception_behavior_array_t *oldb*, exception_flavor_array_t *oldf*); **host_get_exception_ports** (...); **host_swap_exception_ports** (...);	Get/Set or swap between the host-level exception handlers (discussed under "Exceptions," in the next chapter).
host_load_symbol_table	As noted in the sources — "This has never and will never be supported on Mac OS X" (would have loaded the kernel symbol table into kernel debugger).
host_processor_sets (host_priv_t *host_priv*, processor_set_name_port_array_t *processor_set_name_list*, mach_msg_type_number_t **count*);	Similar to `host_processor` but get array of `processor_sets`. Processor sets are primitives that group the machine's CPUs. They are discussed later.
set_dp_control_port (host_priv_t *host*, mach_port_t *control_port*); **get_dp_control_port** (host_priv_t *host*, mach_port_t **contorl_port*);	Get or set Dynamic Pager control port. The Dynamic Pager is discussed in Chapter 12.
host_set_UNDServer (host_priv_t *host_priv*, UNDServerRef *server*) **host_get_UNDServer** (host_priv_t *host_priv*, UNDServerRef **server*)	Wrappers over `host_get/set_user_notification_port`. Used in XNU's UNC mechanisms to export kernel messages to user mode. This is a deprecated API which allows drivers and other kernel-level code to display GUI prompts.

kext_request (host_priv_t *hp*, uint32_t *clientLogSpec*, vm_offset_t *requestIn*, mach_msg_type_number_t *reqLen*, vm_offset_t * *responseOut*, mach_msg_type_number_t * *lenOut*, vm_offset_t * *logDataOut*, mach_msg_type_number_t * *ldoLen*, kern_return_t * *op_result*)	Apple-specific extension to support Kernel Extensions — used in place of the kmod_* api to insert kexts. The message is used to load, query and remove kernel extensions (described in detail in Chapter 18).

An interesting observation is that, for a privileged user, the host's "regular" and "privileged" port appear alike (i.e. comparing the port numbers reveals they are very much the same), whereas the unprivileged user gets a "0" when attempting to retrieve the privileged port.

Experiment: Rebooting Using the Privileged Port

The following (very simple) listing (Listing 10-13) shows how to reboot the system if access to the privileged port can be obtained. Naturally, you will need root permissions to access this (but do be careful, as — unlike the OS X GUI, which gives you a chance to change your mind — this will halt/restart your machine without warning):

LISTING 10-13: Rebooting the system, via the host API

```
#include <mach/mach.h>
void main()
{

    mach_port_t     h = mach_host_self();
    mach_port_t     hp;
    kern_return_t   rc;

    /* request host privileged port. Will only work if we are root   */
    /* Note, this is the "right" way of doing it.. but we could also */
    /* use a short cut, left as an exercise                          */
    rc = host_get_host_priv_port (h, &hp);

    if (rc == KERN_SUCCESS) host_reboot(hp, 0);

        // If we are root, this won't even be reached.
        printf ("sorry\n");

}
```

As an exercise, run the preceding program, but change the hp parameter — the privileged host port — to h. What happens? What does that tell you about the necessity of host_get_host_priv_port? Validate this by examining host_priv_self() and host_self() in osfmk/kern/host.c.

Clock Object

The Mach kernel provides a simple abstraction of a "clock" object. This object is used for timekeeping and alarms, and is defined in `osfmk/kern/clock.h`, shown in Listing 10-14:

LISTING 10-14: The clock object, from osfmk/kern/clock.h

```
struct  clock_ops {
        int             (*c_config)(void);            /* configuration */
        int             (*c_init)(void);              /* initialize */
        kern_return_t   (*c_gettime)(   /* get time */
                                mach_timespec_t                 *cur_time);
        kern_return_t   (*c_getattr)(   /* get attributes */
                         clock_flavor_t           flavor,
                         clock_attr_t             attr,
                         mach_msg_type_number_t  *count);

struct  clock {
        clock_ops_t                     cl_ops;                 /* operations list */
        struct ipc_port         *cl_service;    /* service port */
        struct ipc_port         *cl_control;    /* control port */
};
```

As can be seen from the listing, the clock is a simple object with two ports — one for "service" functions (e.g. time-telling or alarms), and the other for "control" functions, such as setting the time of day.

From user mode, however, the visible API is fairly basic, as detailed in `<mach/clock.h>`, and shown in Table 10-17:

TABLE 10-17: The Mach user-mode visible APIs

MACH CLOCK API	USED FOR
`clock_get_time` `(clock_serv_t *clock_serv*, mach_timespec_t **cur_time*);`	Get the current time from `clock_serv` into `cur_time`.
`clock_get_attributes` `(clock_serv_t *clock_serv*, clock_flavor_t *flavor*, clock_attr_t *clock_attr*, mach_msg_type_number_t **clock_attrCnt*);`	Get clock `clock_serv`'s attribute, of selected flavor, into `clock_attr_t`. Currently defined attributes: `CLOCK_GET_TIME_RES` `CLOCK_ALARM_CURRES` `CLOCK_ALARM_MINRES` `CLOCK_ALARM_MAXRES`.
`clock_alarm` `(clock_serv_t *clock_serv*, alarm_type_t *alarm_type*, mach_timespec_t *alarm_time*, clock_reply_t *alarm_port*);`	Request an alarm message from the`clock_serv`. This message will be sent to the`alarm_port` at the specified `alarm_time`. Time is specified as `TIME_ABSOLUTE` or `TIME_RELATIVE`.

In all the API functions shown, the client first obtains a handle to the clock (`clock_serv_t`) by calling `host_get_clock_service`. Mach exposes two types of clocks — `SYSTEM_CLOCK/REALTIME_CLOCK`, and `CALENDAR_CLOCK` (`SYSTEM` and `REALTIME` are both the same clock) — and the caller needs to specify the clock type as the second parameter to this call. Whereas `SYSTEM_CLOCK` keeps the time since boot, `CALENDAR_CLOCK` is synchronized with the machine's RTC to provide both the time and date.

Internally, however, there are quite a few clock functions. XNU provides a newer API than the original Mach and has deprecated the original API to "old" status, so if you examine the sources you are likely to see references to both the new functions and their "old" counterparts.

All the clocks are created as part of the kernel's initialization process. The clocks are defined in a global `clock_list` (in `osfmk/i386/AT386/conf.c`):

```
struct  clock   clock_list[] = {

        /* SYSTEM_CLOCK */
        { &sysclk_ops, 0, 0 },

        /* CALENDAR_CLOCK */
        { &calend_ops, 0, 0 }
};
int     clock_count = sizeof(clock_list) / sizeof(clock_list[0]);
```

The `clock_init()` function, called from `kernel_bootstrap()`, falls through to `clock_oldinit()` and initializes each clock in the list by calling its `c_init` function. For the system clock, which is the important abstraction of the system's timer tick, the `sysclk_ops` are defined in `osfmk/kern/clock_oldops.c`, as follows:

```
struct clock_ops sysclk_ops = {
        rtclock_config,             // the c_config member
        rtclock_init,               // the c_init member
        rtclock_gettime,
        rtclock_getattr,
};
```

The `kernel_bootstrap_thread()` then calls `clock_service_create()`, which in turn calls `ipc_clock_init()` to create each clock's service and configuration port, and then `ipc_clock_enable()` to enable IPC access to it. Finally, it wraps up by allocating a global `alarm_zone` called "alarms," which is used for clock alarms.

Clock alarms are really just wrappers over the well-known Mach messages. These alarms, defined in `osfmk/kern/clock_oldops.c`, are stored in a linked list of `struct alarm`, defined as follows:

```
struct alarm {
        struct  alarm   *al_next;                       /* next alarm in chain */
        struct  alarm   *al_prev;                       /* previous alarm in chain */
        int                al_status;                   /* alarm status */
        mach_timespec_t al_time;                        /* alarm time */
        struct {                                        /* message alarm data */
                int             type;                   /* alarm type */
                ipc_port_t       port;                  /* alarm port */
                mach_msg_type_name_t   port_type;       /* alarm port type */
```

```
                struct  clock   *clock;                 /* alarm clock */
                void                    *data;          /* alarm data */
        } al_alrm;
```

The `clock_alarm` function, callable from both user and kernel mode, validates the arguments and sets up an alarm by obtaining the global `alarm_lock`, allocating a new alarm object from the `alarm_zone`, copying the arguments to it, and posting it using `post_alarm`, which in turn calls `set_alarm` to set the `alarm_expire_timer` to the time specified in the alarm, converted to absolute time.

When the alarm expires, the clock thread wakes up into `alarm_done`, which delivers the alarm to the `al_port` specified — i.e. sends a message by calling `clock_alarm_reply()`.

The most important internal API clocks offer is `clock_deadline_for_periodic_event`: This API is used by schedulers (discussed next chapter) to set up a recurring notification — and thus, a call-back into the scheduler, which keeps the system's multitasking engine running.

Processor Object

The processor object represents a logical CPU or core present on the machine. In today's multicore default architecture, each core is considered to be a CPU, and Mach does not make the distinction between the two terms. Processors are assigned to processor sets, which are logical groupings of one or more processors.

The processor is a simple abstraction of a CPU, used by Mach for basic operations, such as starting and stopping a CPU or core and dispatching threads to it. The structure is defined in `osfmk/kern/processor.h` and is fairly well commented, as shown in Listing 10-15:

LISTING 10-15: The processor object, from osfmk/kern/processor.h

```
struct processor {
        queue_chain_t  processor_queue;/* idle/active queue link,
                                        * MUST remain the first element */
        int            state;          /* one of OFFLINE,SHUTDOWN,START,INACTIVE,
                                        * IDLE, DISPATCHING, or RUNNING */
        struct thread *active_thread,  /* thread running on processor */
                      *next_thread,    /* next thread when dispatched */
                      *idle_thread;    /* this processor's idle thread. */

        processor_set_t         processor_set;  /* assigned set (discussed later) */
        int                     current_pri;    /* priority of current thread */
        sched_mode_t            current_thmode; /* sched mode of current thread */
        int                     cpu_id;         /* platform numeric id */

        timer_call_data_t       quantum_timer;  /* timer for quantum expiration */
        uint64_t                quantum_end;    /* time when current quantum ends */
        uint64_t                last_dispatch;  /* time of last dispatch */

        uint64_t                deadline;      /* current deadline */
        int                     timeslice;     /* quanta before timeslice ends */

       /* Specific thread schedulers defined in the mach kernel require expanding this
```

```
        * structure with their own fields—this will be explained next chapter
        */
#if defined(CONFIG_SCHED_TRADITIONAL) || defined(CONFIG_SCHED_FIXEDPRIORITY)
        struct run_queue        runq;                   /* runq for this processor */
        int                     runq_bound_count; /* # of threads bound to this
                                                   * processor */
#endif
#if defined(CONFIG_SCHED_GRRR)
        struct grrr_run_queue   grrr_runq;      /* Group Ratio Round-Robin runq */
#endif
        processor_meta_t        processor_meta; /* meta data on processor */

        struct ipc_port *       processor_self; /* port for operations */

        processor_t             processor_list; /* all existing processors */
        processor_data_t        processor_data; /* per-processor data */
};
```

Most important in the processor object is the `runq` element, which is the processor's local queue of threads that have been dispatched to it. Run queues are discussed in Chapter 11.

The processors on a host can be obtained by a call to `host_processors()`, which will return an array of `processor_t` objects. Mach defines the operations shown in Table 10-18, on the `processor_t`:

TABLE 10-18: Processor operations

MACH PROCESSOR API	USED TO
`processor_start` `(processor_t` *`p`*`);`	Start the processor or core *`p`*. Cannot start an already active processor.
`processor_exit``(processor_t` *`p`*`)`	Exit (shut down) the processor or core *`p`*.
`processor_info``(processor_t p,` `processor_flavor_t` *`flavor`*`,` `host_t *`*`host`*`,` `processor_info_t` *`pi_out`*`,` `mach_msg_type_number_t *`*`outCnt`*`)`	Return information on processor according to `flavor` requested. Flavors supported are `PROCESSOR_BASIC_INFO` and `PROCESSOR_CPU_LOAD_INFO`. Information will be placed into `pi_out` and will be `outCnt` bytes.
`processor_control``(processor_t` *`p`*`,` `processor_info_t` *`cmd`*`,` `mach_msg_type_number_t` *`cnt`*`);`	Pass *`cnt`* commands (in *`cmd`*) to processor *`p`*. Not implemented on Intel architectures.
`processor_assign` `(processor_t` *`p`*`,` `processor_set_t` *`new_set`*`, boolean_t` *`wait`*`);`	Assign processor *`p`* to processor set *`new_set`*, possibly waiting until the process queue is empty.
`processor_get_assignment` `(processor_t` *`p`*`,` `processor_set_name_t *`*`pset`*`);`	Get the `pset` the current processor is assigned to.

The APIs in the preceding table are simple, yet quite powerful. They can be used, among other things, to display detailed information about the processors in the system, as in the next experiment.

Experiment: Fun with Mach processor_ts

Listing 10-16 demonstrates using `processor_info()` to display the information on the current processors in a system:

LISTING 10-16: Using processor_info()

```
#include <stdio.h>           // fprintf, stderr, and friends
#include <mach/mach.h>       // Generic Mach stuff, like kern_return_t
#include <mach/processor.h>  // For the processor_* APIs
#include <mach-o/arch.h>     // For NXArch

int main(void) {

        kern_return_t    kr;
        host_name_port_t host = mach_host_self();
        host_priv_t      host_priv;
        processor_port_array_t processors;
        natural_t              count, infoCount;
        processor_basic_info_data_t basicInfo;
        int                    p;

        // First, get the privileged port - otherwise we can't query the processors

        kr = host_get_host_priv_port(host, &host_priv);

        if (kr != KERN_SUCCESS)
          { fprintf(stderr, "host_get_host_priv_port %d (you should be root)", kr);
          exit(1); }

        // If we're here, we can try to get the process array
        kr = host_processors (host_priv, &processors, &count);
        if (kr != KERN_SUCCESS) { fprintf(stderr, "host_processors %d", kr); exit(1); }

        // And if we got this far, we have it! Iterate, then:
        for (p = 0; p < count; p++)
        {
                // infoCount is in/out, so we have to reset it on each iteration
                infoCount = PROCESSOR_BASIC_INFO_COUNT;

                // Ask for BASIC_INFO. It is left to the reader as an exercise
                // to implement CPU_LOAD_INFO
                kr = processor_info (processors[p],        // the processor_t
                                     PROCESSOR_BASIC_INFO, // Information requested
                                    &host,                 // The host
                 (processor_info_t) &basicInfo,            // Information returned here
                                    &infoCount);           // Sizeof(basicInfo) (in/out)


                if (kr != KERN_SUCCESS) {fprintf(stderr, "?!\n"); exit(3);}
```

```
                // Dump to screen. We use NX APIs to resolve the cpu type and subtype
                printf("%s processor %s in slot %d\n",
                     (basicInfo.is_master ? "Master" : "Slave"),
                     NXGetArchInfoFromCpuType(basicInfo.cpu_type,
                                              basicInfo.cpu_subtype)->description,
                     basicInfo.slot_num);
        }
}
```

As suggested in the comments, you are encouraged to adapt this exercise to `PROCESSOR_CPU_LOAD_INFO`. If you look at `<mach/processor_info.h>`, you will see references to two other informational types: `PROCESSOR_PM_REGS_INFO` and `PROCESSOR_TEMPERATURE` — but neither are supported on Intel or ARM. ARM supports the `PROCESSOR_CPU_STAT` flavor, which allows obtaining processor exception statistics (defined in `<mach/arm/processor_info.h>`, in the iPhone SDK).

Another interesting feature enabled by the Mach APIs is the starting and stopping (shutting down) of processors on-the-fly. Consider the following program (Listing 10-17):

LISTING 10-17: A program to stop all but the main processor on a system

```
#include <mach/mach.h>
#include <stdio.h>
void main(int argc, char **argv)
{

  host_t        myhost = mach_host_self();
  host_t        mypriv;

  int           proc;
  kern_return_t kr;
  processor_port_array_t processorPorts;
  mach_msg_type_number_t        procCount;


  kr = host_get_host_priv_port(myhost,&mypriv);
  if (kr ) { printf ("host_get_host_priv_port: %d\n", kr); exit(1);}


   // Get the ports of all the processors in the system
   kr = host_processors (mypriv,         // host_t host,
                      &processorPorts, // processor_port_array_t *out_processor_ports,
                      &procCount);     // mach_msg_type_number_t *out_processorCnt

   if (kr) { printf ("host_processors: %d\n", kr); exit(2);}

   printf ("Got %d processors . kr %d\n", procCount, kr);
   for (proc = 0 ; proc <procCount; proc++)
        {
                printf ("Processor %d\n", processorPorts[proc]);
                // you really want to leave proc 0 active!
                if (proc > 0) { processor_exit(processorPorts[proc]);
                                if (kr != KERN_SUCCESS) printf ("Unable to stop %d\n",
                                proc);}
        }
```

You can easily adapt the following program (on a multi-core CPU or SMP system) to selectively disable or enable processors. It's worth stating the obvious — that it is possible to modify this program to stop all processors in your system, which will require you to reboot. Be warned.

Processor Set Object

One or more `processor_t` objects can be grouped into a *processor set*, or a *pset* (this is the `processor_set` member of the `processor` object), shown in Listing 10-18. A processor set is a logically coupled group of processors and allows Mach to efficiently scale to SMP architectures by using the set as a container for related processors.

Processors in a pset are maintained in one of two queues: an `active_queue`, for those processors that are currently executing threads, and an `idle_queue`, for processors that are idle (i.e. executing the `idle_thread`). The processor set also has a global `run_queue` (`pset_runq`), which contains threads to execute on the set's processors. Like all other objects, processor sets expose ports: `pset_self`, — for operations on the set, and `pset_name_self`, used for operations on the processor set.

LISTING 10-18: processor_set definition (from osfmk/kern/processor.h)

```
struct processor_set {
        queue_head_t            active_queue;   /* active processors */
        queue_head_t            idle_queue;     /* idle processors */

        processor_t                     low_pri, low_count;

        int                                     online_processor_count;

        int                                     cpu_set_low, cpu_set_hi;
        int                                     cpu_set_count;

        decl_simple_lock_data(,sched_lock)      /* lock for above */

#if defined(CONFIG_SCHED_TRADITIONAL) || defined(CONFIG_SCHED_FIXEDPRIORITY)
        struct run_queue        pset_runq;      /* runq for this processor set */
        int                                     pset_runq_bound_count;
                /* # of threads in runq bound to any processor in pset */
#endif

        struct ipc_port *       pset_self;      /* port for operations */
        struct ipc_port *       pset_name_self; /* port for information */

        processor_set_t         pset_list;      /* chain of associated psets */
        pset_node_t             node;
}
```

The operations provided by the processor set are shown in Table 9-10:

TABLE 9-10: Processor set APIs

MACH PROCESSOR SET API	USAGE
processor_set_statistics (processor_set_name_t pset, processor_set_flavor_t flavor, processor_set_info_t info_out, mach_msg_type_number_t *ioCnt)	Get processor set statistics of flavor about *pset* into info_out, with size ioCnt.
processor_set_destroy (processor_set_t pset);	Destroy the processor set *pset*. This function is not implemented (returns KERN_FAILURE). There is also a processor_set_create in kernel mode, though it, too, is unimplemented.
processor_set_max_priority (processor_set_t pset, int max_prio, boolean_t change_threads);	Set maximum priority on new threads assigned to pset. If change_threads is true, also set maximum priority for existing threads.
processor_set_policy_enable (processor_set_t pset, int policy);	Apply *policy* on processor set *pset*.
processor_set_policy_disable (processor_set_t pset, int policy, boolean_t change_threads);	Disable *policy* on processor set *pset*. Optionally, change thread behavior due to disablement.
processor_set_tasks (processor_set_t set, task_array_t *task_list, mach_msg_type_number_t *tlCnt);	Obtain the tlCnt tasks in the task_list array on processor_set.
processor_set_threads (processor_set_t set, thread_act_array_t *thread_list, mach_msg_type_number_t *tlCnt);	Same, for threads. Apparently intentionally unsupported on iOS.

continues

TABLE 9-10 *(continued)*

MACH PROCESSOR SET API	USAGE
kern_return_t **processor_set_policy_control** (processor_set_t *pset*, processor_set_flavor_t *flavor*, processor_set_info_t *info*, mach_msg_type_number_t *infoCnt*, boolean_t *change*);	Change policy on processor set. Unsupported (returns KERN_INVALID_ARGUMENT).
kern_return_t **processor_set_stack_usage** (processor_set_t pset, unsigned **ltotal*, vm_size_t **space*, vm_size_t **resident*, vm_size_t **maxusage*, vm_offset_t **maxstack*);	In debug kernels only.
processor_set_info (processor_set_name_t *pset*, int *flavor*, host_t **host*, processor_set_info_t *iout*, mach_msg_type_number_t **ioCnt*);	Obtain info of type *flavor* on *pset*. flavor can be one of many constants defined in `<mach/processor_info.h>`.

The `processor_set_tasks` and `processor_set_threads` are both internally implemented over an internal function, `processor_set_things`, which abstracts the array argument and takes an additional argument, "type," which specifies `THING_TASK` or `THING_THREAD`.

Experiment: Listing Tasks on the Current Processor Set

As an example, consider the following `ps` type process listing program (Listing 10-19), which takes a processor set object, and obtains a list of its tasks. For now, both tasks and threads are left as opaque structures. The listing will be developed in the next chapter, however, to further show detailed information for the tasks and threads.

LISTING 10-19: Displaying the tasks on the default processor set

```
void main(int argc, char **argv)
{

  host_t                 myhost = mach_host_self();
  mach_port_t            psDefault;
  mach_port_t            psDefault_control;
  task_array_t           tasks;
  mach_msg_type_number_t numTasks;
  int                    t;                    // a task index

  kern_return_t kr;

  // Get default processor set
  kr = processor_set_default(myhost, &psDefault);

  // Request control port
  kr = host_processor_set_priv(myhost, psDefault, &psDefault_control);
  if (kr != KERN_SUCCESS) { fprintf(stderr, "host_processor_set_priv - %d", kr);
  exit(1); }

  // Get tasks. Note this behaves a bit differently on iOS.
  // On OS X, you can also get the threads directly (processor_set_threads)

  kr = processor_set_tasks(psDefault_control, &tasks, &numTasks);
  if (kr != KERN_SUCCESS) { fprintf(stderr,"processor_set_tasks - %d\n",kr); exit(2); }


  // Iterate through tasks. For now, just display the task ports and their PIDs
  // We use "pid_for_task" to map a task port to its BSD process identifier

  for (t = 0; t < numTasks; i++)
        {
                int pid;
                pid_for_task(tasks[t], &pid);
                printf("Task: %d pid: %d\n", tasks[i],pid);

            // Stay tuned:
            // In the next chapter, this experiment will be expanded to list task
            // information, as well as the threads of each task
        }
  }
```

The output of the program in this example differs slightly in iOS: `processor_set_tasks` *will not return PID 0 (the* `kernel_task`*), as getting a handle to the kernel_task can open up potentially dangerous access to the kernel memory maps. Likewise,* `processor_set_threads` *is (apparently intentionally) not supported. There is therefore no legitimate way (jailbreaks not withstanding) to obtain kernel thread or memory handles from user mode — which is just the way Apple would like to keep it.*

SUMMARY

This chapter describes the basic principles of Mach. Ports are the underlying primitives on top of which virtually all other objects in Mach are implemented. Messages are passed between ports, and allow performing various operations on them. Additionally, messages enable IPC, a feature which is built into the Mach kernel, and extended using the synchronization primitives — spinlocks, mutexes, semaphores, and lock sets.

Mach also defines basic machine-level primitives — the host, clock, processor and `processor_set` abstractions. These are essential in performing various system-related tasks, primarily scheduling, which is covered in the next chapter.

REFERENCES

1. "Mach Tutorials," `http://www.cs.cmu.edu/afs/cs/project/mach/public/www/doc/tutorials.html`
2. Loepere, Keith, ed. "Mach 3 Kernel Interfaces," `http://www.cs.cmu.edu/afs/cs/project/mach/public/doc/osf/kernel_interface.ps`
3. Loepere, Keith. "Mach 3 Kernel Principles," `http://www.cs.cmu.edu/afs/cs/project/mach/public/doc/osf/kernel_principles.ps`
4. Apple Developer. "Mach Port Dumper Utility Sample Code," `https://developer.apple.com/library/mac/#samplecode/MachPortDump/Listings/MachPortDump_c.html`
5. Draves, et al. "The Mach Interface Generator," `http://www.cs.cmu.edu/afs/cs/project/mach/public/doc/unpublished/mig.ps`

11

Tempus Fugit — Mach Scheduling

Based on the core primitives discussed in Chapter 10, Mach provides many important features, almost all of which revolve around the management of system resources — hardware devices, virtual memory, and the CPU itself. Managing the CPU is also referred to as *scheduling*, because it refers to the operation of deciding which of the many programs vying for the CPU will get to use it and when.

This chapter focuses on scheduling. It is divided into the following sections:

- **Scheduling Primitives**: Describes tasks and threads, and the application programming interfaces (APIs) they offer.
- **Scheduling**: Discusses high-level concepts of scheduling, such as the algorithms.
- **Asynchronous Software Traps (ASTs)**: Explains Mach's concept of ASTs, which are instrumental in scheduling.
- **Exception Handling**: Discusses Mach's unique approach to hardware traps — exceptions.
- **Scheduling Algorithms**: Details Mach's default thread scheduler, as well as the scheduling framework, which allows extending or replacing the scheduler with other algorithm implementations.

SCHEDULING PRIMITIVES

Like all modern operating systems, the kernel sees threads, not processes. Mach, in fact, does not recognize the notion of a process as UN*X does. It employs a slightly different approach, using the concepts of the more lightweight *tasks* rather than processes. Classic UN*X uses a top-down approach, in which the basic object is a process that is further divided into one or more threads. Mach, on the other hand, uses a bottom-up approach in which the fundamental unit is a thread, and one or more threads are contained in a task.

Threads

A *thread* defines the atomic unit of execution in Mach. It represents the underlying machine register state and various scheduling statistics. Defined in `kern/thread.h`, a thread is designed to provide the maximum information required for scheduling, while maintaining the lowest overhead possible. (See Listing 11-1.)

LISTING 11-1: The Mach thread structure, from osfmk/kern/thread.h

```
struct thread {
        /*
         *      NOTE:   The runq field in the thread structure has an unusual
         *      locking protocol.  If its value is PROCESSOR_NULL, then it is
         *      locked by the thread_lock, but if its value is something else
         *      then it is locked by the associated run queue lock.
         *
         *      When the thread is on a wait queue, these first three fields
         *      are treated as an unofficial union with a wait_queue_element.
         *      If you change these, you must change that definition as well
         *      (kern/wait_queue.h).
         */
        /* Items examined often, modified infrequently */
        queue_chain_t   links;                  /* run/wait queue links */
        processor_t             runq;           /* run queue assignment */
        wait_queue_t    wait_queue;             /* wait queue we are currently on */
        event64_t               wait_event;     /* wait queue event */
        integer_t               options;        /* options set by thread itself */
#define TH_OPT_INTMASK          0x03            /* interrupt / abort level */
#define TH_OPT_VMPRIV           0x04            /* may allocate reserved memory */
#define TH_OPT_DTRACE           0x08            /* executing under dtrace_probe */
#define TH_OPT_SYSTEM_CRITICAL  0x10            /* Thread must always be allowed to run -
even under heavy load */

        /* Data updated during assert_wait/thread_wakeup */
        decl_simple_lock_data(,sched_lock)      /* scheduling lock (thread_lock()) */
        decl_simple_lock_data(,wake_lock)       /* for thread stop / wait (wake_lock())
*/
        boolean_t               wake_active;    /* wake event on stop */
        int                     at_safe_point;  /* thread_abort_safely allowed */
        ast_t                   reason;                 /* why we blocked */
        wait_result_t           wait_result;    /* outcome of wait -
                                                 * may be examined by this thread
                                                 * WITHOUT locking */
        thread_continue_t       continuation;   /* continue here next dispatch */
        void                            *parameter;             /* continuation parameter
*/

        /* Data updated/used in thread_invoke */
    struct funnel_lock  *funnel_lock;           /* Non-reentrancy funnel */
    int                  funnel_state;
#define TH_FN_OWNED             0x1                             /* we own the funnel */
#define TH_FN_REFUNNEL          0x2                             /* re-acquire funnel on
dispatch */
```

```
        vm_offset_t             kernel_stack;       /* current kernel stack */
        vm_offset_t             reserved_stack;     /* reserved kernel stack */

        /* Thread state: */
        int                     state;
/*
 *      Thread states [bits or'ed]
 */
#define TH_WAIT                 0x01                /* queued for waiting */
#define TH_SUSP                 0x02                /* stopped or requested to stop */
#define TH_RUN                  0x04                /* running or on runq */
#define TH_UNINT                0x08                /* waiting uninteruptibly
#define TH_TERMINATE    0x10                        /* halted at termination */
#define TH_TERMINATE2   0x20                        /* added to termination queue */

#define TH_IDLE                 0x80                /* idling processor */

        /* Scheduling information */
        sched_mode_t            sched_mode;         /* scheduling mode */
        sched_mode_t            saved_mode;         /* saved mode during forced mode
demotion */
  // Bitmask of miscellaneous TH_SFLAG bits
        unsigned int            sched_flags;        /* current flag bits */
        integer_t               sched_pri;          /* scheduled (current) priority */
        integer_t               priority;           /* base priority */
        integer_t               max_priority;       /* max base priority */
        integer_t               task_priority;      /* copy of task base priority */
        integer_t               promotions;         /* level of promotion */
        integer_t               pending_promoter_index;
        void                    *pending_promoter[2];
        integer_t               importance;         /* task-relative importance */

                                                    /* real-time parameters */
        struct {                                    /* see mach/thread_policy.h */
        uint32_t                period;
        uint32_t                computation;
        uint32_t                constraint;
        boolean_t               preemptible;

        uint64_t                deadline;
        }                       realtime;

        uint32_t                was_promoted_on_wakeup;
        uint32_t                current_quantum;    /* duration of current quantum */
        uint64_t  last_run_time;                    /* time when thread was switched away
from */
        uint64_t  last_quantum_refill_time;         /* time current_quantum refilled after
expiration */

  /* Data used during setrun/dispatch */
        timer_data_t   system_timer;                /* system mode timer */
        processor_t    bound_processor;             /* bound to a processor? */
        processor_t    last_processor;              /* processor last dispatched on */
```

continues

LISTING 11-1 *(continued)*

```
        processor_t    chosen_processor;         /* Where we want to run this thread */

        /* Fail-safe computation since last unblock or qualifying yield */
        uint64_t       computation_metered;
        uint64_t       computation_epoch;
        uint64_t       safe_release;             /* when to release fail-safe */

        /* Call out from scheduler */
        void           (*sched_call)( int                    type,
                                      thread_t        thread);
#if defined(CONFIG_SCHED_PROTO)
        uint32_t       runqueue_generation;      /* last time runqueue was drained */
#endif

        /* Statistics and timesharing calculations */
#if defined(CONFIG_SCHED_TRADITIONAL)
        natural_t    sched_stamp;                /* last scheduler tick */
        natural_t    sched_usage;                /* timesharing cpu usage [sched] */
        natural_t    pri_shift;                  /* usage -> priority from pset */
        natural_t    cpu_usage;                  /* instrumented cpu usage [%cpu] */
        natural_t    cpu_delta;                  /* accumulated cpu_usage delta */
#endif
        uint32_t     c_switch;                   /* total context switches */
        uint32_t     p_switch;                   /* total processor switches */
        uint32_t     ps_switch;                  /* total pset switches */

        /* Timing data structures */
        timer_data_t user_timer;                 /* user mode timer */
        uint64_t     user_timer_save;            /* saved user timer value */
        uint64_t     system_timer_save;          /* saved system timer value */
        uint64_t     vtimer_user_save;           /* saved values for vtimers */
        uint64_t     vtimer_prof_save;
        uint64_t     vtimer_rlim_save;

        /* Timed wait expiration */
        timer_call_data_t      wait_timer;
        integer_t              wait_timer_active;
        boolean_t              wait_timer_is_set;

        /* Priority depression expiration */
        timer_call_data_t      depress_timer;
        integer_t              depress_timer_active;
        /* Processor/cache affinity
         * - affinity_threads links task threads with the same affinity set
         */
        affinity_set_t         affinity_set;
        queue_chain_t          affinity_threads;

        /* Various bits of stashed state */
        union {
           struct {
             mach_msg_return_t      state;      /* receive state */
             ipc_object_t           object;     /* object received on */
```

```
            mach_vm_address_t     msg_addr;         /* receive buffer pointer */
            mach_msg_size_t       msize;            /* max size for recvd msg */
            mach_msg_option_t     option;           /* options for receive */
            mach_msg_size_t       slist_size;       /* scatter list size */
            mach_port_name_t      receiver_name;    /* the receive port name */
            struct ipc_kmsg       *kmsg;            /* received message */
            mach_port_seqno_t     seqno;            /* seqno of recvd message */
            mach_msg_continue_t   continuation;
              } receive;
          struct {
            struct semaphore      *waitsemaphore;   /* semaphore ref */
            struct semaphore      *signalsemaphore; /* semaphore ref */
            int                   options;          /* semaphore options */
            kern_return_t         result;           /* primary result */
            mach_msg_continue_t continuation;
                 } sema;
          struct {
            int                   option;           /* switch option */
                 } swtch;
          int                     misc;             /* catch-all for other state */
        } saved;
/* IPC data structures */
        struct ipc_kmsg_queue ith_messages;
        mach_port_t ith_rpc_reply;                  /* reply port for kernel RPCs */

        /* Ast/Halt data structures */
        vm_offset_t               recover;          /* page fault recover(copyin/out) */
        uint32_t                  ref_count;        /* number of references to me */

        queue_chain_t             threads;          /* global list of all threads */

        /* Activation */
        queue_chain_t                  task_threads;

        /*** Machine-dependent state ***/
        struct machine_thread   machine;

        /* Task membership */
        struct task                            *task;
        vm_map_t                               map;

        decl_lck_mtx_data(,mutex)

        /* Kernel holds on this thread  */
        int                                         suspend_count;

        /* User level suspensions */
        int                                         user_stop_count;

        /* Pending thread ast(s) */
        ast_t                                  ast;

        /* Miscellaneous bits guarded by mutex */
        uint32_t   active:1,                        /* Thread is active and has not been
        terminated */
```

continues

LISTING 11-1 *(continued)*

```
                  started:1,                  /* Thread has been started after
                                                 creation */
                  static_param:1,             /* Disallow policy parameter changes */
                  :0;

        /* Return Handers */
        struct ReturnHandler {
               struct ReturnHandler    *next;
               void            (*handler)(
                     struct ReturnHandler            *rh,
                     struct thread                           *thread);
                } *handlers, special_handler;

        /* Ports associated with this thread */
        struct ipc_port             *ith_self;   /* not a right, doesn't hold ref */
        struct ipc_port             *ith_sself;  /* a send right */
        struct exception_action exc_actions[EXC_TYPES_COUNT];

        /* Owned ulocks (a lock set element) */
        queue_head_t                    held_ulocks;

#ifdef  MACH_BSD
        // this field links us from the Mach layer to the BSD layer
        void                                    *uthread;
#endif

#if CONFIG_DTRACE
                uint32_t t_dtrace_predcache;/* DTrace per thread predicate value hint */
                int64_t t_dtrace_tracing;       /* Thread time under dtrace_probe() */
                int64_t t_dtrace_vtime;
#endif

  uint32_t    t_page_creation_count;
                clock_sec_t t_page_creation_time;

                uint32_t t_chud;                /* CHUD flags, used for Shark */

                integer_t mutex_count;          /* total count of locks held */

                uint64_t thread_id;             /*system wide unique thread-id*/

        /* Statistics accumulated per-thread and aggregated per-task */
        uint32_t                syscalls_unix;
        uint32_t                syscalls_mach;
        zinfo_usage_store_t     tkm_private;     /* private kernel memory allocs/frees */
        zinfo_usage_store_t     tkm_shared;      /* shared kernel memory allocs/frees */
        struct process_policy ext_actionstate;   /* externally applied actions */
        struct process_policy ext_policystate;   /* externally defined process policy
states*/
        struct process_policy actionstate;       /* self applied acions */
        struct process_policy policystate;       /* process wide policy states */
};
```

The preceding structure is huge, and therefore most threads are created by cloning off of a generic template, which fills the structure with default values. This template is the `thread_template` defined in `osfmk/thread/thread.c`. It is filled by `thread_bootstrap()`, which is called as part of the kernel boot (in `i386_init`), and is copied off of in `thread_create_internal()`, which implements the `thread_create()` Mach API.

One particular field of interest is the `uthread` member, which is a void pointer to the BSD layer. This member points to a BSD user thread, which is opaque to Mach, and remains opaque, as it will in this chapter (although we will explore it in Chapter 13, which unravels the BSD layer).

Notice that while it is full of miscellaneous fields, a thread contains no actual resource references. Mach defines the *task* as a thread container, and it is the task level in which resources are handled. A thread has access (via ports) to only the resources and memory allocated in its containing task.

Tasks

A *task* serves as a container object, under which the virtual memory space and resources are managed. These resources are devices and other handles. The resources are further abstracted by ports. Sharing resources thus becomes a matter of providing access to their corresponding ports.

Strictly speaking, a task is not what other operating systems call a process, as Mach, being a microkernel, provides no process logic, only the bare bones implementation. In the BSD model, however, a straightforward 1:1 mapping exists between the two concepts, and every BSD (and therefore, OS X) process has an underlying Mach task object associated with it. This mapping is accomplished by specifying an opaque pointer, `bsd_info`, to which Mach remains entirely oblivious. Mach represents the kernel by a task as well, (globally referred to as the `kernel_task`) though this task has no corresponding PID (technically, it can be thought of as PID 0).

The task is a relatively lightweight structure (at least, compared to the threads), defined in `osfmk/kern/task.h` as shown in Listing 11-2. The noteworthy fields are emphasized.

LISTING 11-2 The Mach task structure, from osfmk/kern/task.h

```
struct task {
        /* Synchronization/destruction information */
        decl_lck_mtx_data(,lock)        /* Task's lock */
        uint32_t        ref_count;      /* Number of references to me */
        boolean_t       active;         /* Task has not been terminated */
        boolean_t       halting;        /* Task is being halted */

        /* Miscellaneous */
        vm_map_t        map;            /* Address space description */
        queue_chain_t   tasks;          /* global list of tasks */
        void            *user_data;     /* Arbitrary data settable via IPC */

        /* Threads in this task */
        queue_head_t            threads;   // Threads, in FIFO queue

        processor_set_t         pset_hint;
        struct affinity_space   *affinity_space;
```

continues

LISTING 11-2 *(continued)*

```
    int           thread_count;        // #threads in threads queue
    uint32_t      active_thread_count; // #active threads (<=thread_count)
    int                     suspend_count;  /* Internal scheduling only */

    /* User-visible scheduling information */
    integer_t               user_stop_count;   /* outstanding stops */

    task_role_t             role;

    integer_t               priority;      /* base priority for threads */
    integer_t               max_priority;/* maximum priority for threads */

    /* Task security and audit tokens */
    security_token_t sec_token;
    audit_token_t   audit_token;
/* Statistics */
    uint64_t                total_user_time;  /* terminated threads only */
    uint64_t                total_system_time;

    /* Virtual timers */
    uint32_t                vtimers;

    /* IPC structures */
    decl_lck_mtx_data(,itk_lock_data)
    struct ipc_port *itk_self;      /* not a right, doesn't hold ref */
    struct ipc_port *itk_nself;     /* not a right, doesn't hold ref */
    struct ipc_port *itk_sself;     /* a send right */
    struct exception_action exc_actions[EXC_TYPES_COUNT];
                                    /* a send right each valid element  */
    struct ipc_port *itk_host;      /* a send right */
    struct ipc_port *itk_bootstrap; /* a send right */
    struct ipc_port *itk_seatbelt;  /* a send right */
    struct ipc_port *itk_gssd;      /* yet another send right */
    struct ipc_port *itk_task_access; /* and another send right */
    struct ipc_port *itk_registered[TASK_PORT_REGISTER_MAX];
                                    /* all send rights */

    // remember that each task has its own private port namespace.
    // (Namespaces are explained in the section dealing with Mach IPC)
    struct ipc_space *itk_space;    // task local port namespace

    /* Synchronizer ownership information */
    queue_head_t    semaphore_list;       /* list of owned semaphores   */
    queue_head_t    lock_set_list;        /* list of owned lock sets    */
    int             semaphores_owned;     /* number of semaphores owned */
    int             lock_sets_owned;      /* number of lock sets owned  */

    /* Ledgers */  // These are likely different in Mountain Lion and iOS
    struct ipc_port *wired_ledger_port;
    struct ipc_port *paged_ledger_port;
    unsigned int     priv_flags;          /* privilege resource flags */
```

```
        MACHINE_TASK

        // If you've ever wondered where top(1) gets its info - this is it
        // These fields can be queried with task_info flavor 2 (task_events_info)
        integer_t faults;              /* faults counter */
        integer_t pageins;             /* pageins counter */
        integer_t cow_faults;          /* copy on write fault counter */
        integer_t messages_sent;       /* messages sent counter */
        integer_t messages_received;   /* messages received counter */
        integer_t syscalls_mach;       /* mach system call counter */
        integer_t syscalls_unix;       /* unix system call counter */
        uint32_t  c_switch;            /* total context switches */
        uint32_t  p_switch;            /* total processor switches */
        uint32_t  ps_switch;           /* total pset switches */

        zinfo_usage_store_t tkm_private;/* private kmem alloc/free stats */
        zinfo_usage_store_t tkm_shared; /* shared kmem alloc/free stats  */
        zinfo_usage_t tkm_zinfo;        /* per-task, per-zone usage statistics */

#ifdef  MACH_BSD
        void *bsd_info;   // MAPPING TO BSD PROCESS OBJECT
#endif
        struct vm_shared_region         *shared_region;
        uint32_t taskFeatures[2];    // 64-bit addressing/register flags.

        mach_vm_address_t  all_image_info_addr; /* dyld __all_image_info  */
mach_vm_size_t         all_image_info_size; /* section location and size */

#if CONFIG_MACF_MACH
        ipc_labelh_t label;
#endif

#if CONFIG_COUNTERS
#define TASK_PMC_FLAG 0x1       /* Bit in "t_chud" signifying PMC interest */
        uint32_t t_chud;                /* CHUD flags, used for Shark */
#endif

        process_policy_t ext_actionstate;  /* externally applied actions */
        process_policy_t ext_policystate;  /* ext. def. process policy states*/
        process_policy_t actionstate;      /* self applied acions */
        process_policy_t policystate;      /* process wide policy states */

        uint64_t rsu_controldata[TASK_POLICY_RESOURCE_USAGE_COUNT];

        vm_extmod_statistics_data_t     extmod_statistics;
};
```

On its own, a task has no life. Its *raison d'être* is to serve as a container of one or more threads. The threads in a task are maintained in the `threads` member, which is a queue containing `thread_count` threads, as highlighted in the preceding code.

Additionally, most of the operations on a task are really just iterations of the same corresponding thread operations for all threads in the given task. For example, to set the task priority, `task_priority()` is implemented as in Listing 11-3:

LISTING 11-3: The implementation of task_priority(), from osfmk/kern/task_policy.c

```
static void task_priority(
        task_t                  task,
        integer_t               priority,
        integer_t               max_priority)
{
        thread_t                thread;

        task->max_priority = max_priority;

        if (priority > task->max_priority)
                priority = task->max_priority;
        else
        if (priority < MINPRI)
                priority = MINPRI;

        task->priority = priority;

        queue_iterate(&task->threads, thread, thread_t, task_threads) {
                thread_mtx_lock(thread);

                if (thread->active)
                        thread_task_priority(thread, priority, max_priority);

                thread_mtx_unlock(thread);
        }
}
```

The `queue_iterate` macro loops over the `queue_head_t`. Each thread, in turn, is locked. If it is active, its priority can be set. The thread can then be unlocked.

Ledgers

Ledgers provide a mechanism to charge quotas and set limits for Mach tasks. This is somewhat similar to the `getrlimit(2)`/`setrlimit(2)` system calls offered by POSIX, but offers more advanced resource throttling capabilties: Resources (typically CPU and memory) can be transferred in between ledgers, and exceeding their limits can result in a Mach exception, callback execution, or thread block until the ledger is “refilled”.

Ledgers have been around since the inception of Mach, but have only recently been implemented in XNU. In fact, they will only be supported officially as of Mountain Lion, having made their debut in iOS. Though the Lion kernel sources have an `osfmk/kern/ledger.c` file, the comment on the file admits it is nothing more than a “half-hearted attempt” for “dysfunctional” ledgers, providing only the `root_wired_ledger` and `root_paged_ledger` ledgers. Both are initialized (by `ledger_init`) to be unlimited (`LEDGER_ITEM_INFINITY`), so the system keeps track, but does not enforce any limits on its wired and paged memory.

A new BSD System call, #373 (aptly named `ledger`) is currently undocumented, but supported in iOS and will likely be supported in Mountain Lion. The call is a BSD bridge to the underlying Mach

APIs of `ledger_info()`, `ledger_entry_info()`, and `ledger_template_info()` for codes of 0, 1, or 2, respectively. It remains, at the time of writing, undocumented. This will enable ledgers to be used on a per-task basis, allowing for greater control over system resources such as CPU and memory, which are especially scarce and precious on iOS.

Task and Thread APIs

The rich structures of `task_t` and `thread_t` presented so far are in some ways too rich — the structures are huge and contain a plethora of detail that most kernel APIs do not need to access, at least not directly. Another problem is that the structures may change in between kernel versions (and, in fact, are slightly different in the closed source iOS). Fortunately, Mach contains an assortment of API calls that you can use on tasks and threads in an object-oriented manner, leaving the actual implementations opaque. You can and should use specific accessor functions for the important fields, such as `get_bsdthread_info()`, `get_bsdtask_info()`, `get_bsdthreadtask_info()`, and so on. Additionally, you can use APIs corresponding to task and thread "methods," discussed next in this section.

Getting the Current Task and Thread

At any given point, the kernel must be able to get the handle of the current task and current thread. It accomplishes this via two functions: `current_task()` and `current_thread()`, respectively.

Although the functions are defined in `osfmk/kern/task.h` and `osfmk/kern/thread.h`, respectively, they are really wrappers over architecture-dependent variants. Both functions are macros over corresponding "fast" functions. The trick involved in both operations is in getting `current_thread()`, i.e., `current_thread_fast()`, because the `current_task()` can be retrieved by simply returning the task field of the current thread (and, in fact, `current_task_fast()` is defined over the `current_thread() -> task`).

If you look through the XNU sources, you will find that `current_thread()` (in `osfmk/i386/machine_routines.c` and as a macro in `osfmk/i386/cpu_data.h`) wraps `current_thread_fast()`, which in turn is `#defined` over `get_active_thread()`. The implementation of `get_active_thread()` wraps `CPU_DATA_GET(cpu_active_thread,thread_t)`, which is inline assembly (relying on the GS register). In iOS, the assembly call relies on the ARM coprocessor's special register c13. If you're interested in the low level specifics, refer to the appendix in this book.

Task APIs

Mach provides a complete subsystem of functions to handle tasks. The APIs exposed to user mode are in `<mach/task.h>`, which includes an architecture header (i.e., `<mach/i386/task.h>`, or `<mach/arm/task.h>`. The latter can be found in the `iPhoneOS5.0`.sdk directories). Table 11-1 details these functions, which are (with the exception of `mach_task_self()`) all implemented over Mach messages (MIG subsystem 3400):

TABLE 11-1: Task APIs available in user mode

MACH TASK APIS	USED FOR
mach_task_self()	Obtains task's port, with names of send rights.
task_create(task_t target_task, ledger_array_t ledgers, mach_msg_type_number_t, boolean_t, task_t *child_task);	Creates child_task from target_task. Initializes with array of ledgersCnt ledgers. Inherits parent's memory task if set. Otherwise, task starts with no memory, and memory must be set up manually. This call is no longer supported. Its body, task_create_internal, is still visible privately from the kernel to support BSD's fork() and cloneproc().
task_terminate(task_t target_task)	Terminates the existing task.
task_threads(task_t target_task, thread_act_array_t *act_list, mach_msg_type_number_t *alCnt);	Enumerates all threads in target_task into array, act_list, containing alCnt entries of the ports of target task.
task_info(task_name_t, task_flavor_t kern/thread.h, task_info_t, task_info_out, mach_msg_type_number_t *task_info_outCnt) **task_set_info**(task_t, task_flavor_t *flavor*, task_info_t, mach_msg_type_number_t);	Queries information on task_name_t. Information is of type task_flavor_t. See the following experiment for an example of flavors. set_info similarly sets information on task.
task_suspend(task_t target_task); **task_resume**(task_t target_task);	Suspends or resumes *target_task*, done by enumerating all the task threads and calling thread_suspend/resume directly Calling task_suspend increments the suspension count; task_resume decrements it. A task will be runnable if its suspend count is 0. Wrapped by the BSD layer's pid_suspend and pid_resume system calls.
get_special_port (task_t task, int which_port, mach_port_t *special_port)	Get special port for a given task. A corresponding set_special_port is available as well.

MACH TASK APIS	USED FOR
task_set_exception_ports (task_t task, exception_mask_t, mach_port_t, exception_behavior_t, thread_state_flavor_t); **task_get_exception_ports** (task_t, exception_mask_t, exception_mask_array_t, mach_msg_type_number_t *, exception_handler_array_t, exception_behavior_array_t, exception_flavor_array_t);	Queries, sets, or swaps between task-level exception ports, which are where Mach exception messages will be sent.
task_policy_set (task_t, task_policy_flavor_t, task_policy_t, mach_msg_type_number_t); **task_policy_get**(task_t, task_policy_flavor_t, task_policy_t, mach_msg_type_number_t *, boolean_t *);	Set or get scheduling policy for a task (i.e., all its threads).
task_sample (task_t task, mach_port_t reply);	Periodically samples and saves IP (Intel) or PC (ARM) of task. Removed.
task_get_state(task_t task, thread_state_flavor_t, thread_state_t, mach_msg_type_number_t *);	Gets the state of a task. A corresponding `task_set_state()` is also available.

Additionally, internal APIs — unexposed to user mode — include the ones in Table 11-2.

TABLE 11-2: Mach kernel private task APIs

MACH TASK APIS	USED FOR
task_priority (task_t, Integer_t priority, Integer_t max);	Sets priority of `task_t` to be priority, and sets maximum allowed priority to be `max`. This is achieved by iterating all threads and calling `thread_task_priority`.
task_importance(task_t, integer_t importance)	Wrapper over `task_priority()`, used when `renice(2)`ing processes. Effectively calls the former with `importance + BASEPRI_DEFAULT`.

> *The task port is the path to complete and unfettered control over the task, its threads and its resources. The APIs shown in the preceding tables are but a fraction of the operations Mach allows on a task. The next section shows how a task's threads can be manipulated externally, and Chapter 12 will show even more APIs (and a companion tool), which enable breaching and defiling the task's sanctum sanctorum — its virtual memory image.*
>
> *These capabilities become immeasurably more potent when applied to the* `kernel_task`*., allowing a privileged user to peek and modify kernel memory. It is for this reason that Apple goes to great lengths to prevent user mode access to the* `kernel_task` *in iOS, and why jailbreaking patches usually target these protections first.*

Experiment: Using the Task APIs

The preceding chapter showed you the `host_info()` function, and it's only natural to expect similar functions to exist for tasks and threads. The chapter ended with a demonstration of enumerating tasks on the default processor set, but did not really show anything other than the corresponding PIDs.

Using `task_info` it is possible to extend Listing 10-19 to also provide highly detailed information about tasks. The second parameter to `task_info` is the `task_flavor_t`, specifying the type of information requested. The flavors are somewhat volatile, and their changes from version to version can make it hard for third parties to rely on them for diagnostics. But the risk of recompiling (and dealing with insipid, obsoleted constants) is well worth the cornucopia of diagnostic information provided by these APIs. It is through `task_info` that `top(1)` gets all the highly detailed and Mach-specific information it displays if its terminal window size permits.

Listing 11-4 shows how `task_info` can be used to query some of the flavors supported in Lion and later:

LISTING 11-4: Using task_info with various flavors from Lion and iOS

```
doTaskInfo(task_t Task)
{
  // proper code does validation checking on calls.
  // Omitted here for brevity
  mach_msg_type_number_t infoSize;

  char infoBuf[TASK_INFO_MAX];
  struct task_basic_info_64     *tbi;
  struct task_events_info       *tei;

#if LION  // Will also work on iOS 5.x or later
  struct task_kernelmemory_info *tkmi;
  struct task_extmod_info       *texi;
  struct vm_extmod_statistics *ves;
#endif

  kern_return_t kr;
```

```
  infoSize = TASK_INFO_MAX;
  kr = task_info(Task,
                TASK_BASIC_INFO_64,
                (task_info_t)infoBuf,
                &infoSize);
  tbi = (struct task_basic_info_64 *) infoBuf;

  printf ("\tSuspend Count: %d\n", tbi->suspend_count);
  printf ("\tMemory:  %dM virtual, %dK resident\n",
         tbi->virtual_size / (1024 * 1024), tbi->resident_size / 1024);
  printf ("\tSystem/User Time: %ld/%ld\n", tbi->system_time, tbi->user_time);

  infoSize = TASK_INFO_MAX; // need to reset (this is an in/out parameter)
kr = task_info(Task,
                TASK_EVENTS_INFO,
               (task_info_t)infoBuf,
                &infoSize);

  tei = (struct task_events_info *) infoBuf;
  printf("Faults: %d, Page-Ins: %d, COW: %d\n", tei->faults, tei->pageins,
        tei->cow_faults);
  printf ("Messages: %d sent, %d received\n", tei->messages_sent, tei->messages_received);
  printf ("Syscalls: %d Mach, %d UNIX\n", tei->syscalls_mach, tei->syscalls_unix);

#if LION
  infoSize = TASK_INFO_MAX; // need to reset (this is an in/out parameter)
  kr = task_info(Task,
                TASK_KERNELMEMORY_INFO, // defined as of Lion
               (task_info_t)infoBuf,
                &infoSize);

  tkmi = (struct task_kernelmemory_info *) infoBuf;

  printf ("Kernel memory: Private: %dK allocated %dK freed,  Shared: %dK allocated, %dK
freed\n",
        tkmi->total_palloc/ 1024, tkmi->total_pfree  /1024,
        tkmi->total_salloc/ 1024, tkmi->total_sfree  /1024);

// Lion and later offer the VM external modification information - really
// useful to detect all sorts of attacks certain tools (like gdb and corerupt, presented
// in the next chapter) utlize to debug/trace processes

  infoSize = TASK_INFO_MAX; // need to reset (this is an in/out parameter)
  kr = task_info(Task,
                TASK_EXTMOD_INFO, // defined as of Lion
               (task_info_t)infoBuf,
                &infoSize);
if (kr == KERN_SUCCESS) {printf("--OK\n");}
  texi = (struct vm_extmod_statistics *) infoBuf;
  ves = &(texi->extmod_statistics);

  if (ves->task_for_pid_count)
    { printf ("Task has been looked up %ld times\n", ves->task_for_pid_count); }
  if (ves->task_for_pid_caller_count)
    { printf ("Task has looked up others %ld times\n", ves->task_for_pid_caller_count); }
```

continues

LISTING 11-4 *(continued)*

```
    if (ves->thread_creation_count || ves->thread_set_state_count)
     {  printf ("Task has been tampered with\n"); }
    if (ves->thread_creation_caller_count || ves->thread_set_state_caller_count)
     {  printf ("Task has tampered with others\n"); }

#endif
}
```

Plugging this function into Listing 10-19 is straightforward. In a manner similar to this experiment, you can drill down further to the thread level by using the `thread_info()` function. This is but one of many thread APIs, discussed next.

Thread APIs

Much as it does for tasks, Mach provides a rich API for thread management. Most of these achieve the same functionality as the task APIs. Indeed, the task APIs often just iterate over the list of threads in each task, and apply these in turn. As can be expected, these calls (aside from `mach_thread_self`) are implemented over Mach messages (and generated by MIG subsystem 3600). Table 11-3 lists the thread APIs. All return a `kern_return_t`, unless otherwise noted.

TABLE 11-3: Mach Thread APIs

MACH THREAD API	USED FOR
`thread_t mach_thread_self``()`	Sends rights to thread's kernel port.
`thread_terminate``(thread_t thread)`	Terminates self.
`[thread/act]_[get/set]_state` `(thread_t thread,` `int flavor,` `thread_state_t state,` `mach_msg_type_number_t *count)`	Gets/sets thread context. The act functions disallow getting/setting the current thread, but otherwise fall through to the thread functions. The `thread_state_t is` platform dependent. In OS X, it is an `x86_thread_state_t` (either 32- or 64-bit). In iOS, it is an `arm_thread_state_t`.
`thread_suspend``(thread_t thread)` **`thread_resume`** `(thread_t thread)`	Suspends or resumes *thread* by incrementing/decrementing the suspend count. The thread may only execute if both its `suspend` count and its containing task `suspend` count is zero.
`thread_abort[_safely]` `(thread_t thread)`	Destroys another thread.
`thread_depress_abort` `(thread_t thread)`	Cancel thread depression (forced lowering of priority).

MACH THREAD API	USED FOR
thread_[get/set]_special_port (thread_act_t thread, int which_port, thread special_port);	Gets or sets one of several special ports for the thread. The only special port supported in XNU is `THREAD_KERNEL_PORT`.
thread_info(thread_t *thread*, thread_flavor_t flavor, thread_info_t tinfo_out, mach_msg_type_number_t *ti_count)	Queries information on `thread` according to `flavor`, and returns it in buffer specified by `tinfo_out`, which is `ti_count` bytes long. GDB uses this call when you use the `info task` or `info thread` command.
thread_get_exception_ports **thread_set_exception_ports** **thread_swap_exception_ports**	Queries, sets, or swaps between exception ports, which are where Mach exception messages will be sent. Discussed later under Exceptions.
thread_policy/thread_set_policy	Obsolete; has been replaced by `thread_policy_get/set`.
thread_policy_[get/set] (thread_t thread, thread_policy_flavor_t flavor, thread_policy_t policy_info, mach_msg_type_number_t *count, boolean_t *get_default))	Threads scheduling policy. `thread_policy_set` is defined similarly (no `get_default_argument`, and `count` is an in `parameter`).
thread_sample	Deprecated and removed. On CMU Mach, this allows the periodic sampling of a thread's program counter (IP/PC) and receiving of the samples using a `receive_samples` API.
etap_trace_thread	Deprecated and removed in Leopard and later. Similar to `thread_sample()`, above, this once enabled tracing a thread using ETAP buffers.
thread_assign(thread_t thread, processor_set_t new_pset)) **thread_assign_default** (thread_t thread)	Assigns (=`affine`) *thread* to a particular processor set *`new_pset`*, or the default one. Unsupported (returns `KERN_FAILURE`).
thread_get_assigment (thread_t thread, processor_set_t *pset)	Returns current thread assignment to processor set (CPU affinity). Always returns a reference to pset0, the default processor set.

As an exercise, you might want to extend the listing in the previous experiment to also list threads. This can be done by calling `task_threads()` on the task port, and `thread_info` (with `THREAD_BASIC_INFO`) on each of the thread ports returned.

In-Kernel Thread APIs

Mach provides a set of thread control functions, which are accessible in kernel mode only. These are declared in `osfmk/kern/sched_prim.h`:, and a subset of them is shown in Table 11-4:

TABLE 11-4 Some of the kernel-internal thread control functions in osfmk/kern/sched_prim.h

MACH THREAD API	USED FOR
`wait_result_t` **`assert_wait`** `(event_t` *`event`*`,` `wait_interrupt_t` *`interruptible`*`)`	Adds the current thread to the wait queue on *`event`*. The event is converted to a wait queue by a `wait_hash()` function.
`wait_result_t` **`assert_wait_deadline`**`(` `event_t` *`event`*`,` `wait_interrupt_t` *`interruptible`*`,` `uint64_t` *`deadline`*`)`	As *`assert_wait()`*, but allows specification of a future deadline.
`kern_return_t` **`thread_wakeup_prim`** `(event_t` *`event`*`,` `boolean_t` *`one_thread`*`,` `wait_result_t` *`result`*`);`	Wakes up a thread (*`one_thread`* `= TRUE`) or threads waiting on specified *`event`*. This function wraps around `thread_wakeup_prim_internal`, which in turn calls `wait_queue_wakeup_[one|all]`. This function is usually wrapped by one of these macros: `thread_wakeup(x)` `thread_wakeup_with_result(x,z)` `thread_wakeup_one(x)`
`wait_result_t` **`thread_block_reason`**`(` `thread_continue_t` *`continuation`*`,` `void` *`*parameter`*`,` `ast_t` *`reason`*`);`	Blocks the current thread, yielding CPU execution, and optionally setting a *`continuation`* routine and a *`parameter`* for it. May specify AST in *`reason`*. This function is usually wrapped by one of lightweight: `thread_block(thread_continue_t`, specifying a NULL parameter, and `AST_NONE` for `reason` `thread_block_parameter (thread_continue_t, void *)`, specifying `AST_NONE` for `reason`.
`thread_bind` `(processor_t` *`processor`*`);`	Sets the CPU affinity of this thread to *processor* or removes affinity `(PROCESSOR_NULL);`.
`int` **`thread_run`** `(thread_t self,` `thread_continue_t` *`continuation`*`,` `void` *`*parameter`*`,` `thread_t` *`new_thread`*`)`	Performs thread handoff; the current thread yields CPU execution (parameters are the same as `thread_block_parameter`), but transfers control directly to *`new_thread`*. Used in handoffs (described later in this chapter). This function wraps around `thread_invoke()`, which is internal to the scheduler.

MACH THREAD API	USED FOR
`kern_return_t thread_go (thread_t thread, wait_result_t wresult);`	Unblock a thread and dispatch it. Used when removing a thread from a wait queue.
`void thread_setrun (thread_t thread, integer_t options)`	Dispatch a thread, to its bound (affined processor) or any (preferably idle) processor. Calls `realtime_setrun` for `realtime` threads, `fairshare_setrun` for fairshare_setrun, or `processor_setrun`.

Thread Creation

Of particular interest is the thread creation API. Since a thread cannot exist outside of some containing task, this API is defined in `task.h` (more specifically, `<mach/ARCH/task.h>`, and implemented in `osfmk/kern/thread.c`. (See Table 11-5.)

TABLE 11-5: Thread creation functions

MACH THREAD API	USED FOR
`thread_create (task_t parent, thread_act_t *child_act)`	Create a thread in the *parent* task, and return it in *child_act*.
`thread_create_running (task_t parent, thread_state_flavor_t flavor, thread_state_t new_state, mach_msg_type_number_t nsCnt, thread_act_t *child_act);`	Create a thread in the *parent* task, and initialize its state to *new_state*. The *thread_state_t* is dependent on machine architecture (and changes between i386, x86_64, and ARM)

Notice the first argument: `task_t` is the task in which the thread will be created. This means that, from Mach's perspective, *a thread can be created in any task the user has the corresponding port for*. This makes the Mach infrastructure extremely flexible in enabling the creation of remote threads.[1]

Thus, when one uses `pthread_create()`, an underlying API call to Mach's `thread_create` ensues, with `mach_task_self()` as the first argument (followed by pthread house keeping, and `bsdthread_create` for the corresponding BSD thread, as will be discussed in Chapter 13). But if you have another task's port, you can inject threads into it. In the right (or wrong?) hands, uncanny functionality can be achieved, as injected threads obtain full access to the virtual memory of their task, and are extremely hard to detect.

[1]Windows also has a powerful thread creation API — using the `CreateRemoteThread()` along with `WriteProcessMemoryEx()`, which enables the user to write to the memory of any process whose handle can be obtained. Mainstream UNIX and Linux, however, do not have this ability, and threads may only be created locally.

> *Creating a thread is simple, but having it do something meaningful is a tad more complicated. For starters, you would usually need to "bring your own code," using the* `mach_vm_write` *API (presented in the next chapter) to inject code into the foreign task. Then, you would need to use* `thread_set_state` *(shown in Table 11-3) to initialize the thread's register state to load and run the supplied code. All of these, however, are mere minutiae, as these APIs will all work once you have the task port at hand.*

SCHEDULING

No matter how many CPUs (or cores) a system has, threads will surely outnumber them. The kernel, therefore, has to be able to "juggle" threads on CPUs, allowing as many threads to execute in what the human user would perceive as concurrency. In actuality, however, because each core can only execute one thread at a time, the kernel has to be able to perform context switches between threads by preempting one thread and replacing it with another.

> *Multiprocessing is now commonplace, and various technologies — hyperthreading, multiple cores, and multiple processors — can be used at the hardware level to enable this functionality. Although each technology has its plusses and minuses, from the kernel's perspective, no real difference exists among the aforementioned technologies. Whether you use hyperthreading, two cores, or two distinct CPUs, most operating systems see two* logical *processors.*

With the `processor-set` abstraction, Mach is somewhat better suited than Linux or Windows and can actually manage cores of the same CPU in the same `pset` and separate CPUs in separate `psets`. The rest of this section makes no distinction between the cases, and uses the term *CPU* for a logical, rather than a physical CPU.

The High-Level View

Recall that context switching is the task of freezing a given thread by recording its register state into a predefined memory location. The register state is machine-specific (because each machine type has a different set of registers). After a thread is preempted, the CPU registers can be loaded with the saved thread of another thread, thereby resuming its execution.

Irrespective of operating system, the basic idea of thread scheduling is the same: A thread executes in the CPU (or core, or hyperthread) for as long as it needs. *Executing* refers to the fact that the CPU registers are filled with the thread state, and — as a consequence — the code the CPU is executing (by EIP/RIP or PC) is the code of the thread function in question. This execution goes on until one of the following occurs:

- The thread terminates. Most threads eventually reach an endpoint. Either the thread function returns, or the thread calls `pthread_exit()`, which will call `thread_terminate`.

- The thread voluntarily gives up the CPU. Even though the thread work is not done, because of waiting for a resource or other blocking operation, continuing at this point in time makes no sense. The thread therefore willingly requests the scheduler to context switch to some other thread. The thread also needs to inform the system on when it would like to return to the CPU, either by specifying some deadline (in clock ticks) or requesting notification of some event.
- An external interrupt interferes with thread execution, directing the CPU to save the thread register state and immediately execute the interrupt-handling code. Since the thread is interrupted anyway, before returning from the interrupt-handling code the system invokes the scheduler to figure out whether a non-voluntary context switch (i.e., preemption) is in order. Such a non-voluntary context switch is the result of the thread's timeslice (quantum) expiring, or some other, higher priority thread waking up.

Priorities

All threads are equal, but some threads are more equal than others. In other words, threads are assigned specific priorities, which directly affect the frequency with which they are scheduled. Every operating system provides a range of such priorities: Windows has 32, Linux has 140, and Mach has 128.

The scheduler's `osfmk/kern/sched.h` file illustrates the usage of priority ranges (which Apple calls "priority bands") with ASCII graphics. Figure 11-1 presents it with more modern graphics:

Setting the kernel threads' minimum priority to 80, high above that of user mode, ensures that kernel and system-housekeeping will preempt user mode threads, except for very specific cases as shown in the next experiment.

Experiment: Viewing Priorities using ps -l

Using `ps(1)`'s OS X specific `-l` switch will display both the priority and nice values of every (`-e`) running processes. First, try this on OS X, and optionally use `tr(1)` and `cut(1)`, as shown in Output 11-1 to isolate the priority, nice value, and command names. Note that in OS X the depressed processes are reniced:

OUTPUT 11-1 Using ps –l to show process priorities and nice values in OS X

```
morpheus@Minion(~)$ ps -le | tr -s ' '  | cut -d' ' -f7,8,16 | sort -n
PRI NI CMD
4   17 …/Frameworks/Metadata.framework/Versions/A/Support/mdworker
4   17 …/CoreServices.framework/Frameworks/Metadata.framework/Versions/A/Support/mdworker
4   20 /usr/sbin/netbiosd
23  10 /usr/libexec/warmd
23  10 /usr/libexec/warmd_agent
31  0  -bash
…
54  0 /System/Library/CoreServices/loginwindow.app/Contents/MacOS/loginwindow
…
63  0 /sbin/dynamic_pager
63  0 /usr/libexec/hidd
97  0 /Applications/iTunes.app/Contents/MacOS/iTunes ; iTunes is real time
(TIME_CONSTRAINT)
97  0 /usr/sbin/coreaudiod                          ; along with the audiod
```

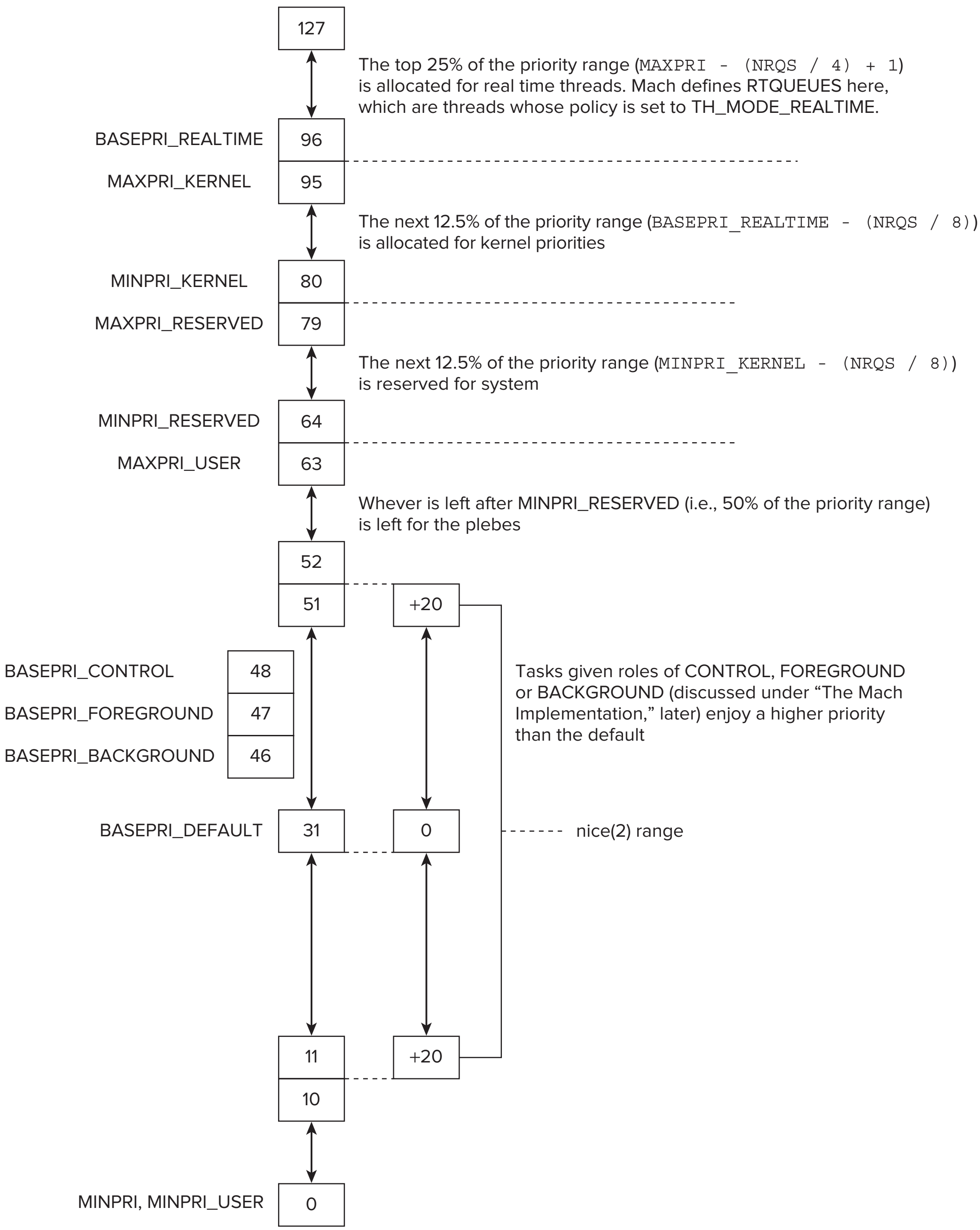


FIGURE 11-1: The Mach priority ranges

Next, if you try the same command on iOS, you will reveal some interesting patterns: The backgrounded apps are all depressed with a priority of 4, the currently active app has a priority of 47, SpringBoard is at 63, and `configd` is actually real time. These priorities are all policy enforced, however, as the nice values for all these processes are 0. (See Output 11-2.)

OUTPUT 11-2: Using ps –l to show process priorities and nice values in iOS

```
root@Padishah (~)# ps -le | tr -s ' '  | cut -d' ' -f7,8,16 | sort -n
PRI NI CMD
4   0  /Applications/AppStore.app/AppStore                              ;
4   0  /Applications/MobileNotes.app/MobileNotes                        ; Background
4   0  /Applications/MobileSafari.app/MobileSafari                      ;
4   0  /Applications/Preferences.app/Preferences                        ;
Applications
4   0  /var/mobile/Applications/0CCB04C5-8D03-4D07-8A0F-E4112F5B6534/WSJ.app/WSJ
..
31  0 -sh
31  0 /sbin/launchd
31  0 /usr/sbin/fairplayd.K95
31  0 /usr/sbin/syslogd
..
47  0 /Applications/MobileMusicPlayer.app/MobileMusicPlayer
47  0 /System/Library/PrivateFrameworks/IAP.framework/Support/iapd
47 /System/Library/PrivateFrameworks/MediaRemote.framework/Support/mediaremoted
47 /usr/libexec/locationd
47 /var/mobile/Applications/70565622-4490-4174-9531-EEB7B7C5715D/Remote.app/Remote ;
foreground
47 /usr/libexec/locationd
61 /usr/sbin/mediaserverd
63 /System/Library/CoreServices/SpringBoard.app/SpringBoard   ; Always at MAXPRI_USER
97 /usr/libexec/configd                                        ; Real time
```

Priority Shifts

Assigning thread priorities is a start, but often those priorities need to be adjusted during runtime. Mach dynamically tweaks the priorities of each thread, to accommodate for the thread's CPU usage, and overall system load. Threads can thus "drift" in their priority bands, decreasing in priority when using the CPU too much, and increasing in priority if not getting enough CPU. The traditional scheduler uses a macro (`do_priority_computation`) and a function (`update_priority`), both in `osfmk/kern/priority.c`, to update dynamically the priority of each thread. The macro toggles the thread priority by subtracting its calculated sched_usage (calculated by the function, accounting for CPU usage delta), shifted by a `pri_shift` value. The `pri_shift` value is derived from the global `sched_pri_shift`, which is updated by the scheduler regularly as part of the system load calculation in `compute_averages (osfmk/kern/sched_average.c)`. Subtracting the CPU usage delta effectively penalizes those threads with high CPU usage (positive usage delta detracts from priority) and rewards those of low CPU usage (negative usage delta adds to priority).

To make sure the thread's CPU usage doesn't accrue to the point where the penalty is lethal, the `update_priority` function gradually ages CPU usage. It makes use of a `sched_decay_shifts` structure, to simulate the exponential decay of the CPU usage by a factor of $(\frac{5}{8})n$, defined in the same file as shown in Listing 11-5. By using the pre-computed shift values, the computation can be sped up, expressed in terms of bit shifts and additions, which take less time than multiplication:

LISTING 11-5 The sched_decay_shifts structure in osfmk/kern/priority.c

```
/*
 *      Define shifts for simulating (5/8) ** n
 *
 *      Shift structures for holding update shifts.  Actual computation
 *      is  usage = (usage >> shift1) +/- (usage >> abs(shift2))  where the
 *      +/- is determined by the sign of shift 2.
 */
struct shift_data {
        int     shift1;
        int     shift2;
};

// The shift data at index i provides the approximation of (5/8)i
#define SCHED_DECAY_TICKS       32
static struct shift_data        sched_decay_shifts[SCHED_DECAY_TICKS] = {
        {1,1},{1,3},{1,-3},{2,-7},{3,5},{3,-5},{4,-8},{5,7},
        {5,-7},{6,-10},{7,10},{7,-9},{8,-11},{9,12},{9,-11},{10,-13},
        {11,14},{11,-13},{12,-15},{13,17},{13,-15},{14,-17},{15,19},{16,18},
        {16,-19},{17,22},{18,20},{18,-20},{19,26},{20,22},{20,-22},{21,-27}
};
```

Mach also supports "throttling" and defines `MAXPRI_THROTTLE(4)` for priority throttled processes, i.e., those processes that are intentionally penalized by the system. In iOS (`CONFIG_EMBEDDED`) the throttled priority is used for the `DEPRESSPRI` constant for apps in the background and affects the calculation of the `do_priority_computation` macro. The Mach host APIs provide the `HOST_PRIORITY_INFO` flavor to the `host_info()` function (discussed in Chapter 10), which returns a `host_priority_info` structure, reporting the various priority levels.

All the threads, with their various and volatile priorities must somehow be managed in an efficient way, to allow the scheduler to find the next runnable thread of the highest priority in the minimum amount of time possible. This is where run queues enter the picture.

Run Queues

Threads are placed into *run queues*, which are priority lists defined in `osfmk/kern/sched.h` as shown in Listing 11-6:

LISTING 11-6 The run queue, from osfmk/kern/sched.h

```
struct runq_stats {
        uint64_t                                  count_sum;
        uint64_t                                  last_change_timestamp;
};

#if defined(CONFIG_SCHED_TRADITIONAL) || defined(CONFIG_SCHED_PROTO) ||
defined(CONFIG_SCHED_FIXEDPRIORITY)
```

```
struct run_queue {
        int             highq;                  /* highest runnable queue */
        int             bitmap[NRQBM];         /* run queue bitmap array */
        int             count;                 /* # of threads total */
        int             urgency;               /* level of preemption urgency */
        queue_head_t queues[NRQS];             /* one for each priority */

        struct runq_stats       runq_stats;
};

#endif /* defined(CONFIG_SCHED_TRADITIONAL) || defined(CONFIG_SCHED_PROTO) ||
defined(CONFIG_SCHED_FIXEDPRIORITY) */
```

The run queue is a multi-level list, or an array of lists, one queue for each of the 128 priorities (`#defined` as NRQS). To make for a quick lookup of the next priority to execute, Mach uses a technique (which was used in Linux 2.6, prior to 2.6.23) called O(1) scheduling. That is, rather than looking at the array, checking each entry until a non-NULL one is found — which is also technically O(1), but really is O(128) scheduling — Mach checks a bitmap, which enables it to look at 32 (`#defined` as NRQBM)[2] simultaneously. This makes the lookup O(4), which is about as fast as possible, and most important, considering that the scheduling logic runs frequently and in critical time.

> *Notice that the very definition of the run queue becomes conditional on using one of several schedulers. Mach uses a "traditional" or default scheduler, but the scheduler is modular, and may be modified or replaced altogether with other schedulers. (See the later section, "Scheduling Algorithms," for more on this topic).*

Code cannot just modify the thread's `sched_pri` field directly, as assigning a new priority for a thread also means moving it from one queue to another. This is performed by `set_sched_pri` (`osfmk/kern/sched_prim.c`), which is called from `compute_priority` (`osfmk/kern/priority.c`). This is shown in Figure 11-2.

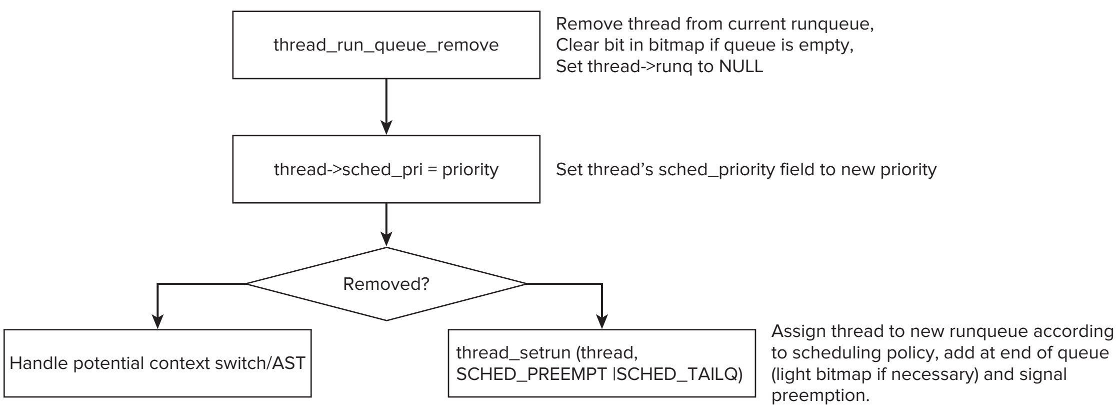


FIGURE 11-2: Setting thread priority and moving the threads between queues

[2]NRQBM is hard `#defined` in `osfmk/kern/sched.h` to be NRQS/32, even for the 64-bit architecture. A `sizeof()` would have been more adequate.

Wait Queues

A thread is optimally either in the running state or the ready state, waiting for the processor. There are times when the thread is blocking, waiting for some IPC object (such as a mutex or semaphore), some I/O operation (for example, a file or socket), or event. In those cases, there is no benefit in considering scheduling the thread, since execution can only be resumed once the object or operation is at hand, or the event has occurred.

In those cases, a thread may be placed into a wait queue. A wait `queue_t` is defined as an opaque point in `osfmk/kern/kern_types.h`, with the implementation in `osfmk/kern/wait_queue.c`, as shown in Listing 11-7:

LISTING 11-7: The wait queue implementation, from osfmk/kern/wait_queue.c

```
/*
 *      wait_queue_t
 *      This is the definition of the common event wait queue
 *      that the scheduler APIs understand.  It is used
 *      internally by the gerneralized event waiting mechanism
 *      (assert_wait), and also for items that maintain their
 *      own wait queues (such as ports and semaphores).
 *
 *      It is not published to other kernel components.  They
 *      can create wait queues by calling wait_queue_alloc.
 *
 *      NOTE:  Hardware locks are used to protect event wait

 *      queues since interrupt code is free to post events to
 *      them.
 */
typedef struct wait_queue {
    unsigned int                    /* flags */
    /* boolean_t */     wq_type:16,     /* only public field */
                        wq_fifo:1,      /* fifo wakeup policy? */
                        wq_prepost:1,   /* waitq supports prepost? set only */
                        :0;             /* force to long boundary */
    hw_lock_data_t      wq_interlock;   /* interlock */
    queue_head_t        wq_queue;               /* queue of elements */
} WaitQueue;
```

The wait queue handling functions are exported for use by kernel components in `osfmk/kern/wait_queue.h`. To add a thread to a wait queue, any of the `wait_queue_assert_wait[64[_locked]]` variants may be used. The functions all enqueue the thread at the tail of the queue (unless the thread is realtime, privileged, or on a FIFO wait queue, in which case it is enqueued at the head of the queue). The functions are further wrapped by `assert_wait` (in `osfmk/kern/sched_prim.c`) and other wrappers, used throughout the kernel, and especially in the BSD layer.

When the wait condition is satisfied, the waiting thread(s) can be unblocked and dispatched again. The `wait_queue_wakeup64_[all|one]_locked` (to wake up one or all threads when an event occurs) are used for this purpose. The functions dequeue the thread(s) from the wait queue, and dispatch them using `thread_go`, which unblocks (using `thread_unblock`) and dispatches the threads (using `thread_setrun`).

CPU Affinity

In modern architectures using multi-core, SMP, or hyperthreading, it is also possible to affine a particular thread with one or more specific CPUs. This can be useful to both the thread and the system as a whole because the thread can benefit from its data being "left behind" in the CPU caches when it returns to execute on the same CPU.

In Mach parlance, a thread's affinity to a CPU is defined as a *binding*. `thread_bind(osfmk/kern/sched_prim.c)` is used for this purpose, and merely updates the `thread_t`'s `bound_processor` field. If the field is set to anything but `PROCESSOR_NULL`, future scheduling decisions involving the thread (e.g., `thread_setrun`) will only dispatch the thread to that processor's run queue.

MACH SCHEDULER SPECIFICS

The view of scheduling presented so far is actually common to all modern operating systems. Mach, however, adds several noteworthy features:

- Handoffs allow a thread to voluntarily yield the CPU, but not to just any other thread. Rather, it hands the CPU off to a particular thread (of its choice). This feature is especially useful in Mach, given that it is a message-passing kernel, and messages pass between threads. This way, the messages can be processed with minimal latency, rather than opportunistically waiting for the next time the message-processing thread, sender or receiver, gets scheduled.
- Continuations are used in cases where the thread does not care much for its own stack, and can discard it, enabling the system to resume it without restoring the stack. This key feature, specific to Mach, and used in many places around the kernel.
- Asynchronous Software Traps (ASTs) are software complements to the low-level hardware traps mechanisms. Using ASTs the kernel can respond to out-of-band events requiring attention such as scheduling events.
- Scheduling algorithms are modular, and the scheduler can be dynamically set on boot (using the `sched boot-arg`). In practice, however, only one scheduler (the so-called *traditional* scheduler) is used.

Handoffs

All operating system support the notion of *yielding*, which is the act of voluntarily giving up the CPU to some other thread. The classic form of yielding does not enable the yielding thread to choose its successor, and the choice is left up to the scheduler.

Mach improves on this by adding the option to *handoff* the CPU. This enables the yielding thread to supply a hint to the scheduler as to what is the next best thread to execute. This doesn't fully obligate the scheduler, which may choose to transfer control to some other thread (if the thread specified is, for example, not runnable). The scheduler does, however, ignore thread policies and so handoffs usually succeed. As a result of a handoff, the current thread's remaining quantum is given to the new thread to be scheduled.

To handoff, rather than yield, a thread calls `thread_switch()`, specifying the port of the thread to switch to, optional flags (such as depressing the replacing thread's priority), and the time these options will be in effect. What's even more interesting is that the thread handoff mechanism is accessible from user mode: Mach exports the `thread_switch()` as a trap (#61), so it can be called from user mode. This is actually one of the few Mach traps that has a manual page (`osfmk/man/thread_switch.html`).

Continuations

Although context switching is straightforward in most operating systems, following a classic model wherein each thread has its own task, Mach offers an alternative by introducing the concept of a *continuation*. A continuation is an optional resumption function (along with a parameter to it), which a thread may specify if it is voluntarily requesting a context switch. If a continuation is specified, when the thread is resumed it will be reloaded from the point of continuation with a new stack and no previous state saved. This makes context switching much faster, since the saving and loading of registers can be omitted (In addition, this saves a significant amount of space on the kernel stack, which is fairly small, only four pages, or 16 K). Threads in a continuation require only 4–5 KB for the thread state, saving an additional 16 K that would be otherwise needed. Instead of a full register state and thread stack, only the continuation and an optional parameter need to be saved, and this can be done on the thread structure itself. A simple test for continuation may be performed and, if one is found, it is simply jumped to, with its parameter passed to it. A thread specifies its desire to be blocked using `thread_block()`, optionally specifying a continuation (or using `THREAD_CONTINUE_NULL`, if the standard mode is preferred). A parameter to the continuation may be specified by `thread_block_parameter()`. Both calls are wrappers over `thread_block_reason()`, which is described in the section "Explicit Preemption," later in this chapter.

Continuations are a quick and efficient mechanism to alleviate the cost of context switching, and they are used primarily in Mach's kernel threads. In fact, Mach's `kernel_thread_create` (and its main caller, `kernel_thread_start_priority`) is built over the idea of a continuation, as shown in Listing 11-8.

LISTING 11-8 kernel_thread_create and its use of continuations

```
kern_return_t
kernel_thread_create(
        thread_continue_t          continuation,
        void                      *parameter,
        integer_t                  priority,
        thread_t                  *new_thread)
{
        kern_return_t           result;
        thread_t                thread;
        task_t                  task = kernel_task;

// thread_create_internal sets the thread.continuation
        result = thread_create_internal
                 (task, priority, continuation, TH_OPTION_NONE, &thread);
        if (result != KERN_SUCCESS)
                return (result);

        task_unlock(task);
```

```
        lck_mtx_unlock(&tasks_threads_lock);

        stack_alloc(thread);
        assert(thread->kernel_stack != 0);
#if CONFIG_EMBEDDED
        if (priority > BASEPRI_KERNEL) // Set kernel stack for high priority threads
#endif
        thread->reserved_stack = thread->kernel_stack;

       // and the parameter is set manually here
        thread->parameter = parameter;

       if(debug_task & 1)
        kprintf("kernel_thread_create: thread = %p continuation = %p\n",
                 thread, continuation);

        *new_thread = thread;

        return (result);
}

..
kern_return_t kernel_thread_start_priority(
        thread_continue_t       continuation,
        void                   *parameter,
        integer_t               priority,
        thread_t               *new_thread)
{
        kern_return_t   result;
        thread_t                thread;

        result = kernel_thread_create(continuation, parameter, priority, &thread);
        if (result != KERN_SUCCESS)
                return (result);

        *new_thread = thread;

        thread_mtx_lock(thread);
        thread_start_internal(thread);
        thread_mtx_unlock(thread);

        return (result);
}
```

Continuations are particularly attractive in kernel threads, since it is a simple matter to set the continuation is simply the thread entry point. Hence, this is the way Mach kernel threads are started. User mode thread creation also makes use of continuations, by setting (in `thread_create_internal2`) the continuation to `thread_bootstrap_return()`. This is just a DTrace hook, followed by `thread_exception_return()`, which returns to user mode.

Note that continuations require the setting thread to be aware of both the preemption and the continuation logic. It follows, therefore, that Mach supports two different models of preemption — explicit and implicit — with the continuation model only available for explicit preemptions. These are discussed next.

Continuations are the brainchild of Richard Draves, one of the original developers of Mach (whose name still adorns the XNU sources in osfmk/ipc and elsewhere). Continuations were introduced in 1991[3], in a paper by Draves, Bershad, and Rashid, part of a Ph.D. thesis at CMU[4]).

Preemption Modes

Threads in a system may be preempted in one of two ways: explicitly, when a thread gives up control of the CPU or enters an operation defined as blocking, and implicitly, due to an interrupt. Explicit preemption is sometimes referred to as *synchronous*, as it is *a priori* predictable. Interrupts, which by their very nature are unpredictable, make implicit preemption *asynchronous*.

Explicit Preemption

Explicit preemption occurs when a thread voluntarily wants to relinquish the CPU. This could be due to waiting for a resource, or I/O, or sleeping for a set amount of time. User mode threads are subject to explicit preemption when calling blocking system calls, such as `read()`, `select()`, `sleep`, and so on.

To provide explicit preemption, Mach offers the `thread_block_reason()` function. This function, defined in `osfmk/kern/sched_prim.c`, takes three parameters: A continuation function, a parameter for it, and a reason. The reason is an `AST_` (Asynchronous Software Trap) constant, discussed later.

`thread_block_reason` is defined as shown in Listing 11-9.

LISTING 11-9: thread_block_reason() in osfmk/kern/sched_prim.c

```
/*
 *      thread_block_reason:
 *
 *      Forces a reschedule, blocking the caller if a wait
 *      has been asserted.
 *
 *      If a continuation is specified, then thread_invoke will
 *      attempt to discard the thread's kernel stack.  When the
 *      thread resumes, it will execute the continuation function
 *      on a new kernel stack.
 */

thread_block_reason(
        thread_continue_t       continuation,
 void                            *parameter,
        ast_t                           reason){
        register thread_t               self = current_thread();
        register processor_t    processor;
        register thread_t               new_thread;
        spl_t                                   s;

        counter(++c_thread_block_calls);
```

```
        s = splsched();

        if (!(reason & AST_PREEMPT))
                funnel_release_check(self, 2);

        processor = current_processor();

        /* If we're explicitly yielding, force a subsequent quantum */
        if (reason & AST_YIELD)
                processor->timeslice = 0;

        /* We're handling all scheduling AST's */
        ast_off(AST_SCHEDULING);


       // Save continuation and its relevant parameter, if any, on our own uthread

        self->continuation = continuation;
        self->parameter = parameter;
       // improbable kernel debug stuff omitted here
     do {
            thread_lock(self);
            new_thread = thread_select(self, processor);
            thread_unlock(self);
        } while (!thread_invoke(self, new_thread, reason)); // thread_invoke will switch
context

        funnel_refunnel_check(self, 5);
        splx(s);

        return (self->wait_result);
}
```

Two helper functions are also defined: `thread_block_parameter()` and `thread_block()`. The former calls `thread_block_reason()` with the `reason` parameter set to `AST_NONE`, and the latter does the same, but also sets the parameter to NULL.

Calling `thread_block` allows the setting of a continuation, which is stored on the `thread_t` structure (`current_thread()->continuation`) along with its parameter (`current_thread()->parameter`). The `thread_block()` function then calls `thread_select()` to get the next thread on the current processor (which may or may not be different from the current), and tries to call `thread_invoke()` on it.

The `thread_invoke()` function is responsible for performing the context switch and handling the continuation. This function is quite long (and could benefit from an overhaul!), but basically checks whether the new thread to be invoked has a continuation function. If it does, the continuation function is directly called. Otherwise, performing a full context switch becomes necessary.

From a higher-level perspective, the operation is actually quite simple, as shown in Figure 11-3.

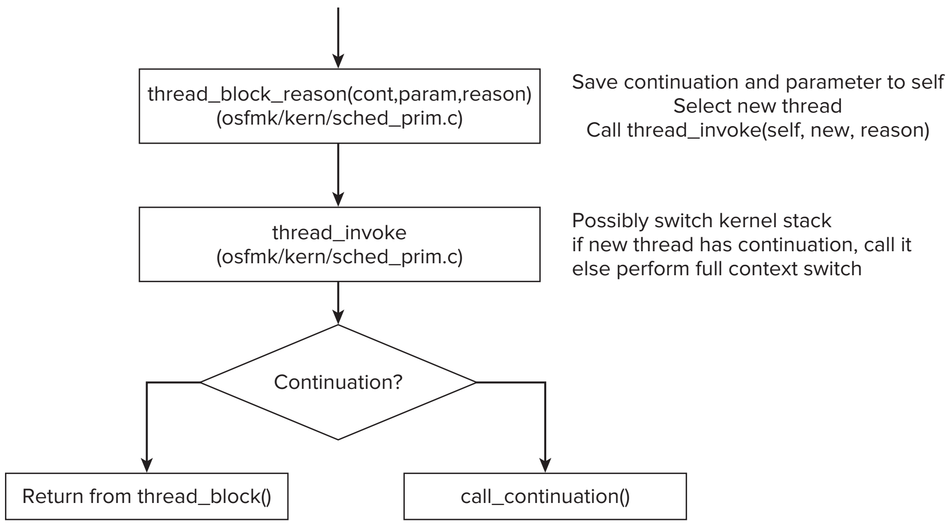


FIGURE 11-3: Thread Invocation

`call_continuation()` is a machine-dependent, much faster mechanism to restore state. Listing 11-10 shows how on x86_64 this can be implemented with efficient code:

LISTING 11-10: the call_continuation implementation on x86_64

```
//prototype: call_continuation(thread_continue_t       continuation,
//                             void                   *parameter,
//                             wait_result_t           wresult);

Entry(call_continuation)
        movq    %rdi,%rcx                     /* get continuation */
        movq    %rsi,%rdi                     /* continuation param */
        movq    %rdx,%rsi                     /* wait result */
        movq    %gs:CPU_KERNEL_STACK,%rsp     /* set the stack */
        xorq    %rbp,%rbp                     /* zero frame pointer */
        call    *%rcx                         /* call continuation */
       // usually not reached - if reached, thread will terminate:
        movq    %gs:CPU_ACTIVE_THREAD,%rdi
        call    EXT(thread_terminate)
```

Implicit Preemption

Mac OS 9 was built entirely around the concept of explicit preemption, which made it a *cooperative multitasking* system. But explicit preemption is inherently limited, as leaving the choice of relinquishing the CPU to the running thread is extremely unreliable. Threads can be caught in time-consuming processing, or worse, endless loops, and never get to a point of explicit preemption.

Mac OS X, by contrast, is a *preemptive multitasking* system. In plain terms, Mach reserves the right to preempt a thread at any given time, whether or not the thread is ready for it. Unlike explicit preemption, this implicit form of preemption is invisible to the thread. The thread remains blissfully unaware, and its state is saved and restored transparently. Most threads won't care about this, as they are likely to be I/O bound anyway. But for CPU-intensive threads, this could be problematic, especially when time-critical performance may be required (for example, video and audio decoding).

Implicit preemption is far simpler, conceptually, from its explicit counterpart. This is because it does not involve any continuations. Since the thread is unaware of its being suspended, it cannot ask for a continuation.

While a thread cannot explicitly control its own scheduling, Mach does offer several pre-set policies that can work toward guaranteeing classes of service. Note "work toward" because Mach is a time-sharing system, not a real-time one, and there can be no true guarantee of service. Using `thread_policy_set()`, which is a Mach trap visible from user mode, it is possible to request such a policy. The function is defined in `osfmk/kern/thread_policy.c` as follows:

```
kern_return_t
thread_policy_set(
        thread_t                                    thread,
        thread_policy_flavor_t  flavor,
        thread_policy_t                    policy_info,
        mach_msg_type_number_t  count);
```

The function verifies its arguments (that is, that `thread` is not `THREAD_NULL` and that `thread->static_param` is `false`), and then calls `thread_policy_set_internal()`, which `switch()`es on the flavor argument, which may be one of the following items in Table 11-6.

TABLE 11-6: Flavor arguments

TASK POLICIES	SPECIFIES
`STANDARD_POLICY`	Fair queuing. Approximately equal share to all computations. No data be provided to the policy. This is deprecated, effectively equivalent to `EXTENDED_POLICY`, below, with timesharing.
`EXTENDED_POLICY`	Fair queuing, but provides a forward hint for long-running computation. An optional parameter, `timeshare`, may be specified.
`TIME_CONSTRAINT_POLICY`	Policy defined by period, computation, constraint, and preemptible — soft real time. This boosts the thread's priority to the real-time range (discussed later).
`PRECEDENCE_POLICY`	Policy defined by thread's importance field, which enables preferring it with respect to other threads in the same task.
`AFFINITY_POLICY`	Thread scheduled by `affinity_tag`, which prefers scheduling by L2 cache affinity.
`BACKGROUND_POLICY`	Policy defined by priority. This is used only if `CONFIG_EMBEDDED` (iOS), suggesting low priority for background tasks (i.e., those not visible as i-Device's primary).

These flavors allow fine-grained control of the scheduling of individual threads. The default policy, `THREAD_STANDARD_POLICY`, is used for fair time sharing. It requires no additional parameters. `THREAD_EXTENDED_POLICY` builds on it, and adds one Boolean parameter, timeshare, which when false, specifies an alternate policy, and when true, falls back to the standard policy.

A more complicated, and closer to real-time policy is `THREAD_TIME_CONSTRAINT_POLICY`, which allows fine-grained tuning of scheduling. Key to this policy is the notion of "processing arrivals," which is the scheduling of the thread in question. Units are measured in the kernel's CPU clock cycles. This policy is based on several parameters:

- **Period**: Requests a time between two consecutive processing arrivals. If this value is not zero, the thread in question is assumed to seek processor time once every `period` cycle.
- **Computation**: A 32-bit integer specifying the computation time needed each time the thread is scheduled.
- **Constraint**: The maximum amount of (real) time between the beginning and the end of the computation.
- **Preemptible**: A Boolean value specifying whether the computation may be interrupted; that is, whether these computation cycles have to be contiguous (`preemptible` = false) or not (`preemptible` = true)

`THREAD_PRECEDENCE_POLICY` takes one parameter, `importance,` which provides the relative importance of this thread compared to other threads of the same task. The value is signed, meaning threads can bump up or down relative to their peers, yet in XNU the minimum priority is `IDLE_PRI`, which is defined as zero.

`THREAD_AFFINITY_POLICY` provides for L2 cache affinity between threads of the same cache. This means that these threads are likely to run on the same CPU, regardless of cores (as all cores share the same L2 cache, anyway), but not likely to cross CPUs in a true SMP environment. To provide this affinity, this policy uses an `affinity_tag` that is shared among related processes (that is, parent and descendants).

`THREAD_BACKGROUND_POLICY` is used for background threads; that is, threads that are of lesser priority and importance to the system. This is not defined in OS X, but is used in iOS, suggesting its use for Apps which are sent to the background by SpringBoard.

Tasks lend an extra level of scheduling, by providing a "role" field, which may be one of the following shown in Table 11-7.

TABLE 11-7: Task roles

TASK ROLES (TASK_CONSTANT)	SPECIFIES
`RENICED`	Any task altered using `nice(1)` or `renice(1)`.
`UNSPECIFIED`	Default value, unless otherwise specified.
`FOREGROUND_APPLICATION`	GUI foreground.

TASK ROLES (TASK_CONSTANT)	SPECIFIES
BACKGROUND_APPLICATION	GUI background.
CONTROL_APPLICATION	Task is a GUI Control application (usually the dock). Only one task can hold this at any given time. The priority range is set to BASEPRI_CONTROL, up to the task's already maximum priority.
GRAPHICS_SERVER	Reserved for Window Manager use. Only one task at a time can hold this role, and it is usually the WindowServer. The priority range is MAXPRI_RESERVED - 3, MAXPRI_RESERVED.
THROTTLE_APPLICATION	Set to the maximum priority (MAXPRI_THROTTLE). Mapped from PRIO_DARWIN_BG.
NONUI_APPLICATION	Mapped from PRIO_DARWIN_NONUI. Priority range is BASEPRI_DEFAULT, MAXPRI_USER.
DEFAULT_APPLICATION	Default, unless otherwise stated.

The task "role" thus affects the scheduling of its threads.

To allow implicit preemption, some mechanism must exist to support asynchronous events and interruptions at the kernel level. This mechanism is Mach's Asynchronous Software Traps (ASTs), and is described next.

Asynchronous Software Traps (ASTs)

The discussion of trap handling in Chapter 8 explained what happens when a transition is made back from kernel mode into user mode, but has intentionally omitted a key component — *Asynchronous Software Traps* (ASTs). An AST is an artificial, non-hardware trap condition that has been raised. ASTs are crucial for kernel operations and serve as the substrate on top of which scheduling events (such as preemption, discussed earlier in this chapter), and BSD's signals (discussed in Chapter 13) are implemented.

An AST is implemented as a field of various bits in the thread's control block, which can be individually set by a call to `thread_ast_set()`. This is a macro, as shown in Listing 11-11:

LISTING 11-11 Setting ASTs in osfmk/kern/ast.h

```
#define thread_ast_set(act, reason) (hw_atomic_or_noret(&(act)->ast, (reason)))
#define thread_ast_clear(act, reason) (hw_atomic_and_noret(&(act)->ast, ~(reason)))
#define thread_ast_clear_all(act) (hw_atomic_and_noret(&(act)->ast, AST_NONE))
```

The "reasons" defined in Mach are in `osfmk/kern/ast.h`, but are really quite poorly documented. Table 11-8 shows the defined ASTs, and their purpose.

TABLE 11-8: Defined ASTs

AST CONSTANT	MEANING
AST_PREEMPT	Current thread is being preempted.
AST_QUANTUM	Current thread's quantum (time slice) has expired.
AST_URGENT	AST must be handled immediately. Used when inserting real time threads.
AST_HANDOFF	Current thread is handing off the CPU to a specific other thread. This is set by `thread_run()` (`osfmk/kern/sched_prim.c`).
AST_YIELD	Current thread has voluntarily yielded the CPU.
AST_APC	Migration.
AST_BSD	Special AST used during BSD initialization to start the init task; that is, `launchd(1)`.
AST_CHUD[_URGENT]	Computer Hardware Understanding ASTs for profiling and tracing. See discussion of CHUD in Chapter 5.

ASTs can also be used in combos, which are bitwise ORs of the preceding flags. These are shown in Table 11-9.

TABLE 11-9: AST Combinations

AST COMBO	BITWISE OR OF	MEANING
AST_NONE	0	Used to clear all AST reasons.
AST_PREEMPTION	(AST_PREEMPT | AST_QUANTUM | AST_URGENT)	Bitmask of all ASTs that involve preempting the current thread. The `ast_taken()` function will cause the thread to block, and force a context switch.
AST_SCHEDULING	AST_PREEMPTION | AST_YIELD | AST_HANDOFF)	Bitmask of all ASTs that can be set by the scheduler.
AST_PER_THREAD	AST_APC | AST_BSD | MACHINE_AST…	Bitmask of ASTs that are used on a per -hread basis. `MACHINE_AST_PER_THREAD` is unused in OS X (set to 0).
AST_CHUD_ALL	AST_CHUD_URGENT | AST_CHUD	All CHUD ASTs.
AST_ALL	0xFFFFFFFF	Used to set all AST reasons. Set by `i386_astintr()`.

The combos are used to group the ASTs into two classes: those that involve preemption, and those that may be set or unset by the scheduler.

When the system returns from a trap (after the call to `user_trap_returns`) or interrupt (after the call to `INTERRUPT`), it doesn't immediately return to user mode. Instead, the code checks for the presence of an AST by looking at the thread's `ast` field. If it is not 0, it calls `i386_astintr()` to process it, as shown in Listing 11-12.

LISTING 11-12: AST checks on return from trap in osfmk/s86_64/idt64.s

```
Entry(return_from_trap)
        movq    %gs:CPU_ACTIVE_THREAD,%rsp
        movq    TH_PCB_ISS(%rsp), %rsp  /* switch back to PCB stack */
        movl    %gs:CPU_PENDING_AST,%eax
        testl   %eax,%eax
        je      EXT(return_to_user)     /* branch if no AST */
  // otherwise we fall through to here:
L_return_from_trap_with_ast:
               …
               …
2:
        STI                      /* interrupts always enabled on return to user mode */

        xor     %edi, %edi              /* zero %rdi */
        xorq    %rbp, %rbp              /* clear framepointer */
        CCALL(i386_astintr)             /* take the AST */

        CLI
        xorl    %ecx, %ecx              /* don't check if we're in the PFZ */
        jmp     EXT(return_from_trap)   /* and check again (rare) */
```

Figure 11-4 shows the AST check points on return from traps and interrupts as shown in Listing 11-12.

ASTs are thus a little bit like Linux's softIRQs in that they run with all interrupts enabled, but still "out of process time."

`i386_astintr()` is a wrapper over `ast_taken()`, as shown in Listing 11-13:

LISTING 11-13: The implementation of i386_astintr

```
i386_astintr(int preemption)
{
        ast_t           mask = AST_ALL;
        spl_t           s;

        if (preemption)
                mask = AST_PREEMPTION;

        s = splsched();

        ast_taken(mask, s);

        splx(s);
}
```

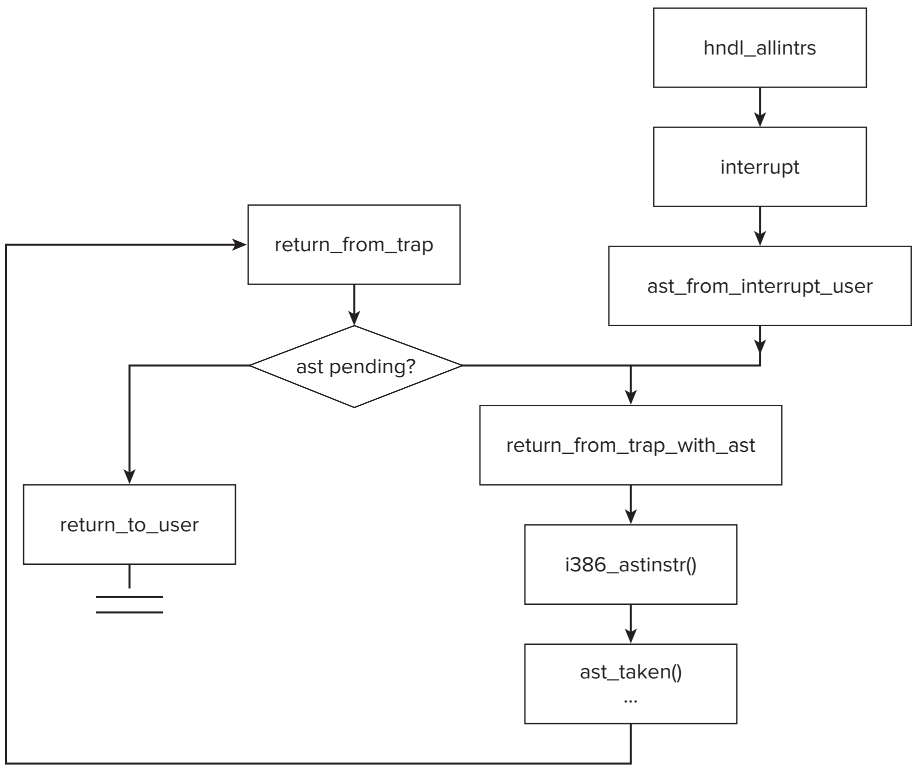


FIGURE 11-4: AST check points on trap and interrupt return

The `ast_taken` function, (which can also be called from kernel traps, and upon kernel thread termination), is responsible for handling the ASTs in all threads save kernel idle threads. ASTs marked as `AST_URGENT` and `AST_PREEMPT` (that is, the `AST_PREEMPTION` combo) cause immediate preemption of the thread. Otherwise, this function checks for `AST_BSD`, which is a temporary hack that was put into Mach for BSD events (such as signals), but remained indefinitely. If a BSD AST is set, `bsd_ast` (from `bsd/kern/kern_sig.c`), is called to handle signals. Chapter 9 covers signals in greater detail.

In IOS, the common code that returns from `fleh_irq`, `undef`, and `prefabt` does something similar, but calls `ast_taken` directly. The `ast_taken` function is also called on `enable_preemption()`.

A special case with ASTs is when function execute in a special region of the commpage (discussed in Chapter 4) known as the Preemption Free Zone (PFZ). Outstanding ASTs are deferred (or *pended*) while in this zone. If you look back at Figure 8-6, you will see in `return_from_trap_with_ast` a call to `commpage_is_in_pfz[32|64]` (both defined for OS X in `osfmk/i386/commpage/commpage.c`). If the address is determined to be in the PFZ, the ASTs are marked pending until the PFZ is exited. Neither PFZ nor commpage are well documented, but what little is provided is shown in Listing 11-14.

LISTING 11-14: The PFZ definition, from osfmk/i386/commpage/commpage.c

```
/* PREEMPTION FREE ZONE (PFZ)
 *
 * A portion of the commpage is speacial-cased by the kernel to be "preemption free",
 * ie as if we had disabled interrupts in user mode.  This facilitates writing
 * "nearly-lockless" code, for example code that must be serialized by a spinlock but
 * which we do not want to preempt while the spinlock is held.
 *
 * The PFZ is implemented by collecting all the "preemption-free" code into a single
 * contiguous region of the commpage.  Register %ebx is used as a flag register;
 * before entering the PFZ, %ebx is cleared.  If some event occurs that would normally
 * result in a premption while in the PFZ, the kernel sets %ebx nonzero instead of
 * preempting.  Then, when the routine leaves the PFZ we check %ebx and
 * if nonzero execute a special "pfz_exit" syscall to take the delayed preemption.
 *
 * PFZ code must bound the amount of time spent in the PFZ, in order to control
 * latency.  Backward branches are dangerous and must not be used in a way that
 * could inadvertently create a long-running loop.
 *
 * Because we need to avoid being preempted between changing the mutex stateword
 * and entering the kernel to relinquish, some low-level pthread mutex manipulations
 * are located in the PFZ.
 */
```

Scheduling Algorithms

Mach's thread scheduling is highly extensible, and actually allows changing the algorithms used for thread scheduling. Table 11-10 shows what you will see if you look at `osfmk/kern/sched_prim.h`.

TABLE 11-10: Supported schedulers in Mach

KSCHED... CONSTANT (STRING)	USED FOR
`TraditionalString ("traditional")`	Traditional (default)
`TraditionalWithPsetRunQueueString ("traditional_with_pset_runqueue")`	Traditional, with PSet affinity
`ProtoString ("proto")`	Global runqueue based scheduler
`GRRRString  ("grrr")`	Group Ratio Round Robin
`FixedPriorityString ("fixedpriority")`	Fixed Priority
`FixedPriorityWithPsetRunqueueString ("fixedpriority_with_pset_runqueue")`	Fixed Priority with PSet affinity

Normally, only one scheduler, the traditional one, is enabled, but the Mach architecture allows for additional schedulers to be defined and selected during compilation using corresponding

CONFIG_SCHED_ directives. The scheduler that will be used can then be specified with the scheduler boot-arg, or a device tree entry.

Each scheduler object maintains a sched_dispatch_table structure, wherein the various operations (think: methods) are held as function pointers. A global table, sched_current_dispatch, holds the currently active scheduling algorithm and allows scheduler switching during runtime. All schedulers must implement the same fields, which the generic scheduler logic invokes using a SCHED macro, as shown in Listing 11-15:

LISTING 11-15: sched_prim.h generic scheduler mechanism

```
/*
 * Scheduler algorithm indirection. If only one algorithm is
 * enabled at compile-time, a direction function call is used.
 * If more than one is enabled, calls are dispatched through
 * a function pointer table.
 */

#if   !defined(CONFIG_SCHED_TRADITIONAL) && !defined(CONFIG_SCHED_PROTO) &&
!defined(CONFIG_SCHED_GRRR
) && !defined(CONFIG_SCHED_FIXEDPRIORITY)
#error Enable at least one scheduler algorithm in osfmk/conf/MASTER.XXX
#endif

#define SCHED(f) (sched_current_dispatch->f)
struct sched_dispatch_table {
           .. // shown in table below //
..
extern const struct sched_dispatch_table *sched_current_dispatch;
```

The scheduler dispatch table itself is described in Table 11-11:

TABLE 11-11: Scheduler dispatch table methods

SCHEDULER METHOD	USED FOR
init()	Initializing the scheduler. Any specific scheduler data structures and bookkeeping is set up here. Called by sched_init().
timebase_init()	Time base initialization.
processor_init(processor_t)	Any per-processor scheduler init code.
pset_init(processor_set_t)	Any per-processor-set scheduler init code.
maintenance_continuation()	The periodic function providing a scheduler tick. This function normally computes the various averages (such as the system load factors), and updates threads on run queues. This function usually re-registers itself.

SCHEDULER METHOD	USED FOR
choose_thread(processor_t, int);	Choosing next thread of greater (or equal) priority int.
steal_thread(processor_set_t)	"Stealing" thread from another processor in pset (used if no runnable threads remain on a processor).
compute_priority(thread_t, boolean_t)	Computing priority of given thread. Boolean is override_depress.
choose_processor(processor_set_t *pset*, processor_t *processor*, thread_t *thread*);	Choosing a processor for thread_t, starting the search at the *pset* specified. May provide a *processor* "hint" if a processor is recommended.
processor_enqueue(processor_t processor, thread_t thread, integer_t options)	Enqueueing thread_t on processor_t by calling run_queue_enqueue on the processor's run queue. Returns TRUE if a preemption is in order. Only option is SCHED_TAILQ - enqueue last.
processor_queue_shutdown(processor_t)	Removing all non-affined/bound threads from processor's run queue.
processor_queue_remove(processor_t, thread_t)	Removing the thread thread_t from the processor queue of the processor_t.
processor_queue_empty(processor_t)	A simple Boolean check for entries in run queue.
priority_is_urgent(int *priority*)	Returns TRUE if the priority is urgent and would mandate preemption.
processor_csw_check(processor_t)	Returns an ast type specifying whether a context switch from (i.e., preemption of) the running thread is required.
processor_queue_has_priority (processor_t, int, boolean_t)	Determining if queue of processor_t has thread(s) with priority greater (boolean_t = false) or greater-equal (true) than priority int.
initial_quantum_size(thread_t)	Returns the initial quantum size of a given thread?
initial_thread_sched_mode(task_t)	Returns a sched_mode_t denoting the scheduling mode for a new thread created in task_t.
supports_timeshare(void)	Returns true if scheduler implementation supports quantum decay.

continues

TABLE 11-11 *(continued)*

SCHEDULER METHOD	USED FOR
can_update_priority(thread_t)	Determines ifhread's priority can be safely updated?
update_priority(thread_t)	Used to update thread thread_t's priority.
lightweight_update_priority(thread_t)	A lighter alternative to update_priority, requiring less processing.
quantum_expire(thread_t)	Denotes quantum expiration for thread_t.
should_current_thread_rechoose_processor (processor_t)	Check whether this processor is preferable for this thread (e.g., because of affinity) or is a better processor available
int **processor_runq_count**(processor_t)	Returning queue load of processor_t. Useful for load balancing.
uint64_t **processor_runq_stats_count_sum**(processor_t)	Aggregating statistics on processor_t's run queue.
fairshare_init()	Any initialization required for fair share threads.
int **fairshare_runq_count**()	Returning number of fair share threads.
uint64_t **fairshare_runq_stats_count_sum** (processor_t)	Aggregating statistics on processor_t's fair-share run queue.
fairshare_enqueue(thread_t thread**)**	Enqueueing fair share thread_t.
thread_t **fairshare_dequeue**()	Dequeueing and returning a fair share thread.
boolean_t **direct_dispatch_to_idle_processors**;	If TRUE, can directly send a thread to an idle processor without needing to enqueue.

To keep the thread scheduling going, every schedule implements a `maintenance_continuation` function. This is just an application of the continuation mechanism described earlier in this chapter for kernel threads. In it, the scheduler thread registers a clock notification using `clock_deadline_for_periodic_event`. A call to `assert_wait_deadline` ensures the thread will run within the specified deadline, and the thread is blocked on the continuation. The process is jumpstarted in the scheduler's `init` function.

The schedulers make heavy use of the Asynchronous Software Trap (AST) mechanism, which was discussed in this chapter. Specifically, the scheduler uses traps of a very specific type: `AST_PREEMPTION`. These tie the scheduling logic to interrupt handling and kernel/user space transitions. It's also worth noting that the scheduling logic is laced with calls to the `kdebug` mechanism (discussed in Chapter 5). The `kdebug` codes (defined with `DBG_MACH_SCHED` and declared in `bsd/sys/kdebug.h`) mark most of the important points in the scheduler's flow.

TIMER INTERRUPTS

This chapter has so far dealt with the primitives and constructs Mach uses in its scheduling logic. In this section, these ideas are integrated with the "engine" which drives scheduling, namely the timer interrupts.

Interrupt-Driven Scheduling

For a system to offer preemptive multitasking, it must support some mechanism to first enable the scheduler to take control of the CPU, thereby preempting the thread currently executing, and then perform the scheduling algorithm, which will decide whether the current thread may resume execution or should instead be "kicked out" to relinquish the CPU to a more important thread.

To usurp control of the CPU from the existing thread, contemporary operating systems (Apple's included) harness the already-existing mechanism of *hardware interrupts*. Because the very nature of interrupts forces the CPU to "drop everything" on interrupt and `longjmp` to the interrupt handler (also known as the interrupt service routine, or ISR), it makes sense to rely on the interrupt mechanism to run the scheduler on interrupt.

One small problem remains, however: Interrupts are *asynchronous*, which means that they can occur at any time and are quite unpredictable. While a busy system processes thousands of interrupts every second, a system with a quiet period of I/O — wherein the usual interrupt sources (the disk, network, and user) are all idle — can also be idle interrupt-wise. There is, therefore, a need for a predictable interrupt source, one that can be relied on to trigger an interrupt within a given time frame.

Fortunately, such an interrupt source exists, and XNU calls it the real time clock, or `rtclock`. This clock is hardware dependent — the Intel architecture uses the local CPU's APIC for this purpose — and can be configured by the kernel to generate an interrupt after a given number of cycles. The interrupt source is often referred to as the *Timer Interrupt*. Older versions of XNU triggered the Timer Interrupt a fixed number of times per second, a value referred to as `hz`. This value is globally defined in the BSD portion of the kernel, in `bsd/kern/clock.c`, (shown in Listing 11-16) and is unappreciated, to say the least:

LISTING 11-16: The now deprecated Hz hardware interval, in bsd/kern/kern_clock.c

```
/*
 * The hz hardware interval timer.
 */

int             hz = 100;               /* GET RID OF THIS !!! */
int             tick = (1000000 / 100);  /* GET RID OF THIS !!! */
```

There is, indeed, good reason to be contemptuous of this. A timer interrupting the kernel at a fixed interval will cause predictable, but extraneous interrupts. Too high a value of `hz` implies too many unnecessary interrupts. On the other hand, too low a value would mean the system is less responsive, as `sub-hz` delays would only be achievable by a tight loop. The old `hertz_tick()` function used in previous versions of OS X is still present, but unused and conditionally compiled only if XNU is compiled with profiling.

The solution is to adopt a different model of a *tick-less* kernel. This model is much like the one from Linux (versions 2.6.21 and above), in which on every Timer Interrupt the timer is *reset* to schedule the next interrupt only when the scheduler deems it necessary. This means that, on every Timer Interrupt, the interrupt handler has to make a (very quick) pass over the list of pending deadlines, which are primarily sleep timeouts set by threads, act on them, if necessary, and schedule the next Timer Interrupt accordingly. More processing in each Timer Interrupt is well worth the savings in spurious interrupts, and the processing can be kept to a minimum by keeping track of only the most exigent deadline.

Timer Interrupt Processing in XNU

XNU defines, per CPU, an `rtclock_timer_t` type (in `osfmk/i386/cpu_data.h`), which is used to keep track of timer-based events. This structure notes the deadline of a timer and a queue of `call_entry` structures (from `osfmk/kern/call_entry.h`), holding the callouts defined as shown in Listing 11-17:

LISTING 11-17: The rtclock_timer_t, from osfmk/i386/cpu_data.h

```
typedef struct rtclock_timer {
        mpqueue_head_t          queue;       // A queue of timer call_entry structures
        uint64_t                deadline;    // when this timer is set to expire
        uint64_t                when_set;    // when this timer was set
        boolean_t               has_expired; // has the deadline passed already?
} rtclock_timer_t;

typedef struct cpu_data
{
  …
        int                     cpu_running;
        rtclock_timer_t         rtclock_timer; // Per CPU timer
        boolean_t               cpu_is64bit;
 …
}
```

The `rtclock_timer`'s queue is kept sorted in order of ascending deadlines, and the `deadline` field is set to the nearest deadline (i.e., the head entry in the queue).

XNU uses another machine-independent concept of an *event timer* (also called the *etimer*) to wrap the `rtclock_timer` and hide the actual machine-level timer interrupt implementation. Its usage is discussed next.

Scheduling Deadlines

Deadline timers are set (read: added to the `rtclock`'s queue) through a call `timer_queue_assign(osfmk/i386/etimer.c)`. This function sets a deadline only if it is earlier (read: expires sooner) than the one already set in the current CPU's `rtclock_timer.deadline`. The actual setting of the deadline at the hardware level is handled by `etimer_set_deadline`, followed by a call to `etimer_resync_deadlines (osfmk/i386/etimer.c)`, which sets the CPU's local APIC, and will be discussed soon.

The scheduler interfaces with `timer_queue_assign` through the higher-level abstraction of a *timer callout*, by using `timer_call_enter`, from `osfmk/kern/timer_call.c`, on the thread's `wait_timer`. The callout is a function pointer with pre-set arguments, defined in `osfmk/kern/timer_call_entry.h` as shown in Listing 11-18:

LISTING 11-18: The callout structure, from osfmk/kern/timer_call_entry.h

```
typedef struct call_entry {
    queue_chain_t       q_link;    // next
    queue_head_t        *queue;    // queue head
    call_entry_func_t   func;      // callout to invoke
    call_entry_param_t  param0;    // first parameter to callout function
    call_entry_param_t  param1;    // second parameter to callout
    uint64_t            deadline;  // deadline to invoke function by
} call_entry_data_t;

// Adjust with flags and a soft deadline, this becomes struct timer_call
typedef struct timer_call {
        struct call_entry       call_entry;
        decl_simple_lock_data( ,lock);          /* protects call_entry queue */
        uint64_t                soft_deadline;  // Tests expiration in
timer_queue_expire()
        uint32_t                flags;
        boolean_t               async_dequeue;  /* this field is protected by
                                                   call_entry queue's lock */
} *timer_call_t;
```

Timer events not deemed critical are added with a so-called "slop" value which coalesces them so as to increase the probability that they expire at the same time (and thus reduce overall timer interrupts). The various callers of `timer_call_enter` can declare their calls to be critical by specifying the `TIMER_CALL_CRITICAL` flag.

The process of setting timer deadlines from the scheduler's end is shown in Figure 11-5.

Timer Interrupt Handling

Timer Interrupt handling is performed by `rtclock_intr` (`osfmk/i386/rtclock.c`). The function itself doesn't do much: It merely asserts all interrupts are disabled determines which mode (kernel or user) was interrupted, and saves the existing thread's registers. The real work is accomplished by a call to `etimer_intr` (`osfmk/i386/etimer.c`), which checks whether the timer deadline (`rtclock_timer->deadline`) or the power management deadline (as returned from `pmCPUGetDeadline()`, in `osfmk/i386/pmCPU.c`) expired, and, if they did, acts on them. If the scheduler can be thought of as the producer of the deadline queue, then this function is its consumer.

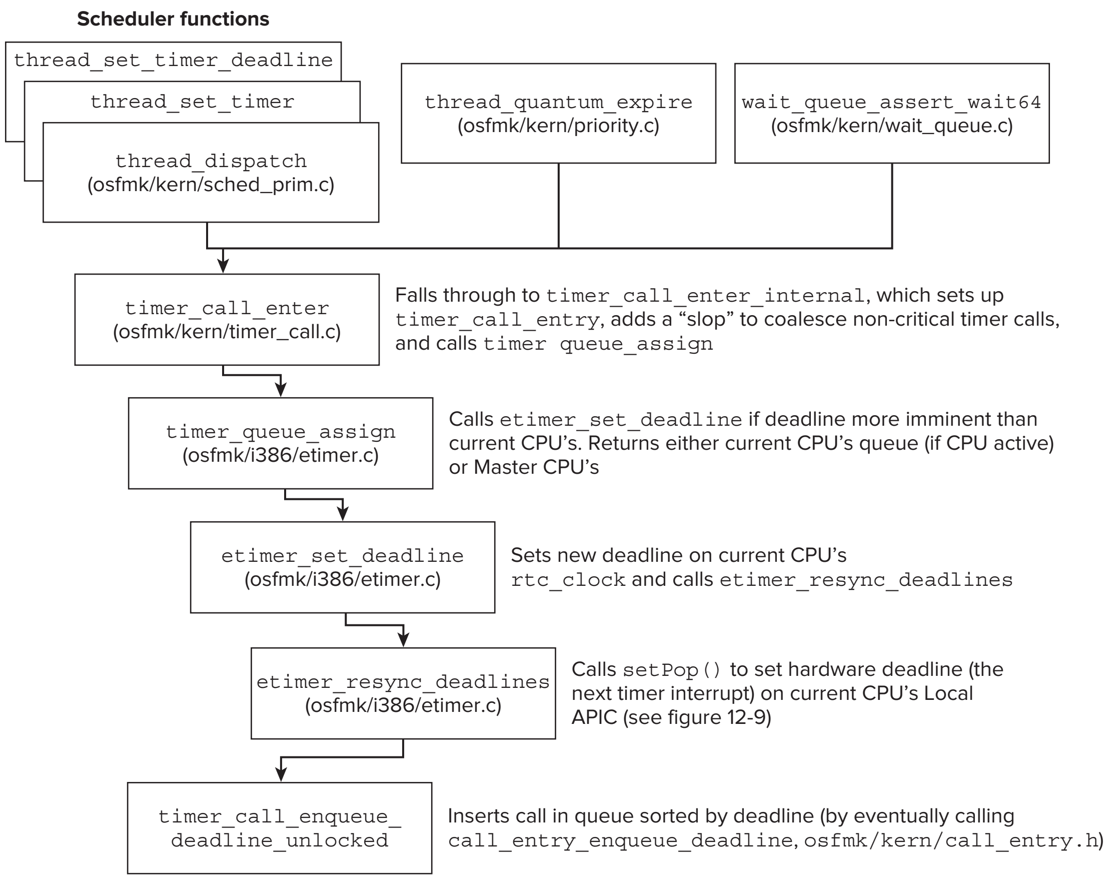


FIGURE 11-5: Setting deadlines

To act on timers `etimer_intr` calls `timer_queue_expire` (or `pmCPUDeadline`, for the power management related deadlines), which walks the queue and invokes the expired timer's callout function, with its two arguments (and also logs a `kdebug` event before and after the call). The function dequeues and invokes callouts until it hits the first callout whose deadline has not yet expired. Because the queue is sorted in order of increasing deadlines, all other deadlines are guaranteed to be pending, as well. The first non-expired deadline effectively becomes the next deadline to process, so it is returned to `etimer_intr`. This is shown in Figure 11-6.

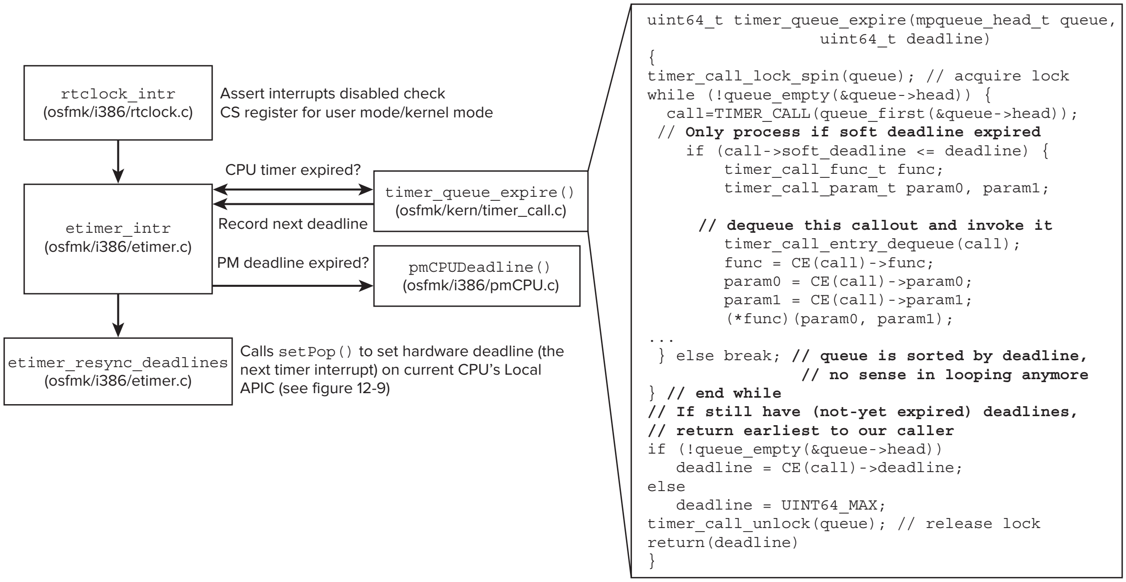


FIGURE 11-6: Timer interrupt processing in XNU

Setting the Hardware Pop

Deadline timers must be communicated to the hardware level, so as to request the hardware to generate the next timer interrupt when they expire. This is why both cases (i.e., scheduling a timer event and acting on timer expiration) involve a call to `etimer_resync_deadlines()`. This function checks on whether either timer or power management deadlines are pending (as they may be rescheduled post expiration). If either type of deadline is found, the function schedules the next interrupt to the earlier of the two by calling `setPop()` (`osfmk/i386/rtclock.c`). If no deadline is pending, `setPop()` is called with a value denoting `EndOfAllTime`. `setPop()` uses the `rtc_timer` global, which sets the timer on the CPU's local APIC. Figure 11-7 shows the flow of `etimer_resync_deadlines`.

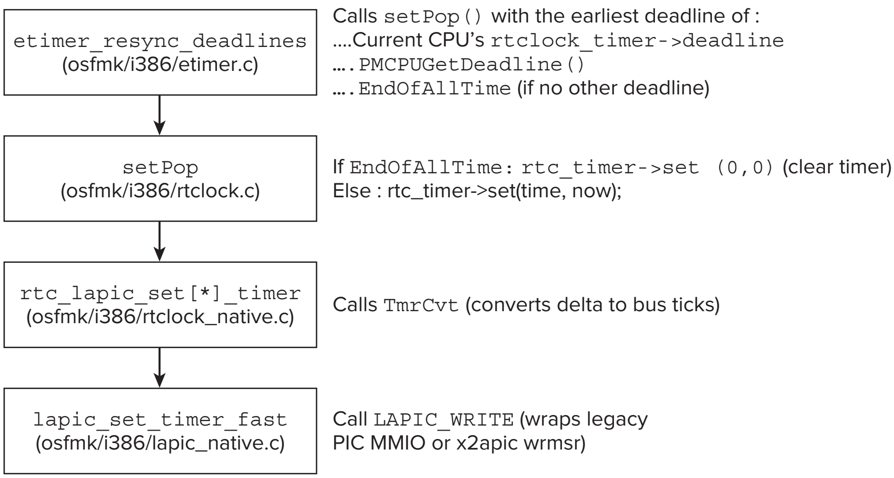


FIGURE 11-7: Setting the hardware pop

> `EndOfAllTime` *is, quite literally, the end of time as we know it. It is set in* `etimer.h` *to* $2^{64}-1$*. Given that there are only some 31.5 million seconds in a year, (*$2^{24.91}$ *or so), this allows for almost* 2^{40} *years to pass, or about* 10^{12}*, which — by some estimates — will be around the time the universe may crunch back into the singularity whence it originated (or expand faster than light could catch up). The Earth will be long gone by then, incinerated by the sun (which will have decayed as well).*

EXCEPTIONS

Recall our low-level discussion of processor traps and exceptions in Chapter 9, one of the kernel's responsibilities is the processing of these events, and in that respect all modern kernels are similar. What is different is the particular approach each kernel may take to achieve this functionality.

Mach takes a unique approach to exceptions implemented over the already-existing message-passing architecture. This model, presented in the following section, is a lightweight architecture and does not actually handle (that is, process and possibly correct) the exception. This is left for an upper layer, which, as you will see in Chapter 13, is BSD.

The Mach Exception Model

The designers of the Mach exception-handling facility mention[1], among others, these factors:

- **Single facility with consistent semantics**: Mach provides only one exception-handling mechanism, for all exceptions, whether user defined, platform agnostic, or platform specific. Exceptions are grouped into exception types, and specific platforms can define specific subtypes.
- **Cleanliness and simplicity**: The interface is very elegant (if less efficient), relying on Mach's already well-defined architecture of messages and ports. This allows extensibility for debuggers and external handlers — and even, in theory, network-level exception handling.

In Mach, exceptions are handled via the primary facility of the kernel: message passing. An exception is little more than a message, which is *raised* (that is, with `msg_send()`) by the faulting thread or task, and *caught* (that is, with `msg_recv()`) by a handler. The handler can then process the exception, and either *clear* the exception (that is, mark the exception as handled, and continue) or decide to *terminate* the thread.

Unlike other models, wherein the exception handler runs in the context of the faulting thread, Mach runs the exception handler in a separate context by making the faulting thread send a message to a predesignated *exception port* and wait for a reply. Each task may register an exception port, and this exception port will affect all threads of the same task. Additionally, individual threads may register their own exception ports, using `thread_set_exception_ports`. Usually, both the task and thread exception ports are NULL, meaning exceptions are not handled. Once created, these ports are just like any other ports in the system, and they may be forwarded to other tasks or even other hosts.

When an exception occurs, an attempt is made to raise the exception first to the thread exception port, then to the task exception port, and finally, to the host (i.e., machine-level registered default) exception port. If none of these result in `KERN_SUCCESS`, the entire task is terminated. As noted, however, Mach does *not* provide exception processing logic — only the framework to deliver the notification of the exception.

Implementation Details

Exceptions usually begin their life as processor traps. To process traps, every modern kernel installs trap handlers. These are low-level functions installed by the kernel's assembly-language core and matching the underlying processor architecture, as described in Chapter 8.

Recall that Mach does not maintain a hardware abstraction layer, yet it aims to provide as clean-cut a dichotomy as possible between the machine-specific and the machine-agnostic parts. The exception codes are included in separate files pertaining to specific architectures and included in the compilation of XNU manually. Architecture-independent exception codes are `#define`d in `<mach/exception_types.h>`. These codes are common to all platforms, and an `#include` of `<mach/machine/exception.h>` provides support for machine-specific subcodes. In the XNU open source, this file is a stub containing an `#include` for `i386/x86_64`'s common `<mach/i386/exception.h>`, and fails compilation (`#error` architecture is not supported) for all other platforms. For iOS, however, Apple defines a `<mach/arm/exception.h>`, which can be found in the iPhone SDK's `usr/include`.

Listing 11-19 shows the common Mach exceptions.

LISTING 11-19: Mach architecture-independent exceptions from <mach/exception_types.h>

```
#define EXC_BAD_ACCESS          1       /* Could not access memory */
                /* Code contains kern_return_t describing error. */
                /* Subcode contains bad memory address. */

#define EXC_BAD_INSTRUCTION     2       /* Instruction failed */
                /* Illegal or undefined instruction or operand */

#define EXC_ARITHMETIC          3       /* Arithmetic exception */
                /* Exact nature of exception is in code field */

#define EXC_EMULATION           4       /* Emulation instruction */
                /* Emulation support instruction encountered */
                /* Details in code and subcode fields   */

#define EXC_SOFTWARE            5       /* Software generated exception */
                /* Exact exception is in code field. */
                /* Codes 0 - 0xFFFF reserved to hardware */
                /* Codes 0x10000 - 0x1FFFF reserved for OS emulation (Unix) */

#define EXC_BREAKPOINT          6       /* Trace, breakpoint, etc. */
                /* Details in code field. */

#define EXC_SYSCALL             7       /* System calls. */

#define EXC_MACH_SYSCALL        8       /* Mach system calls. */

#define EXC_RPC_ALERT           9       /* RPC alert */

#define EXC_CRASH               10      /* Abnormal process exit */
// Mountain Lion/iOS Add code 11 (constant unknown) for ledger resource exceptions
```

Likewise, the Mach exception handler, `exception_triage()` (in `osfmk/kern/exception.c`), is a generic handler responsible for converting exceptions into Mach messages. In both iOS and OS X it is called from `abnormal_exit_notify (osfmk/kern/exception.c)`, with `EXC_CRASH` from BSD's `proc_prepareexit` (`bsd/kern/kern_exit.c`) whenever a process exits with a core dump. Its invocation elsewhere in the kernel, however, is architecture dependent.

On i386/x64, the `i386_exception()` function (from `osfmk/i386/trap.c`) calls `exception_triage()` (shown in Figure 11-8). `i386_exception()` itself can be called from several locations:

- **Low level Interrupt Descriptor Table (IDT) handlers** — `idt.s` and `idt64.s` call `i386_exception()` for kernel mode exceptions by using the CCALL3 and CCALL5 macros (the latter passes five arguments, although `i386_exception()` only takes three).
- **`user_trap()`** (**`osfmk/i386/trap.c`**) — Itself called from the IDT handlers, it calls `i386_exception()` with a code.
- **`mach_call_munger_xx`** functions (i386 and x64, both in **`osfmk/bsd_i386.c`**) — These call `i386_exception()` with `EXC_SYSCALL` on an invalid Mach system call.
- **`fpextovrflt`** (**`osfmk/i386/fpu.c`**) — A specific FPU fault, this is called when the floating point processor generates a memory access fault, either from user-mode or kernel mode.

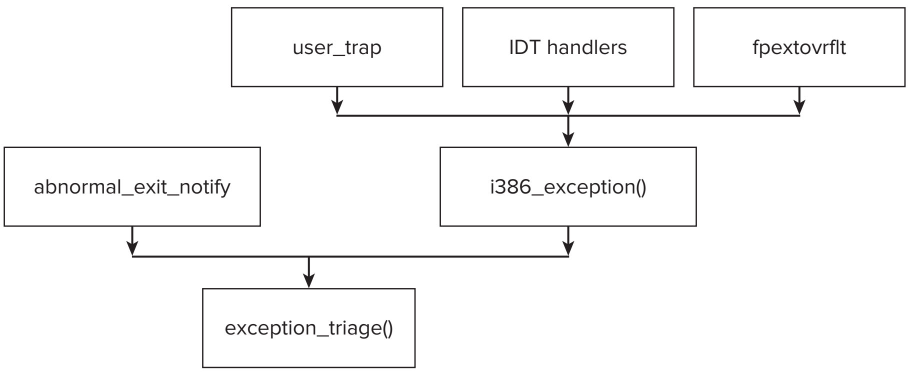


FIGURE 11-8 Exception Triage on OS X

On ARM, it seems that there is no equivalent `arm_exception`, because `exception_triage()` is called directly by the low-level exception handlers:

- **`fleh_swi`** — The system call handler, it calls `exception_triage` with `EXC_SYSCALL` on an invalid system call, or `EXC_BAD_ACCESS`.
- **`sleh_undef`** — This is called from `fleh_undef`, the undefined instruction handler, on an undefined instruction.
- **`sleh_abort`** (called from **`fleh_prefabt`** or **`fleh_dataabt`**, for instruction prefetch or data abort handlers) — From a processor instruction or data abort, it calls `exception_triage` with a code of `EXC_BAD_ACCESS`.

`exception_triage()` works the main exception logic, which — being at the Mach message level — is the same for both architectures. This function attempts to deliver the exception in the manner described previously — thread, task, and finally, host — using `exception_deliver()` (also in `osfmk/kern/exception.c`).

Each thread or task object, as well as the host itself, has an array of exception ports, which are initialized (usually to `IP_NULL`), and may be set using the *`xxx`*`_set_exception_ports()` call, where *`xxx`* is thread, task, or host. The former two are both defined in `osfmk/kern/ipc_tt.c`, and the latter in `ipc_host.c`. Their prototypes are all highly similar:

```
set_exception_ports(xxx_priv_t   xxx_priv,   // xxx is thread, task, or host
        exception_mask_t         exception_mask,
        ipc_port_t               new_port,
        exception_behavior_t     new_behavior,
        thread_state_flavor_t    new_flavor)
```

The "behaviors" (see Table 11-12) are machine-independent indications of what type of message will be generated on exception. Each behavior has a (possibly operating system–specific) "flavor."

TABLE 11-12: Exception behaviors (defined in exception_types.h)

BEHAVIOR	PURPOSE
`EXCEPTION_DEFAULT`	Passes thread identity to exception handler.
`EXCEPTION_STATE`	Passes thread register state to exception handler. Specific "flavors" are in `mach/ARCH/thread_status.h`, and include `THREAD_STATE_X86`, `THREAD_STATE_X64`, and possibly `THREAD_STATE_ARM` in iOS.
`EXCEPTION_STATE_IDENTITY`	Passes both identity and state to exception handler.

The behaviors are implemented by corresponding functions: `[mach]_exception_raise` for `EXCEPTION_DEFAULT`, `[mach]_exception_state_raise` for `EXCEPTION_STATE`, and so on where the function names are the same as the behavior constants (albeit lowercase), and `[mach]` functions are used instead, if the exception code is a 64-bit code.

The various behaviors are handled at the host level by hard-coded exception catchers, `catch_[mach]_exception_xxx`. As before, the function names map to the behaviors (and the `[mach]` variants are for the 64-bit `mach_exception_data_t`). These functions, all in `</bsd/uxkern/ux_exception.c>`, eventually convert the exception to the corresponding UNIX signal by calling `ux_exception`, and deliver it to the faulting thread by `threadsignal`, as discussed in Chapter 12.

The exception ports are the mechanism that enables one of OS X's most important features — the crash reporter. The `launchd(8)` registers its exception ports, and — as ports are inherited across forking — the same exception ports apply to all of its children. Launchd sets ReportCrash as the MachExceptionHandler. This way, when an exception occurs in a launchd job, the crash reporter can be automatically started on demand. Debuggers also make use of exception ports to trap exceptions and break on errors. The following experiment demonstrates aspects of exception handling.

Experiment: Mach Exception Handling

To try exception handling for yourself, code the basic example shown in Listing 11-20:

LISTING 11-20: Mach sample exception handling program, step 1

```
#include <mach/mach.h>
#include <mach/port.h>            // port rights
#include <mach/exception.h>
#include <mach/exception_types.h> // EXC_MASK_*
#include <mach/task.h>            // mach_task_self, etc
#include <stdio.h>                // fprintf..

mach_port_t      myExceptionPort; // Global, for reasons which will become clear later

void signalHandler (int SigNum)
{
   printf("Got signal %d\n", SigNum);
   exit(1);

} // signalHandler

void causeSomeException(int WantUNIXSignals)
{

  char *nullPtr = NULL;
  // If we want UNIX signals, also install a signal handler
  if (WantUNIXSignals) signal(11, signalHandler);

  // Null pointer dereference will result in SIGSEGV, 11.
  // You can try other exceptions (e.g. zero-divide), but
  // remember to change the signal number (e.g. SIGFPE, 8)
  nullPtr[0] = 1;

} // end causeSomeException

void catchMACHExceptions(mach_port_t  TargetTask)
{
  // Simple code to catch exceptions occuring in TargetTask.
  // In step 1, code simply catches, and does nothing.
  kern_return_t  rc;

  exception_mask_t myExceptionMask;

  // create an exception port
  rc = mach_port_allocate (mach_task_self(), MACH_PORT_RIGHT_RECEIVE, &myExceptionPort);

  if (rc != KERN_SUCCESS) { fprintf (stderr, "Unable to allocate exception port\n");
exit(1); }

  // We next call port_insert_right to allow MAKE_SEND, which is required for
  // set_exception_ports
  rc = mach_port_insert_right (mach_task_self(),
                               myExceptionPort, // mach_port_name_t
```

```
                              myExceptionPort, // mach_port_poly_t
                              MACH_MSG_TYPE_MAKE_SEND);

  if (rc != KERN_SUCCESS) { fprintf(stderr,"Unable to insert right\n"); exit(2); }

  myExceptionMask = EXC_MASK_ALL;

  // Now set this port as the target task's exception port
  rc = task_set_exception_ports(TargetTask,
                               myExceptionMask,
                               myExceptionPort,
                               EXCEPTION_DEFAULT_IDENTITY, // Msg 2403
                               MACHINE_THREAD_STATE);

  if (rc != KERN_SUCCESS) { fprintf(stderr,"Unable to set exception\n"); exit(3); }

   // For now, do nothing.

} // end catchMACHExceptions

void main (int argc, char **argv)
{

   int arg, wantUNIXSignals = 0, wantMACHExceptions = 0;

   for (arg = 1; arg < argc; arg++)
     {
         if (strcmp(argv[arg], "-m") == 0) wantMACHExceptions++;
         if (strcmp(argv[arg], "-u") == 0) wantUNIXSignals++;
     }

   // Example first starts capturing our own exceptions. Step 2 will soon
   // illustrate other tasks, so pass ourself as parameter for now

   if (wantMACHExceptions) catchMACHExceptions(mach_task_self());

   causeSomeException(wantUNIXSignals);

   fprintf(stderr,"Done\n"); // not reached

}
```

This simple code offers you three choices:

- **No arguments —** Code will run with the default exception handling.
- **`-u` —** Use this if you want UNIX signals. UNIX signals (in this example, SIGSEGV, Segmentation Fault) will be caught by the signal handler.
- **`-m` —** Use this if you want Mach exception handling. Mach exceptions will be caught by the special setting of exception ports.

Running this code as is will result in a crash if neither the Mach exception nor resulting UNIX signal is caught. Running it with `-u` will indeed catch the UNIX signal, as expected. With `-m`, however, the code will hang, rather than crash. Take a moment to contemplate why that may be.

The program is hanging because it has triggered an exception, and the message is sent to its registered exception port. There is no active receiver on this port, however, and therefore the message hangs indefinitely on the port. Mach exception handling occurs before UNIX exception handling, and therefore the UNIX signal does not get to your process. Because we asked for `EXC_MASK_ALL`, you can replace the crash with other faults, such as a zero divide. You can also experiment with the `EXC_` constants, shown in Listing 11-19.

The program as shown here is useless — it catches an exception, but does not do any handling. A much more useful approach would be to actually do something when notified of an exception. To achieve this, use `mach_msg` to create an active listener on the exception port. This can be accomplished by another thread in the same program, though a more interesting effect is achieved if a second program altogether implements the exception handling part. This is similar to `launchd(1)`'s registration of processes' exception ports, by means of which it can launch `CrashReporter`. The modifications required to turn Listing 11-20 into an external exception handler are shown in Listing 11-21:

LISTING 11-21: Mach sample exception handling program, step 2

```
// Adding an exception message listener:

static void *exc_handler(void *ignored) {

   // Exception handler - runs a message loop. Refactored into a standalone function
   // so as to allow easy insertion into a thread (can be in same program or different)

   mach_msg_return_t rc;

   fprintf(stderr, "Exc handler listening\n");


  // The exception message, straight from mach/exc.defs (following MIG processing)
  // copied here for ease of reference.
  typedef struct {
              mach_msg_header_t Head;
              /* start of the kernel processed data */
              mach_msg_body_t msgh_body;
              mach_msg_port_descriptor_t thread;
              mach_msg_port_descriptor_t task;
              /* end of the kernel processed data */
              NDR_record_t NDR;
              exception_type_t exception;
              mach_msg_type_number_t codeCnt;
              integer_t code[2];
              int flavor;
              mach_msg_type_number_t old_stateCnt;
              natural_t old_state[144];
      } Request;

    Request exc;

    for(;;) {

      // Message Loop: Block indefinitely until we get a message, which has to be
```

```
        // an exception message (nothing else arrives on an exception port)

        rc = mach_msg(
            &exc.Head,
            MACH_RCV_MSG|MACH_RCV_LARGE,
            0,
            sizeof(Request),
            myExceptionPort,       // Remember this was global - that's why.
            MACH_MSG_TIMEOUT_NONE,
            MACH_PORT_NULL);

        if(rc != MACH_MSG_SUCCESS) {  /*... */ return; };

     // Normally we would call exc_server or other. In this example, however, we wish
     // to demonstrate the message contents:

     printf("Got message %hd. Exception : %d Flavor: %d. Code %d/%d. State count is %d\n"
,
             exc.Head.msgh_id, exc.exception, exc.flavor,
             exc.code[0], exc.code[1], // can also print as 64-bit quantity
             exc.old_stateCnt);



#ifdef IOS

   // The exception flavor on iOS is 1

   // The arm_thread_state (defined in the SDK's <mach/arm/_structs.h>)
   // and contains r0-r12, sp, lr, pc and cpsr (total 17 registers). Its count is 17
   // In this example, we print out CPSR and PC.

   struct arm_thread_state *atsh   = &exc.old_state;


   printf ("CPSR is %p, PC is %p, etc.\n", atsh->cpsr, atsh->pc);

#else // OS X


   struct x86_thread_state *x86ts = &exc.old_state;

   printf("State flavor: %d Count %d\n", x86ts->tsh.flavor, x86ts->tsh.count);

   if (x86ts->tsh.flavor == 4) // x86_THREAD_STATE64
    {
       printf ("RIP: %p, RAX: %p, etc.\n",
            x86ts->uts.ts64.__rip, x86ts->uts.ts64.__rax);
    }
   else {
      // Could be x86_THEAD_STATE32 on older systems or 32-bit binaries
      ...
     }
#endif
```

continues

LISTING 11-21 *(continued)*

```
      // You are encouraged to extend this example further, to call on exc_server and
      // perform actual exception handling. But for our purposes, q.e.d.
      exit(1);
     }
} // end exc_handler

…
…

void catchMACHExceptions(mach_port_t  TargetTask)
{
…
  // at the end of catchMachExceptions, spawn the exception handling thread
  pthread_t     thread;
  pthread_create(&thread,NULL,exc_handler,NULL);

} // end catchMACHExceptions

// and simplify the main to be:
int main()
{

   int rc;

   mach_port_t  task;

   // Note: Requires entitlements on iOS, or root on OS X!
   rc = task_for_pid(mach_task_self(),atoi(argv[argc -1]), &task);
   catchMACHExceptions(task);
   sleep (1000);  // Can also loop endlessly. Processing will be in another thread
}
```

To test this code on arbitrary programs, create a simple program to sleep for a few seconds, then crash (pick your poison: NULL pointer dereferencing, zero division, etc.). While the program sleeps, quickly attach the exception handling program. The code will show you something similar to outputs 11-3 and 11-4, on OS X and iOS, respectively (note that the iOS binary needs to be pseudo-signed to allow the `task_for_pid-allow`/`get-task-allow` entitlements).

OUTPUT 11-3: Output of modified exception handling sample, on OS X

```
root@Ergo (/tmp)# cat /tmp/a.c
int main (int argc, char **argv) {
  int c = 24;
  sleep(10);
  c = c /0;
  printf ("Boom\n"); // Not reached
  return(0);
}
```

```
root@Ergo (/tmp)# cc /tmp/a.c -o a
/tmp/a.c: In function 'main':
/tmp/a.c:4: warning: division by zero         # Duh!
/tmp/a.c:5: warning: incompatible implicit declaration of built-in function 'printf'

root@Ergo (/tmp)# /tmp/a  &
[1] 67934

# Attaching to the program, while it sleeps. (Note we are root)
root@Ergo (/tmp)$ ./exc 67934  &
Exc handler listening
Got message 2403. Exception : 3 Flavor: 7 Code: 1/0
State: 44 bytes State flavor: 4 Count 42
RIP: 0x100000ee8, RAX: 0xffff, etc.
```

```
3: EXC_ARITHMETIC
1: EXC_I386_DIV
```

```
morpheus@Ergo (/tmp)$ gdb ./a
Program received signal EXC_ARITHMETIC, Arithmetic exception.
0x0000000100000ee8 in main ()
(gdb) info reg
rax            0xffff  65535
…
rip            0x100000ee8   0x100000ee8 <main+88>
…
```

```
Comparing with
GDB:  perfect
match
```

OUTPUT 11-4: Output of modified exception handling sample, on iOS

```
root@Padishah (…/test)# cat a.c
int main()
{
 char *c = 0L;
 sleep(10);
 c[0] = 1;
 return(0); // not reached
}

root@Padishah (…/test)# ./a &
[1] 2978
```

```
1: EXC_BAD_ACCESS
2: KERN_PROTECTION_FAILURE
```

```
root@Padishah (…/test)# ./exc 2978  &
Exc handler listening
Got message 2403. Exception : 1 Flavor: 1 Code 2/0. State count is: 17
CPSR is 0x10, PC is 0x2250, etc.

root@Padishah (…/test)# gdb ./a
…
Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_PROTECTION_FAILURE at address: 0x00000000
0x00002250 in main ()
…
```

```
Again, compare with GDB.
```

Exception ports are revisited in Chapter 13, which shows how XNU's BSD layer converts the low level Mach exception to the well known UNIX Signals.

SUMMARY

Mach is the microkernel core of XNU. Although Mach is relatively obscure and poorly documented architecture, it still dominates XNU in both OS X and iOS. The chapter thus aimed to demystify and clearly explain the architecture by focusing on its primitive abstractions: at the machine level (host, processor, processor_set, clock), application level (tasks, threads), scheduling (schedulers and exceptions), and virtual memory (pagers).

Implementing additional layers on top of these abstractions is possible. In Chapter 12 you will see the main "personality" XNU exposes to the user, which is the BSD layer. This layer, which uses Mach for its underlying primitives and abstractions, exposes the popular POSIX API to applications, making OS X compatible with many other UNIX implementations. Mach is still, however, the core of XNU, and is present in both OS X and iOS.

REFERENCES

1. Black, David L. et.al., *The Mach Exception Handling Facility*. `http://www.cs.cmu.edu/afs/cs/project/mach/public/www/doc/publications.html`

2a. Abraham Silberschatz, Peter B. Galvin, and Greg Gagne et.al., *Operating System Concepts*. `http://www.amazon.com/Operating-System-Concepts-Windows-Update/dp/0471250600/ref=sr_1_4?ie=UTF8&qid=1343088692&sr=8-4&keywords=tannenbaum+operating+system+concepts`

2b. Tannenbaum, Albert S., `http://www.amazon.com/Modern-Operating-Systems-3rd-Edition/dp/0136006639/ref=pd_sim_b_1`

3. Draves, Bershad, and Rashid, "Using Continuations to Implement Thread Management and Communication in Operating Systems," Oct 1991. Carnegie Mellon University, `http://zoo.cs.yale.edu/classes/cs422/2010/bib/draves91continuations.pdf`

4. CMU-CS-94-142. "Control Transfer in Operating System Kernels," May 13, 1994, Carnegie Mellon University, `citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.55.2132`

12

Commit to Memory: Mach Virtual Memory

The most important resource a kernel manages aside from the CPU itself (see Chapter 11, “Mach Scheduling”) is memory. Mach, like all kernels, devotes a large portion of its code to efficiently handling virtual memory (VM).

This chapter delves into Mach’s powerful VM primitives, as well as the extensible framework of external virtual memory managers, which is used in XNU.

We begin by examining the virtual memory architecture, at a glance. We then discuss physical memory management, followed by an overview of the myriad memory allocators the kernel offers. Finally, we discuss pagers and custom memory managers.

VIRTUAL MEMORY ARCHITECTURE

The most important mechanism provided by Mach is the abstraction of virtual memory, through *memory objects* and *pagers*. As with scheduling and the Mach primitives, we are dealing with an abstraction layer here, with low-level primitives meant to be utilized by an upper layer which, in XNU’s case, is BSD.

The implementation is intentionally broad and generic. It is composed of two layers: the hardware-specific aspects, on top of which are built hardware agnostic, and common aspects. OS X and iOS use a nearly identical underlying mechanism, with the hardware agnostic layer (and the overlying BSD mechanisms) the same, and only the architecture-specific portion changed to the semantics of ARM virtual memory.

This section builds on the discussion of virtual memory started in Chapter 4, “Process Internals,” so if you’ve skipped that chapter and are wondering about the nomenclature, it is defined there. This chapter offers a detailed look at the internals of memory management, and how the commands covered in Chapter 4 actually work. You might also want to have a look at

Chapter 8, which details the kernel's boot process, and details the initialization of the various components listed in this chapter.

The 30,000-Foot View of Virtual Memory

Mach's VM subsystem is, justifiably, as complex and detail-ridden as the virtual memory it seeks to manage. From a high-level view, however, you can see two distinct planes, the virtual and the physical.

The Virtual Memory Plane

The virtual memory plane handles the virtual memory management in a manner that is entirely machine agnostic and independent. Virtual memory is represented by several key abstractions:

- **The `vm_map` (`vm_map.h`):** Represents one or more regions of virtual memory in a task's address space. Each of the regions is a separate `vm_map_entry`, maintained in a doubly linked list of `vm_map_links`.
- **The `vm_map_entry` (`vm_map.h`):** This the key structure, yet it is accessed only within the context of its containing map. Each `vm_map_entry` is a contiguous region of virtual memory. Each such region may be protected with specific access protections (the usual `r/w/x` pertaining to virtual memory pages). Regions may also be shared between tasks. A `vm_map_entry` usually points to a `vm_object`, but may also point to a nested `vm_map`, i.e. a submap.
- **The `vm_object` (`vm_object.h`):** Used to connect a `vm_map_entry` with the actual backing store memory. It contains a linked list of `vm_pages`, as well as a Mach port (called a `memory_object`) to the appropriate *pager*, by means of which the pages may be retrieved or flushed.
- **The `vm_page` (`vm_page.h`):** This is the actual representation of the `vm_object` or a part thereof (as identified by an offset into the `vm_object`). The `vm_page` may be resident, swapped, encrypted, clean, dirty, and so on.

Mach allows for more than one pager. In fact, by default three or four pagers exist. Mach's pagers are considered external entities: dedicated tasks, somewhat akin to the kernel-swapping threads one finds on other systems. Mach's design allows for pagers to be separate kernel tasks, or even user mode ones. Likewise, the underlying backing store can reside on disk swap (handled by the `default_pager` in OS X), can be mapped from a file (and handled by the `vnode_pager`), a device (and its `device_pager`), or even (though unused in OS X) a remote machine.

Note that in Mach, each pager handles the paging request of pages which belong to it, but that request must be made by a `pageout daemon`. These daemons (in reality, kernel threads) maintain the kernel's page lists and decide which pages need to be flushed. There is, therefore, a separation between the paging policy, which the daemons maintain, and the paging operation, which the pagers implement.

The Physical Memory Plane

The physical memory plane handles the mapping to physical memory, because virtual memory eventually has to be stored somewhere. Only one abstraction exists here — the "`pmap`" — but it is an important one, because it offers a machine-independent interface. This interface hides underneath it the platform specifics, which allow paging operations at the processor level — the hardware page table entries (PTEs), translation lookaside buffers (TLBs), and so on.

The Bird's Eye View

Figure 12-1 shows a closer, yet somewhat simplified view of how all these objects connect. It might be a bit overwhelming at first (and remember, it *is* the simplified view!), but the rest of this chapter aims to make sense of it, and discuss each of the abstractions, in detail.

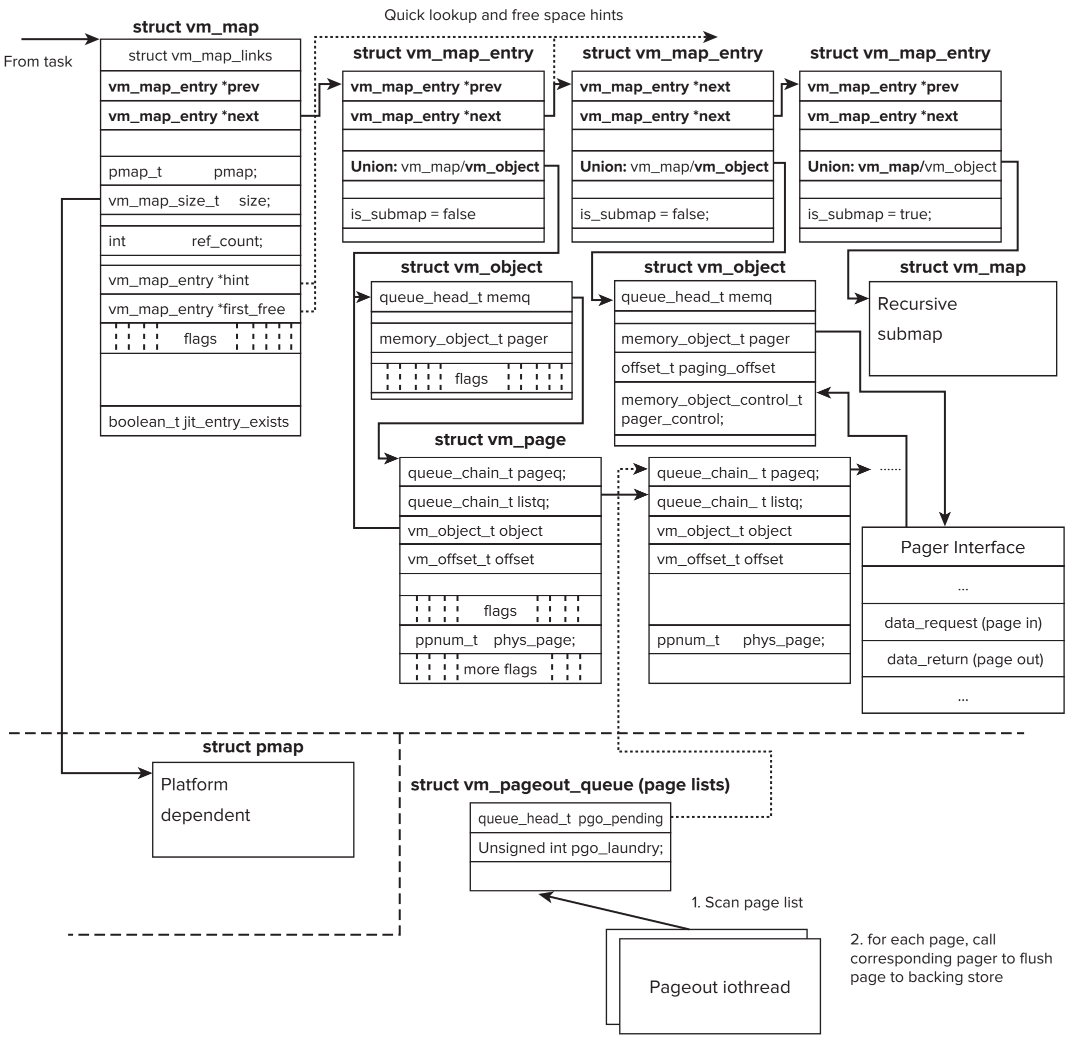


FIGURE 12-1: The menagerie that is the Mach VM

Every Mach task has a virtual memory space of its own, which is held in its "map" member of its `struct task`. This field is a `vm_map` struct. This struct is defined in `osfmk/vm/vm_map.h` as shown in Listing 12-1:

LISTING 12-1: The vm_map struct

```
struct vm_map_header {
        struct vm_map_links     links;          /* first, last, min, max */
        int                     nentries;       /* Number of entries */
        boolean_t               entries_pageable;
                                                /* are map entries pageable? */
        vm_map_offset_t         highest_entry_end_addr; /* The ending address of the
                                                        /* highest allocated
                                                        /* vm_entry_t */
#ifdef VM_MAP_STORE_USE_RB
        struct rb_head  rb_head_store;
#endif
};

struct _vm_map {
        lock_t                lock;           /* uni- and smp-lock */
        struct vm_map_header  hdr;            /* Map entry header */
#define min_offset            hdr.links.start /* start of range */
#define max_offset            hdr.links.end   /* end of range */
#define highest_entry_end     hdr.highest_entry_end_addr
        pmap_t                pmap;           /* Physical map */
        vm_map_size_t         size;           /* virtual size */
        vm_map_size_t         user_wire_limit;/* rlimit on user locked memory */
        vm_map_size_t         user_wire_size; /* current size of user locked memory in
                                              /* this map*/
        int                   ref_count;      /* Reference count */
#if     TASK_SWAPPER
        int                   res_count;      /* Residence count (swap) */
        int                   sw_state;       /* Swap state */
#endif  /* TASK_SWAPPER */
        decl_lck_mtx_data(,   s_lock)         /* Lock ref, res fields */
        lck_mtx_ext_t         s_lock_ext;
        vm_map_entry_t        hint;           /* hint for quick lookups */
        vm_map_entry_t        first_free;     /* First free space hint */
        unsigned int
        /* boolean_t */       wait_for_space:1,  /* Should callers wait for space? */
        /* boolean_t */       wiring_required:1, /* All memory wired? */
        /* boolean_t */       no_zero_fill:1,    /* No zero fill absent pages */
        /* boolean_t */       mapped:1,          /*has this map been mapped */
        /* boolean_t */       switch_protect:1,  /* Protect from write faults while
                                                 /* switched */
        /* boolean_t */       disable_vmentry_reuse:1, // entry alloc. Monotonically
                                                     // increases
        /* boolean_t */       map_disallow_data_exec:1,// set NX bit, if possible
        /* reserved */        pad:25;
        unsigned int          timestamp;      /* Version number */
        unsigned int          color_rr;       /* next color (not protected by a lock) */
#if CONFIG_FREEZE // default freezer — we get to that later.
```

```
        void                  *default_freezer_toc;
#endif
        boolean_t             jit_entry_exists; // used for dynamic codesigning (iOS)
} ;
```

The `vm_map` represents the total memory of `vm_map.size` bytes, maintained in a list (`vm_map.hdr.links`) of `vm_map.hdr.nentries` entries. Each of the links is a `vm_map_entry`, representing a contiguous chunk of virtual memory, with plenty of details about the page range, as shown in Listing 12-2:

LISTING 12-2: A vm_map_entry

```
struct vm_map_entry {
        struct vm_map_links     links;          /* links to other entries */
#define vme_prev                links.prev
#define vme_next                links.next
#define vme_start               links.start
#define vme_end                 links.end

        struct vm_map_store     store;
        union vm_map_object     object;         /* object I point to */
        vm_object_offset_t      offset;         /* offset into object */
        unsigned int
        /* boolean_t */         is_shared:1,    /* region is shared */
        /* boolean_t */         is_sub_map:1,   /* Is "object" a submap? */
        /* boolean_t */         in_transition:1, /* Entry being changed */
        /* boolean_t */         needs_wakeup:1,  /* Waiters on in_transition */
        /* vm_behavior_t */     behavior:2,     /* user paging behavior hint */
                /* behavior is not defined for submap type */
        /* boolean_t */         needs_copy:1,   /* object need to be copied? */
                /* Only in task maps: */
        /* vm_prot_t */         protection:3,   /* protection code */
        /* vm_prot_t */         max_protection:3,/* maximum protection */
        /* vm_inherit_t */      inheritance:2,  /* inheritance */
        /* boolean_t */         use_pmap:1,     /* nested pmaps */
        /*
         * IMPORTANT:
         * The "alias" field can be updated while holding the VM map lock
         * "shared".  It's OK as along as it's the only field that can be
         * updated without the VM map "exclusive" lock.
         */
        /* unsigned char */     alias:8,         /* user alias */
        /* boolean_t */         no_cache:1,      /* should new pages be cached? */
        /* boolean_t */         permanent:1,      /* mapping can not be removed */
        /* boolean_t */         superpage_size:3, /* use superpages of a certain size */
        /* boolean_t */         zero_wired_pages:1, // zero out wired pages on entry
                                                    // deletion
        /* boolean_t */         used_for_jit:1,     // added for dynamic codesigning
                                                    // (iOS)
        /* unsigned char */     pad:1;            /* available bits */
        unsigned short          wired_count;      /* can be paged if = 0 */
        unsigned short          user_wired_count; /* for vm_wire */
};
```

The key element in the `vm_map_entry` is the `vm_map_object`, a union which either holds another `vm_map` (as a submap) or a `vm_object_t` (Because it is a union, determining its contents requires a separate field, the `is_sub_map` boolean). The `vm_object` is a huge, but opaque structure (defined in `osfmk/vm/vm_object.h`, but not readily visible anywhere outside the VM system), which contains all the data necessary to deal with the underlying VM.

In the interest of keeping the avid reader avid (and saving a tree or two), we'll stop short of showing the `vm_object` listing — the structure is, after all, fairly well documented in the header file. Most of the fields in it are bit-wise flags, denoting the underlying memory state (wired, physically contiguous, persistent, etc.) or counters (reference, resident, wired, and so on). Three fields, however, deserve specific mention:

➤ **memq:** Holds the linked list of `struct vm_page` objects, each corresponding to a resident virtual memory page. Though an object can correspond to a single page, more often than not containing an object takes quite a few pages, which is why each page links back to an object at a given offset.

➤ **pager:** Is a `memory_object` structure, which is a Mach port to the pager. A pager connects the non-resident pages to the *backing store* — a memory-mapped file, device, or swap, which holds the pages when they are not in memory. In other words, the pagers (as there can be more than one) are charged with moving data in and out of memory, to their backing store. Pagers are of extreme importance to the virtual memory subsystem, and are discussed in their own section later in this chapter.

➤ **internal**: is one of the many bit-fields in the `vm_page`, and is true if it is used internally by the kernel. This bit affects which pageout queue the page ends up in.

The `vm_page` is a smaller structure, with many bit fields. It participates in two different lists: its `listq` field points to a list of related pages of the same `vm_object`, and is used by the VM Map layer. Its `pageq` field points to one of the kernel's page lists, which is used by the kernel's pageout threads. The `vm_page` also contains a pointer back to its owner `vm_object`, which is used by the kernel's pageout threads to contact its pager when the pageout thread decides to flush this page.

A particularly important `vm_map` instance is the `kernel_map`. This is the *virtual* memory map of the kernel space, and it is used frequently to determine user space or kernel space memory access.

The User Mode View

As with the task and thread APIs discussed in the previous chapter, Mach allows for a remarkable user-level view of virtual memory. User mode can remain blissfully unaware of the gory details, keeping API calls to a `vm_map_t` level, (which is itself an opaque `mach_port_t`) and just ask for specific address ranges, using the rich API presented next.

In Table 12-1, the `vm_map_t` is actually a task parameter; that is, you would pass in a Mach task, whose corresponding VM map would be affected by the calls. There exist variants of these calls with and without the `mach_` prefix: The former is considered to be the "newer" set of APIs (for both 32- and 64-bit), but either set generally works, as in many cases they end up using the same underlying implementation in the kernel.

TABLE 12-1: Mach User-Mode Visible Calls of the VM Subsystem (osfmk/mach/mach_vm.h)

VM SUBSYSTEM FUNCTION	DESCRIPTION
`mach_vm_region``(vm_map_t map, mach_vm_address_t *address, mach_vm_size_t *size, vm_region_flavor_t flavor, vm_region_info_t info, mach_msg_type_number_t *cnt, mach_port_t *object_name);`	Displays information on VM region of task *`map`*, at *`address`* according to *`flavor`*. Currently, only the `VM_BASIC_INFO_64` flavor is supported. *`info`* contains the returned information, in the form of *`count`* entries of structs corresponding to the flavor. `vmmap(1)` uses this extensively; see example. This function calls `vm_map_region()` internally, which calls on `vm_map_lookup_entry()` to find the corresponding entry, and copy its properties into the info struct.
`mach_vm_region_recurse`` ( vm_map_t map, mach_vm_address_t *address, mach_vm_size_t *size, uint32_t *depth, vm_region_recurse_info_t info, mach_msg_type_number_t *infoCnt);`	Similar to `mach_vm_region`, but also recurses into submaps, up to the *`depth`* specified.
`mach_vm_allocate``( vm_map_t map, mach_vm_address_t *address, mach_vm_size_t size, int flags);`	Allocates size bytes in *`map`*, according to *`flags`*. *`Address`* is an in/out parameter — i.e. the kernel will attempt to allocate at the address specified, unless `VM_FLAGS_ANYWHERE` is specified. Note that *`map`* is usually `mach_task_self()`, but given the right permissions, could be any task on the system! When used on `mach_task_self()` this is the underlying system call used by `malloc()` and its ilk. In pre-Leopard OS X, this was the underlying call supporting user mode's `malloc()`. It calls `vm_map_enter()` internally.
`mach_vm_deallocate`` (vm_map_t` *map*`, mach_vm_offset_t` *start*`, mach_vm_size_t` *`size`*);	Inverse of `vm_allocate`. In pre-Leopard OS X, this was the underlying call supporting user mode's `free()`. Calls `vm_map_remove()` internally.
`mach_vm_protect``(vm_map_t map, mach_vm_offset_t start, mach_vm_size_t size, boolean_t` *`set_maximum`*`, vm_prot_t` *`new_protection`*`);`	Sets the protection of the memory region from *`start`* to *`start+size`* in *`map`* to either the maximum defined (if *`set_maximum`*) or *`new_protection`*. Implements BSD's `mprotect(2)`. Calls `vm_map_protect()` internally.

continues

TABLE 12-1 *(continued)*

VM SUBSYSTEM FUNCTION	DESCRIPTION
mach_vm_inherit(vm_map_t map, mach_vm_offset_t start, mach_vm_size_t size, vm_inherit_t new_inherit)	Sets inheritance flags *new_inherit* in the specified range (*start* to *start+size*) of the specified *map*. Implements BSD's minherit(2). Calls vm_map_inherit() internally.
mach_vm_read(vm_map_t map, mach_vm_address_t addr, mach_vm_size_t size, pointer_t *data, mach_msg_type_number_t**dsize*);	memcpy from foreign task: Reads *size* bytes of memory from *addr* in *map* into *data* (of *dsize* bytes). Uses vm_map_copyin() internally.
mach_vm_read_list (vm_map_t *map*, vm_read_entry_t *data_list*, natural_t *count*)	Copies list *data_list* of *count* addresses from the target *map*. Loops over data_list and uses vm_map_copyin() and vm_map_copyout() internally.
mach_vm_write(vm_map_t map, vm_address_t address, pointer_t data, unused mach_msg_type_number_t)	memcpy to foreign task: Writes *data* into *address* in *map*. Uses vm_map_copy_overwrite().
mach_vm_copy(vm_map_t *map*, mach_vm_address_t *source*, mach_vm_size_t *size*, mach_vm_address_t *dest*)	memcpy in foreign task: Copy *size* bytes from *source* to *dest* in *map*. Unlike mach_vm_write, both source and dest are in the foreign map. Implemented using vm_map_copy_in() and vm_map_copy_overwrite().
mach_vm_read_overwrite (vm_map_t *map*, mach_vm_address_t *address*, mach_vm_size_t *size*, mach_vm_address_t *data*, mach_vm_size_t **data_size*)	Similar to vm_read, but overwrites the *data* pointer in the current map. Whereas vm_read would allocate more memory in the current task's map, vm_read_overwrite simply overwrites memory in it. Uses vm_map_copy_overwriteinternally, rather than vm_map_copy_in.
mach_vm_msync(vm_map_t *map*, mach_vm_address_t *address*, mach_vm_size_t *size*, vm_sync_t *sync_flags*);	Synchronizes region, *(address)-(address+size)*, in *map* according to *sync_flags*. Used by BSD's msync(2) system call, and calls on vm_map_msync internally.
mach_vm_behavior_set (vm_map_t *map*, mach_vm_offset_t *start*, mach_vm_size_t *size*, vm_behavior_t *new_behavior*);	Sets paging behavior on range *(start-(start+size))* in *map* to *new_behavior*. Used by BSD's madvise(). Calls on vm_map_behavior_set internally.

VM SUBSYSTEM FUNCTION	DESCRIPTION
mach_vm_map (vm_map_t target_task, mach_vm_address_t *address, mach_vm_size_t size, mach_vm_offset_t mask, int flags, mem_entry_name_port_t object, memory_object_offset_t offset, boolean_t copy, vm_prot_t cur_protection, vm_prot_t max_protection, vm_inherit_t inheritance);	Creates a new memory mapping (as `mmap(2)` does). Maps *object* to address space of *target_task*, at *address*, for *size* bytes, according to *flags*. If *object* is NULL, the map is a zero-filled, anonymous memory. Flags can include: `VM_MAP_ANYWHERE`, allowing the kernel to determine the address `VM_MAP_OVERWRITE`, allowing the kernel to overwrite an existing address and other flags from `<mach/vm_statistics.h>`. The address will be aligned as specified in the *mask*. The mapping can optionally create a *Copy* of object if set (otherwise mapping is direct), and set protection (`VM_PROT_READ, _WRITE, _EXECUTE`) to *cur_protection*, with *max_protection* being the maximum achievable. Likewise, *inheritance* controls this mapping availability to child tasks, if set, by `VM_INHERIT_SHARE, _COPY` (on write), or `_NONE`. Actual work done by the kernel private `vm_map_enter_mem_object()`, which also underlies BSD's `mmap(2)`
mach_vm_machine_attribute(vm_map_t map, mach_vm_address_t addr, mach_vm_size_t size, vm_machine_attribute_t attr, vm_machine_attribute_val_t* value);	Sets machine-specific *attr/value* in *map* for region *addr-(addr+size)*. Calls *vm_map_machine_attribute()* internally.
mach_vm_remap(vm_map_target, mach_vm_offset_t *address, mach_vm_size_t size, mach_vm_offset_t mask, int flags, vm_map_t src, mach_vm_offset_t mem_address, boolean_t copy, vm_prot_t *cur_protection, vm_prot_t *max_protection, vm_inherit_t inheritance);	Remaps memory in task, or between tasks (that is, from *smap* to *tmap*, which may be the same). Also is used to change permissions of a memory mapping. Uses `vm_map_remap()` internally.

continues

TABLE 12-1 *(continued)*

VM SUBSYSTEM FUNCTION	DESCRIPTION
mach_make_memory_entry(vm_map_t target_task, memory_object_size_t *size, memory_object_offset_t offset, vm_prot_t permission, mem_entry_name_port_t *object_handle, mem_entry_name_port_t parent_handle);	Create a "name" reference for a memory region, for later referencing, sharing or changing this region's settings. The named entry can be passed to another task over IPC.
mach_vm_map_page_query (vm_map_t map, mach_vm_offset_t offset, int *disposition, int *ref_count);	Queries information — *ref_count* and *disposition* on the page specified by *offset* in *map*. A passthrough `vm_map_page_query_internal()`.
mach_vm_page_query (vm_map_t target_map, mach_vm_offset_t offset, integer_t *disposition, integer_t *ref_count);	Query residency information about a page. Provides reference count of page in *ref_count*, and `VM_PAGE_QUERY_PAGE_*` flags in disposition. Used by BSD's `mincore(2)`, which translates the `VM_PAGE_QUERY_PAGE_*` flags to `MINCORE_*` flags.
mach_vm_page_info (vm_map_t target_task, mach_vm_address_t address, vm_page_info_flavor_t flavor, vm_page_info_t info, mach_msg_type_number_t *iCnt);	Returns *info* corresponding to mapped page at *address* in *task*. Only flavor supported is `VM_PAGE_INFO_BASIC`. Not to be confused with `vm_page_info()`, which is a function supported only `#if MACH_VM_DEBUG`, and provides virtual/physical mapping information (used by `host_virtual_physical_table()`).
mach_vm_purgable_control(vm_map_t map, mach_vm_offset_t address, vm_purgable_t control, int *state);	Controls purgeable settings of `vm_map` and underlying objects. Purgeable objects may be lost — freed without committing to a backing store — on low memory conditions.

One of the issues addressed by jailbreakers in their iOS kernel patches is the removal of various custom security measures imposed by Apple on memory map handling. Specifically, the `vm_map_protect()` *and* `vm_map_enter()` *are intentionally broken so as to disallow memory regions which are both executable and writable (with the exception of Just-In-Time (JIT) mappings allowed for dynamic-codesigning entitlements). This is meant to discourage hackers from creating code on-the-fly. You can see this for yourself in the code (though why Apple left it public, eludes this author) for* `vm_map_enter()`, *from* `osfmk/vm/vm_map.c`*:*

```
#if CONFIG_EMBEDDED
        if (cur_protection & VM_PROT_WRITE){
                if ((cur_protection & VM_PROT_EXECUTE) && !(flags
                & VM_FLAGS_MAP_JIT)){
                        printf("EMBEDDED: %s curprot cannot be
                        write+execute. turning off execute\n",
                        __PRETTY_FUNCTION__);
                        cur_protection &= ~VM_PROT_EXECUTE;
                }
        }
#endif /* CONFIG_EMBEDDED */
```

Similarly, in the same file, the implementation of `vm_map_protect()` *makes it so that an executable page cannot be made writable:*

```
#if CONFIG_EMBEDDED
                if (new_prot & VM_PROT_WRITE) {
                       if ((new_prot & VM_PROT_EXECUTE) && !
                        (current->used_for_jit)) {
                                printf("EMBEDDED: %s can't have
                                both write and exec at the same
                                time\n", __FUNCTION__);
                                new_prot &= ~VM_PROT_EXECUTE;
                        }
                }
#endif
```

Jailbreakers simply patch both functions, so as to NOP out the check in `vm_map_enter()` *and the flag clearing in* `vm_map_protect()`*. By patching the low-level Mach APIs, they handle both Mach calls and BSD.*

An important function that was left out of `osfmk/mach/mach_vm.h` (and therefore Table 11-1) is `[mach_]vm_wire()`. It is defined instead in `osfmk/mach/host_priv.h` (and implemented in `osfmk/vm/vm_user.c` as shown in Listing 12-3:

LISTING 12-3: mach_vm_wire, from osfmk/vm/vm_user.c:

```
/*
 * NOTE: these routine (and this file) will no longer require mach_host_server.h
 * when mach_vm_wire and vm_wire are changed to use ledgers.
 */
#include <mach/mach_host_server.h>
/*
 *      mach_vm_wire
 *      Specify that the range of the virtual address space
 *      of the target task must not cause page faults for
 *      the indicated accesses.
 *
 *      [ To unwire the pages, specify VM_PROT_NONE. ]
 */
kern_return_t
mach_vm_wire(
        host_priv_t             host_priv,
        vm_map_t                map,
        mach_vm_offset_t        start,
        mach_vm_size_t  size,
        vm_prot_t               access)
```

The function allows its caller to "hard-wire" virtual memory (read: part of a `vm_map`), so that it remains resident and unpageable. Because this affects the host's RAM and thereby impacts other programs as well, it is defined as a privileged host level operation (ergo the `host_priv` port as its first argument). The function has yet, at this time of writing, to be converted to using Mach ledgers (see Chapter 10), but it is possible that in Mountain Lion it finally will.

Many of Mach VM functions are also functionally equivalent to POSIX system calls. In fact, BSD memory management system calls (in `bsd/kern/kern_mman.c`) are usually implemented directly over the Mach system calls. This is indicated in the table. For example, BSD's `msync(2)` calls `mach_vm_msync`. `madvise(2)` calls `mach_vm_behavior_set()`. The `mlock(2)`/`munlock(2)` calls are simple wrappers over `mach_vm_wire()`, and so on. User mode memory allocation, which used to be implemented over the Mach calls, has been moved to POSIX. Chapter 13 discusses the POSIX memory management calls.

The Mach APIs, however, are far stronger than those offered by POSIX, particularly due to the ease with which they allow one task to invade another's address space. Permissions are required for this (specifically, the foreign task's port, which is the "map" argument in Table 12-1's Mach calls). Barring this minor technicality, however, these calls offer virtually boundless power. Indeed, many process invasion and thread injection techniques in OS X rely on these Mach calls, not on those of BSD.

Experiment: Emulating vmmap(1) with mach_vm_region_recurse

The `mach_vm_region_recurse` is the main Mach call used in `vmmap(1)` and GDB's `show regions` command. You can see a good example of its usage in the GDB sources (specifically, `macos_debug_regions()`, in `gdb/macosx/macosx-nat-inferior-debug.c`). The output of `vmmap(1)` is, for the most part, that of `vm_region64` with `VM_REGION_BASIC_INFO` , as shown in Listing 12-4:

LISTING 12-4: The VM_REGION_BASIC_INFO_64 struct, from <mach/vm_region.h>

```
struct vm_region_basic_info_64 {
        vm_prot_t               protection;     // VM_PROT_* flags
        vm_prot_t               max_protection; // likewise, for max possible
        vm_inherit_t            inheritance;    // VM_INHERIT_[SHARE|COPY|NONE]
        boolean_t               shared;
        boolean_t               reserved;
        memory_object_offset_t  offset;
        vm_behavior_t           behavior;       // VM_BEHAVIOR_*, like madvise(2)
        unsigned short          user_wired_count;
};
```

Constructing a quick and dirty implementation of `vmmap(1)` is straightforward, by relying on this call, as is shown in Listing 12-5:

LISTING 12-5: A simple implementation of vmmap(1)

```
// Region listing code adapted from GDB's macosx_debug_regions, from open source GDB

void show_regions (task_t task, mach_vm_address_t address)
{
  kern_return_t kr;
  vm_region_basic_info_data_t info, prev_info;
  mach_vm_address_t prev_address;
  mach_vm_size_t size, prev_size;

  mach_port_t object_name;
  mach_msg_type_number_t count;

  int nsubregions = 0;
  int num_printed = 0;
  int done = 0;

  count = VM_REGION_BASIC_INFO_COUNT_64;
  // Call mach_vm_region, which obtains the vm_map_entry containing the address,
  // and populates the vm_region_basic_info_data_t with its statistics

  kr = mach_vm_region (task, &address, &size, VM_REGION_BASIC_INFO,
                      (vm_region_info_t) &info, &count, &object_name);
  if (kr != KERN_SUCCESS)
    {
      printf ("Error %d - %s", kr, mach_error_string(kr));
      return;
    }
  memcpy (&prev_info, &info, sizeof (vm_region_basic_info_data_t));
  prev_address = address;
  prev_size = size;
  nsubregions = 1;

  while (!done)
```

continues

LISTING 12-5 *(continued)*

```
    {
      int print = 0;

      address = prev_address + prev_size;

      /* Check to see if address space has wrapped around. */
      if (address == 0)
        {
        print = done = 1;
        }

      if (!done)
        {
          // Even on iOS, we use VM_REGION_BASIC_INFO_COUNT_64. This works.

          count = VM_REGION_BASIC_INFO_COUNT_64;


          kr =
            mach_vm_region (task, &address, &size, VM_REGION_BASIC_INFO,
                              (vm_region_info_t) &info, &count, &object_name);

          if (kr != KERN_SUCCESS)
            {
               fprintf (stderr,"mach_vm_region failed for address %p - error %d\n",
               address, kr);
               size = 0;
               print = done = 1; // bail on error, but still print
            }
        }

      if (address != prev_address + prev_size)
        print = 1;
  // Print if there has been any change in region settings
      if ((info.protection != prev_info.protection)
          || (info.max_protection != prev_info.max_protection)
          || (info.inheritance != prev_info.inheritance)
          || (info.shared != prev_info.reserved)
          || (info.reserved != prev_info.reserved))
        print = 1;

      if (print)
        {
          int   print_size;
          char *print_size_unit;
          if (num_printed == 0)
            printf ("Region ");
          else
            printf ("   ... ");
```

```
          /* Quick hack to show size of segment, which GDB does not */
          print_size = prev_size;
          if (print_size > 1024) { print_size /= 1024; print_size_unit = "K"; }
          if (print_size > 1024) { print_size /= 1024; print_size_unit = "M"; }
          if (print_size > 1024) { print_size /= 1024; print_size_unit = "G"; }
          /* End Quick hack */

          // the xxx_to_yyy functions merely change the flags/bits to a more readable
          // string representation. Their implementation is left as an exercise to
          // the reader

          printf (" %p-%p [%d%s](%s/%s; %s, %s, %s) %s",
                           (prev_address),
                           (prev_address + prev_size),
                           print_size,
                           print_size_unit,
                           protection_bits_to_rwx (prev_info.protection),
                           protection_bits_to_rwx (prev_info.max_protection),
                           unparse_inheritance (prev_info.inheritance),
                           prev_info.shared ? "shared" : "private",
                           prev_info.reserved ? "reserved" : "not-reserved",
                           behavior_to_xxx (prev_info.behavior));

          if (nsubregions > 1)
            printf (" (%d sub-regions)", nsubregions);

          printf ("\n");

          prev_address = address;
          prev_size = size;
          memcpy (&prev_info, &info, sizeof (vm_region_basic_info_data_t));
          nsubregions = 1;

          num_printed++;
        }
      else
        {
          prev_size += size;
          nsubregions++;
        }


      if (done)
        break;
    }

} // end show_regions
void main(int argc, char **argv)
{
        struct vm_region_basic_info vmr;
        kern_return_t   rc;
        mach_port_t     task;
```

continues

LISTING 12-5 *(continued)*

```
        mach_vm_size_t  size = 8;
        vm_region_info_t        info = (vm_region_info_t) malloc(10000);
        mach_msg_type_number_t  info_count;
        mach_port_t             object_name;
        mach_vm_address_t       addr =1;
        int pid;
        if (!argv[1]) { printf ("Usage: %s <PID>\n"); exit (1);}
        pid = atoi(argv[1]);

        // Obtain task port, using task_for_pid().
        rc = task_for_pid(mach_task_self(),pid, &task);

        if (rc) {
             fprintf (stderr, "task_for_pid() failed with error %d - %s (Am I entitled?)
             \n", rc,
             mach_error_string(rc));
             exit(1);
                }
        printf ("Task: %d\n", task);
        show_regions (task, addr);
        printf("Done\n");

}
```

You are encouraged to try this code in OS X, and especially in iOS — wherein `vmmap(1)` is a much needed binary. In iOS, however, running this code will fail in the `task_for_pid()` call, even if you are root! One extra step is required — getting past the kernel's `task_for_pid()` protection, by entitling your code to use `task_for_pid()`. To do this, you can use the entitlement file from Chapter 3, which enables the `task_for_pid-allow` entitlement. Try putting in "0" as the PID for a pleasant surprise.

> *This* `vmmap(1)` *example in Listing 12-5 can easily be adapted to be even more intrusive, including dumping the process memory map to disk, and even writing to it. Amit Singh's excellent website contained a program called gcore to dump an active process' memory map to a core compatible format, which can be then inspected with GDB. This book provides a companion tool, corerupt, which expands these abilities further in order to provide support for iOS, as well as dumping encrypted segments or modifying the active memory image!*

PHYSICAL MEMORY MANAGEMENT

Although the kernel, like user space, operates almost exclusively in the virtual address space, virtual memory must inevitably be translated into physical addresses. The machine's RAM is, in effect, a window into virtual memory, providing access to finite, often disjointed regions of virtual memory, up to however much memory the machine has. The rest of the virtual memory is either lazily allocated, shared, or swapped to external stores, most often the disk.

Physical memory management, however, is specific to the underlying architecture. Although the concepts of virtual and physical memory are inherently the same across all architectures, the underlying implementations are full of idiosyncrasies. XNU builds on Mach's physical memory abstraction layer, called pmap. This layer, by its very design, allows for a uniform interface to the physical memory, which hides the architecture specifics. This is naturally of great use to XNU, which was previously adapted to the physical memory landscape of PowerPC, is now primarily on Intel, and — in iOS — is built on ARM. In the words of Rashid and Tevanian themselves, a pmap implementor "needs to know very little about the way Mach functions, but will need to know very much about underlying architecture."[1]

The pmap layer of the x86 architecture, as well as the now-deprecated PowerPC, are both part of the open-source XNU employed in OS X. The same, lamentably, cannot be said for ARM. This section thus focuses more on the interface, which is largely the same in all cases, and shows some implementation specifics on the Intel architecture.

The PMAP APIs

Mach's pmap is logically comprised of two sublayers:

- **The machine-independent layer:** Provides a set of APIs that are largely machine agnostic, These APIs, defined in `<osfmk/vm/pmap.h>`, require only that the machine support the basic concepts of VM paging. Note, we say "largely," because the header isn't perfectly free of `#ifdef`'s for `_i386` and `__LP64__`, though it does remain at a higher level. The VM layer only sees and passes around a `pmap_t`, which is a pointer to a `struct pmap`, effectively a void pointer.
- **The machine-dependent layer:** Ties pmap to a specific implementation, and deals with the nooks and crannies of the underlying architecture. These are the set of `#defines` specific to the particular hardware, such as PTE (page table entry) macros, bitmasks, registers (Intel's CR3 and ARM's c7-c8), as well as the definition of the basic `struct pmap`, (in `osfmk/_arch_/pmap.h`), which the `pmap_t` is only a reference to.

This layer is tied to the machine-independent one via `#ifdefs` and `#includes`: From `<osfmk/machine/pmap.h>`, which in turn includes the hardware specific header; that is, `<osfmk/i386/pmap.h>`, ppc, arm, and so on. Additionally, the implementation of the machine-independent functions, from `osfmk/vm/pmap.h` , is in the machine-dependent `pmap.c` file, which is in `osfmk/_arch_/pmap.c`.

In object-oriented terms, the machine-independent layer can be considered to be the interface to pmap, and the machine-dependent layer is the implementation. From a software-engineering standpoint, as long as the interface does not change, its clients (i.e, the Mach VM subsystem) can remain blissfully unaware of the details. The pmap specifics are thus opaque to Mach's VM. This maximizes portability, but does come at the cost of performance.

Table 12-2 shows some pmap APIs, from the machine independent layer:

TABLE 12-2: Some of the pmap APIs, from osfmk/vm/pmap.h

PMAP FUNCTION	USED FOR
`pmap_t` **`pmap_create`** `(vm_map_size_t` *`size`*`,` `boolean_t` *`is_64bit`*`);`	A constructor for `pmap_t` objects. Note the `pmap_t` (struct pmap) is architecture dependent, and therefore the returned value is opaque to the caller. The *size* argument is always 0 for a hardware-backed pmap. The second argument — *`is_64bit`* — is used only on Intel 32-bit platforms (`__i386__`). The `pmap_t` is created (the struct pmap is allocated from the pmap zone, discussed in the section, "Mach Zones," later this chapter). Additionally, any hardware page table entries are initialized. An internal reference count is also set to 1.
`void` **`pmap_reference`**`(pmap_t` *`pmap`*`);`	Increases the reference count of a `pmap_t`. Throughout the kernel this is only used by `kmem_suballoc()`, which (as you will see later) can be used to allocate memory as a suballocation of an existing allocation.
`void` **`pmap_destroy`**`(pmap_t` *`pmap`*`);`	Decreases the reference count of a `pmap_t`. This also serves as the destructor of `pmap_t` objects, when the reference count drops to 0.
`void` **`pmap_enter[_options]`** `( pmap_t` *`pmap`*`,` `vm_map_offset_t v,` `ppnum_t` *`pn`*`,` `vm_prot_t` *`prot`*`,` `unsigned int` *`flags`*`,` `boolean_t` *`wired`*`,` `[unsigned int` *`options`*`]);`	Establishes a mapping from virtual address *`v`* to physical page number *`pn`* in *`pmap`*. Sets MMU page protection to *`prot`* (the standard *`rwx`* page permissions). The flags can include `VM_MEM_GUARDED` and `VM_MEM_NOT_CACHEABLE`, which toggle the page cacheability. *`Wired`* marks the page as such, as in resident and not swappable. Uppercase wrapper macros are available for both `_enter` variants, which first ensure the page is not encrypted.
`void` **`pmap_page_protect`** `(ppnum_t` *`phys`*`,` `vm_prot_t` *`prot`*`);`	Changes `VM_PROT` bits on physical page number *`phys`* according to *`prot`*.
`void` **`pmap_zero_page`** `(ppnum_t` *`pn`*`);`	Zeros physical page

PMAP FUNCTION	USED FOR
`unsigned int` **`pmap_disconnect`**`(ppnum_t` *`pa`*`)`	Disconnects a previous page mapping (and returns `VM_MEM_MODIFIED` and `VM_MEM_REFERENCED` flags, if set)
`void` **`pmap_remove`**`(pmap_t` *`map`*`, addr64_t` *`s64`*`, addr64_t` *`e64`*`);`	Removes addresses from s64 through e64. Internally, this method converts the s64–e64 range to a set of page table entries, and calls `pmap_remove_range()`.
`void` **`pmap_switch`**`(pmap_t` *`tpmap`*`);`	Switches to a new pmap. On Intel, this merely disables interrupts and calls `set_dirbase()`, which changes the value of CR3, unless switching between related threads, or between kernel and user (with CR3 shared). Most switching is done by the `PMAP_[DE]ACTIVATE` family of macros, which on Intel is `set_dirbase()` as well.
`void *`**`pmap_steal_memory`**`(vm_size_t` *`size`*`);`	"Steals" physical memory before VM is fully initialized

The pmap's low-level memory functions, which accept `pnum_t` arguments, can operate directly on physical pages.

The pmaps can be nested (so as to contain other pmaps). This is a fairly common technique, which is relied upon heavily to allow the sharing of memory — both implicit (shared libraries) and explicit (`mmap(2)`). Also, similarly to the `kernel_map vm_map`, there exists a global `kernel_pmap`, which holds the *physical* memory pages used by the kernel.

API Implementation Example on Intel Architecture

To further comprehend how pmap can present a machine-independent interface to its clients, consider a specific case — page entry bits on the Intel architecture, as shown in Figure 12-2. The illustration specifically follows `VM_MEM_SUPERPAGE` and `VM_PROT_WRITE` (`osfmk/mach/vm_prot.h`), but you can also deduce `VM_NOT_CACHEABLE` and other flags as well.

Figure 12-2 shows how the flags in `osfmk/vm/pmap.h` are translated (by `pmap_enter`, in `osfmk/i386/x86_common.c`) to the specific page entry bits for Intel PTEs, as defined in the Intel architecture manuals. The conversion is done in the platform-specific implementation of `pmap_enter()`, which maintains the platform-independent interface, flags, and options. Many other pmap functions are implemented in this manner.

The `pmap_t` implementation on Intel architectures is defined in `osfmk/i386/pmap.h` as in Listing 12-6. The reader is encouraged to make a segue here to the appendix in this book, which refreshes the Intel architecture implementation of virtual memory.

FIGURE 12-2: Translation of platform-independent pmap flags to platform-dependent ones

LISTING 12-6: The Intel pmap_t implementation:

```
struct pmap {
        decl_simple_lock_data(,lock)    /* lock on map */
        pmap_paddr_t    pm_cr3;         /* physical addr */
        boolean_t       pm_shared;
        pd_entry_t      *dirbase;       /* page directory pointer */
#ifdef __i386__
        pmap_paddr_t    pdirbase;       /* phys. address of dirbase */
        vm_offset_t     pm_hold;        /* true pdpt zalloc addr */
#endif
        vm_object_t     pm_obj;         /* object to hold pde's */
        task_map_t      pm_task_map;
        pdpt_entry_t    *pm_pdpt;       /* KVA of 3rd level page */
        pml4_entry_t    *pm_pml4;       /* VKA of top level */
        vm_object_t     pm_obj_pdpt;    /* holds pdpt pages */
        vm_object_t     pm_obj_pml4;    /* holds pml4 pages */
#define PMAP_PCID_MAX_CPUS      (48)    /* Must be a multiple of 8 */
        pcid_t          pmap_pcid_cpus[PMAP_PCID_MAX_CPUS];
        volatile uint8_t pmap_pcid_coherency_vector[PMAP_PCID_MAX_CPUS];
        struct pmap_statistics  stats;  /* map statistics */
        int             ref_count;      /* reference count */
        int             nx_enabled;     // Data Execution Prevention
};
```

MACH ZONES

Zones are Mach's (and XNU's) idea of what Linux calls memory caches, and Windows call Pools (q.v. Windows has its `ExAllocatePool/WithTag`). Zones are memory regions used for the quick allocation and deallocation of frequently used objects of fixed size. The Zone API is internal to the kernel and cannot be accessed from user mode. Nonetheless, zones are used extensively in Mach.

> *This section discusses kernel zones, which are entirely different from and not to be confused with* `malloc()` *zones (i.e.* `malloc_create_zone(3)` *and friends). The latter are in user mode, part of the C runtime library, and well documented in man pages.*

To display zones, you can use the `zprint(1)` command. the command relies on the `mach_zone_info()` functionality exposed by the host port. Lion adds a `task_zone_info()` function, displaying zone utilization by a particular task (and also enables `zprint(1)`'s `-p` switch, which displays a zone listing for a particular process). Since `zprint(1)` is open source and fairly short, the intrigued reader is encouraged to have a look at its source.

The Mach Zone Structure

A zone is a structure defined in `osfmk/kern/zalloc.h`, as shown in Listing 12-7:

LISTING 12-7 : Mach zones

```
struct zone {
        int             count;          /* Number of elements used now */
        vm_offset_t     free_elements;  // Linked list of free elements
        decl_lck_mtx_data(,lock)        /* zone lock */
        lck_mtx_ext_t   lock_ext;       /* placeholder for indirect mutex */
        lck_attr_t      lock_attr;      /* zone lock attribute */
        lck_grp_t       lock_grp;       /* zone lock group */
        lck_grp_attr_t  lock_grp_attr;  /* zone lock group attribute */
        vm_size_t       cur_size;       /* current memory utilization */
        vm_size_t       max_size;       /* how large can this zone grow */
        vm_size_t       elem_size;      /* size of an element */
        vm_size_t       alloc_size;     /* size used for more memory */
        uint64_t        sum_count;      /* count of allocs (life of zone) */
        // the following italicized fields can be changed with zone_change()
        unsigned int
        /* boolean_t */ exhaustible :1, /* (F) merely return if empty? */
        /* boolean_t */ collectable :1, /* (F) garbage collect empty pages */
        /* boolean_t */ expandable :1,  /* (T) expand zone (with message)? */
        /* boolean_t */ allows_foreign :1,/* (F) allow non-zalloc space */
        /* boolean_t */ doing_alloc :1, /* is zone expanding now? */
        /* boolean_t */ waiting :1,     /* is thread waiting for expansion? */
        /* boolean_t */ async_pending :1,/* asynchronous allocation pending? */
#if CONFIG_ZLEAKS
        /* boolean_t */ zleak_on :1,   /* Are we collecting allocation info? */
#endif  /* ZONE_DEBUG */  // they mean CONFIG_ZLEAKS — mistake in source
        /* boolean_t */ caller_acct: 1,/* account allocation/free to caller? */
        /* boolean_t */ doing_gc :1,    /* garbage collect in progress? */
        /* boolean_t */ noencrypt :1;
        int             index;  /* index into zone_info arrays for this zone */
        struct zone *   next_zone;      /* Link for all-zones list */
        call_entry_data_t  call_async_alloc; /* callout for asynch alloc */
        const char      *zone_name;     /* a name for the zone */

#if     ZONE_DEBUG
        queue_head_t    active_zones;   /* active elements */
#endif  /* ZONE_DEBUG */

#if CONFIG_ZLEAKS
        uint32_t num_allocs;    /* alloc stats for zleak benchmarks */
        uint32_t num_frees;     /* free stats for zleak benchmarks */
        uint32_t zleak_capture; /* per-zone counter for capturing every N allocations */
#endif /* CONFIG_ZLEAKS */
};
```

Aside from the plentiful debug information (which is enabled on zones only if XNU is compiled with `CONFIG_ZLEAKS`), a zone is really a rather small structure containing a linked list of free elements, and the zone statistics.

To create and handle zones, Mach offers several functions, all defined in the same header file, and implemented in `osfmk/kern/zalloc.c` as shown in Table 12-3.

TABLE 12-3: Zone Functions from osfmk/kern/zalloc.h

ZONE FUNCTION	DESCRIPTION
`zone_t zinit(` `vm_size_t size,` `vm_size_t maxmem,` `vm_size_t alloc,` `const char *name);`	Returns a new zone named *name*, which can hold elements of *size* bytes. If the zone is full, an additional *alloc* bytes will be allocated. Allocation of the zone is done asynchronously by the `thread_call_daemon` (and the `call_async_alloc` data).
`void *zalloc(zone_t zone);` `void *zalloc_noblock` `(zone_t zone);` `void *zalloc_canblock` `(zone_t zone,` `boolean_t canblock);`	Allocates an element from the *zone*. The element allocated is of the fixed size set when the zone was created, by *zinit*. Both the former use the last, passing `canblock = TRUE` and `FALSE`, respectively.
`void zcram(` `register zone_t zone,` `void *newaddr,` `vm_size_t size)`	Adds ("crams") the memory at `newaddr`, of *size* bytes to the zone specified by *zone*.
`void zfree(` `zone_t zone,` `void *elem);`	Frees the element pointed to by *elem*, which must be in the zone specified by *zone*. Free elements may be garbage collected.
`void zone_change` `(zone_t zone,` `unsigned int item,` `boolean_t value);`	Changes zone properties by setting corresponding field in *zone* to *value*. `Z_NOENCRYPT`: Zone is unencrypted during hibernation (true for virtually all zones) `Z_EXHAUSTIBLE`: Zone is of finite size, and may be empty. `Z_COLLECT`: Toggles garbage collection `Z_EXPAND`: Zone may be expanded `Z_FOREIGN`: Zone can contain non-`zalloc()`ed object `Z_CALLERACCT`: The calling thread will be held accountable, memory quota-wise, for zone allocations.

All zones memory is effectively pre-allocated in the call to `zinit()` (by a call to `kernel_memory_allocate()`, which is a low-level allocator, discussed in the next section). Calls to `zalloc()` are effectively wrappers over a `REMOVE_FROM_ZONE` macro, which returns the next element from the zone's free list (and resorts to `kernel_memory_allocate()` of the zone's `alloc_size` bytes, if the zone is full). A `zfree()` uses the opposite macro, `ADD_TO_ZONE`. Both functions also perform a fair amount of sanity checking, which hasn't helped much so far: Zone allocation bugs in the past have provided several exploitable memory corruptions. The more important client of `zalloc()` is the kernel's `kalloc()`, which allocates from `kalloc.*` zones (discussed in the next section). BSD's `mcache` mechanism (see Chapter 13) also allocates from its own zone (also called `mcache`), as do BSD kernel zones, which are built directly over the Mach ones.

Zone Setup During Boot

Zones are set up during the kernel boot by two calls from `vm_mem_bootstrap()` (refer to Chapter 8 for the full details on this function)

- The first, to `zone_bootstrap()`, sets up the master zone ("zones") wherein all other zone data is stored.
- The second, to `zone_init()`, initializes the zone subsystem locks and pages (using `zone_page_init()`).

The zone handling functions are in `osfmk/kern/zalloc.c`. Individual zones can then be created by various subsystems.

The `zone_init()` function takes an argument — `zsize`. This argument is set by default to one quarter of `maxmem`, but may be overridden by a kernel command-line argument (specified in MB), in which case it must be between `ZONE_MAP_MIN` and `ZONE_MAP_MAX`. You can set these values as part of the kernel configuration (that is, using `CONFIG_*`) macros.

There are quite a few zones in XNU — about 120 in SL and more than 170 in Lion. These zones are, for the most part, created by their corresponding subsystem's `init` function during the kernel boot. Table 12-4 lists but a few.

TABLE 12-4: Some of the More Important Mach Zones Used in OS X

ZONE NAME	ALLOCATED BY	USED FOR
`Alarms`	`clock_service_create()` `osfmk/kern/clock_oldops.c`	Clock alarms.
`buf headers` `buf.`*nn*	`Bufzoneinit` `bsd/vfs/vfs_bio.c`	VFS buffers. The *nn* zones are powers of two, from 512 through 8192.
`dtrace.dtrace_probe_t`	`dtrace_init` `bsd/dev/dtrace/dtrace.c`	DTrace probes.
`ipc spaces` `ipc tree entries` `ipc ports` `ipc port sets`	`ipc_bootstrap` `osfmk/ipc/ipc_init.c`	Various Inter Process Commication constructs.

ZONE NAME	ALLOCATED BY	USED FOR
`kalloc.`*nn* `kalloc.large` (fake zone)	`kalloc_init` `osfmk/kern/kalloc.c` `osfmk/kern/zalloc.c`	Kernel allocations. Zones are created for powers of 2 from 16 to 8192, as well as a "large" zone. Calls to `kalloc()` then allocate from the corresponding zone, or use `kmem_alloc()` if too large. iOS 5 also has zones which are not powers of 2.
`kernel_stacks` (fake zone)	`osfmk/kern/zalloc.c`	Records kernel stack utilization.
`maps` `non-kernel.map.entries` (iOS: VM map entries) `kernel.map.entries` (iOS: reserved VM map entries) `map.copies`	`vm_map_init() osfmk/vm/vm_map.c`	Zones used for the various kernel `vm_map`.
`mcache` `mcache.bkt_`*nn* `mcache.audit`	`mcache_init` `bsd/kern/mcache.c`	BSD's `Mcaches`, which are implemented over zones.
Tasks	`task_init` `osfmk/kern/task.c`	Mach task objects.
Threads	`thread_init` `osfmk/kern/thread.c`	Mach thread objects.
`page_tables` (fake zone)	`osfmk/kern/zalloc.c`	PTEs. This is among the largest zones in the kernel on i386/x86_64.
Pmap	`map_init` `osfmk/x86_64/pmap.c`	Page maps.
Uthreads	`uthread_zone_init` `bsd/kern/kern_fork.c`	BSD Thread objects.
Zones	`zone_bootstrap()` `osfmk/kern/zalloc.c`	The "zone of zones," where all zone data is stored.

Zone Garbage Collection

If the system is low on memory, zones may undergo garbage collection. This is handled by `consider_zone_gc()`(from `osfmk/kern/zalloc.c`) which is called by the `vm_pageout_garbage_collect` thread. `consider_zone_gc` may choose to invoke the zone garbage collection (`zone_gc`) in one of the following situations:

- `zfree()` has freed an element in a zone that was more than one page, and the system `vm_pool` is low
- It has been a while since `zone_gc` last ran, as specified by `zone_gc_time_throttle`.
- The system is hibernating, and `hibernate_flush_memory()` has been called.

These situations are depicted by the Figure 12-3.

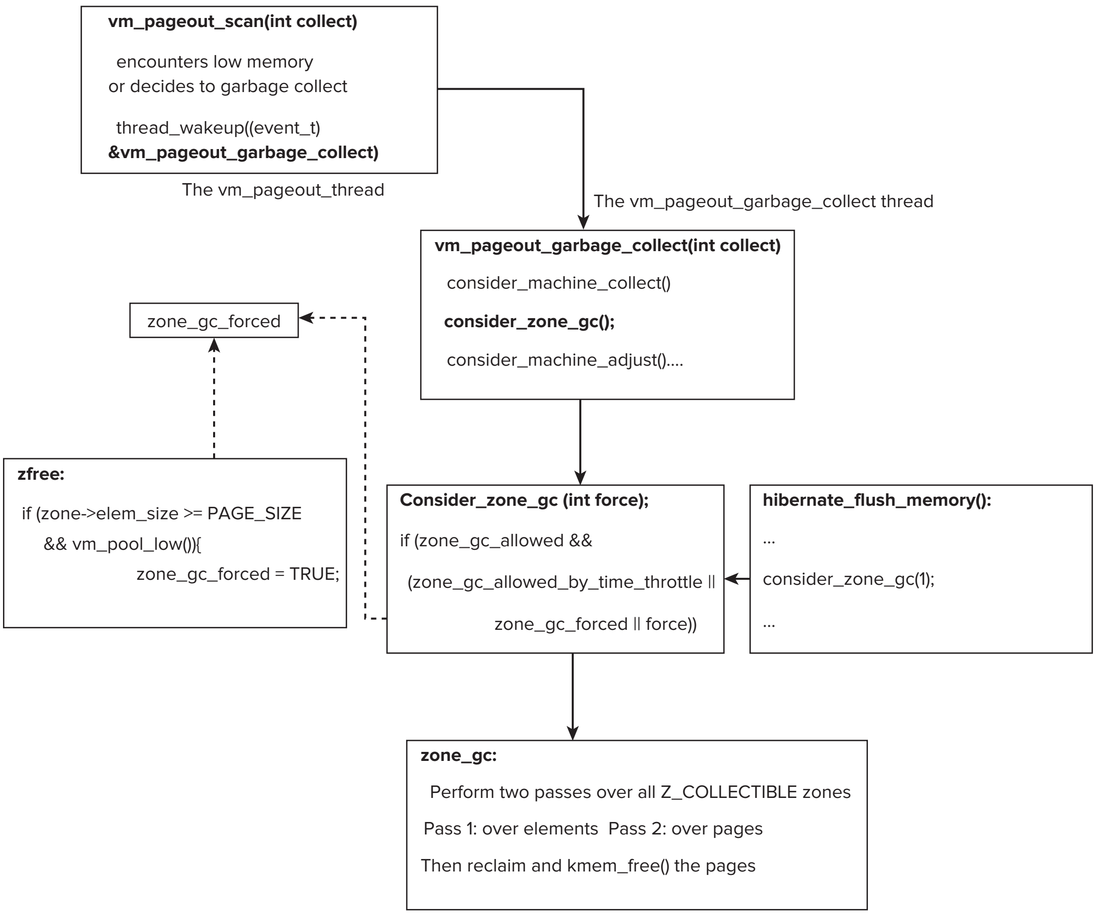


FIGURE 12-3 Zone garbage collection

The garbage collection is a two-pass process, wherein the system first goes over all zones (skipping over zones marked as non-collectable), examining their free lists and seeing which objects can be claimed. On the second pass, the objects are translated into pages: Objects that share a page with non-freed objects are of no use to the system, as only full pages can be freed. Finally, when the pages to be freed are determined, they can be freed by a simple `kmem_free()`.

Zone Debugging

In the unlikely case you will ever need to, it is possible to debug zones — past the simple functionality provided `zprint(1)` command — in several ways:

- **Compile with `CONFIG_ZLEAKS`:** This, as you saw, allocates more data per struct zone to check on memory leaks. `CONFIG_ZLEAKS` also makes `zleaks` toggleable from the BSD layer and user mode by means of `sysctl(8)` calls on the `kern.zleaks` (as defined in `bsd/kern/kern_malloc.c`).
- **Toggle zone element checking:** with the `-zc` boot argument
- **Toggle zone poisoning:** with the `-zp` boot argument
- **Save zone info in each task:** with the `-zinfop` boot argument
- **Specific zone logging boot arguments:** by using `zlog` you can specify the exact name of a zone to log, and with `zrecs` you can specify how many records will be kept in the log (up to 8000).

KERNEL MEMORY ALLOCATORS

The VM abstractions detailed thus far are important, yet when kernel code needs to allocate memory, especially within its own `vm_map` (that is, the `kernel_map`), it needs to rely on actual allocator functions, that can allocate the virtual memory as well as back it up with physical pages. This section covers the rich hierarchy of allocators in XNU (with one exception, BSD's cache and slab allocators), shown in Figure 12-4:

kernel_memory_allocate()

All kernel memory allocation paths (save contiguous physical memory), sooner or later, end up using a single function, `kernel_memory_allocate()`. This function, defined in `osfmk/vm/vm_kern.c`, performs the actual allocation of memory, handling both the `vm_map` and the pmap. It is shown in Listing 12-8:

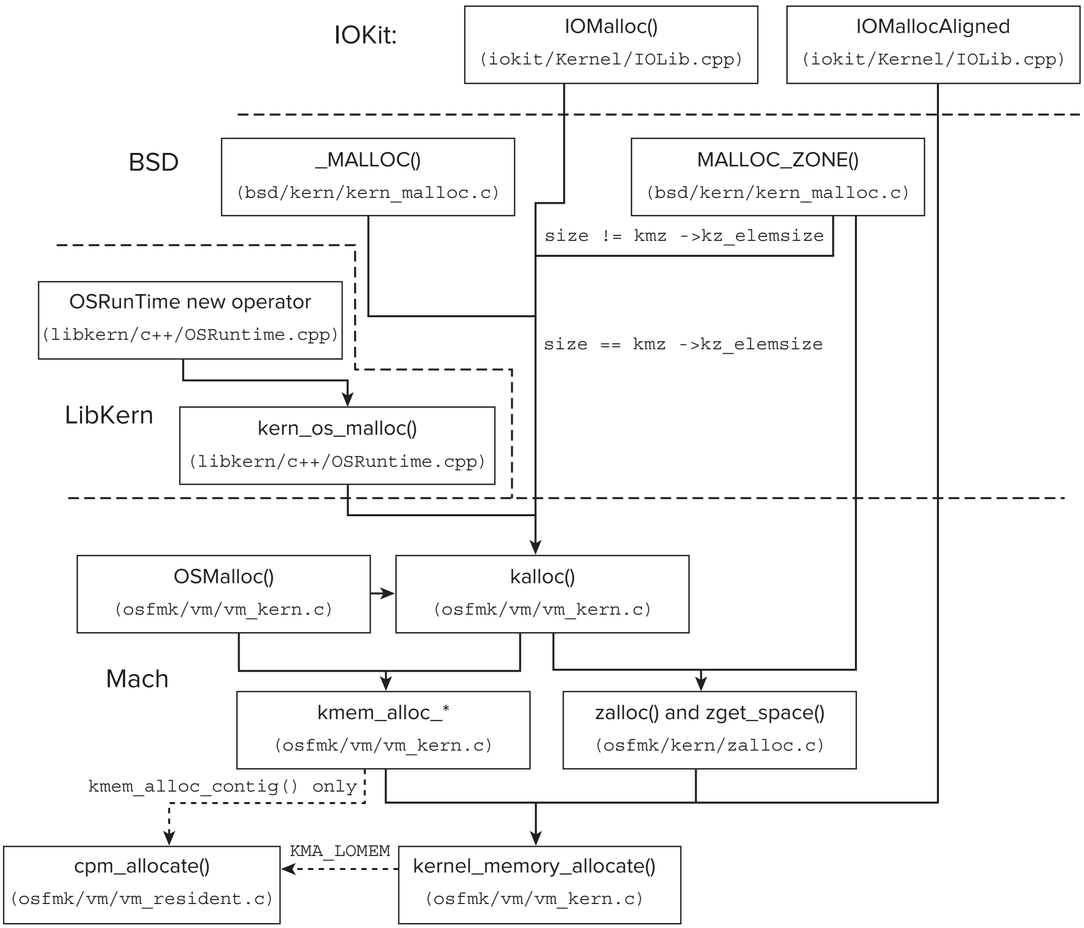


FIGURE 12-4: The XNU memory allocator hierarchy

LISTING 12-8: kernel_memory_allocate(), from osfmk/vm/vm_kern

```
/*
 * Master entry point for allocating kernel memory.
 * NOTE: this routine is _never_ interrupt safe.
 *
 * map          : map to allocate into
 * addrp        : pointer to start address of new memory
 * size         : size of memory requested
```

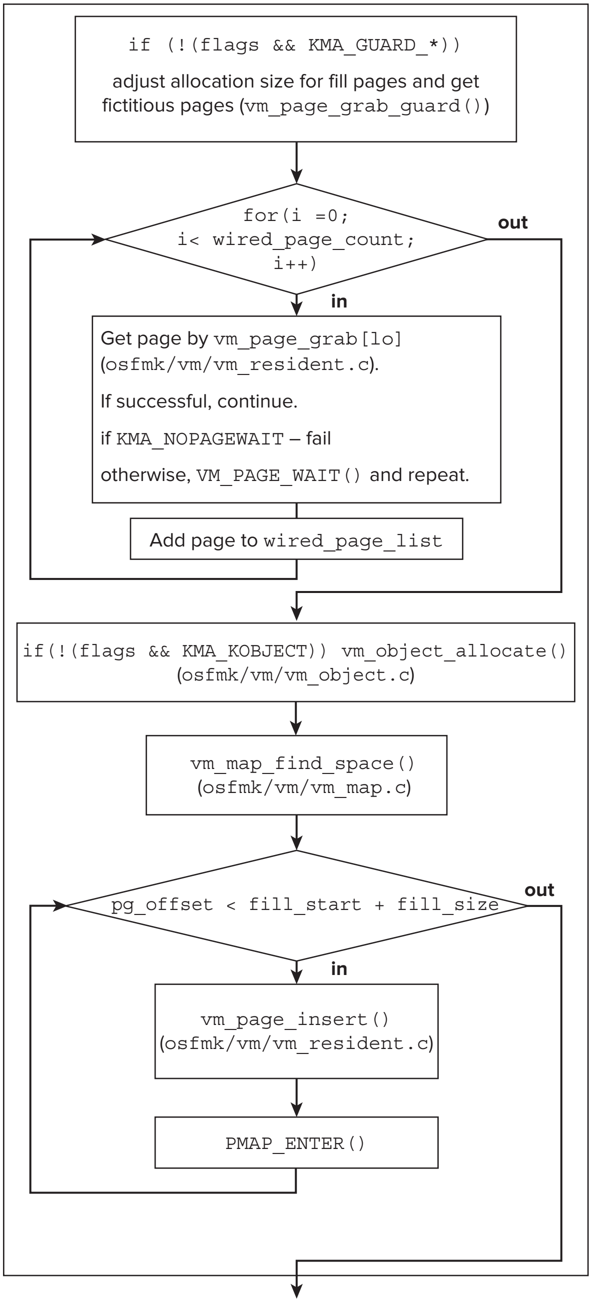
```
 * flags        : options
 *                KMA_HERE             *addrp is base address, else "anywhere"
 *                KMA_NOPAGEWAIT       don't wait for pages if unavailable
 //                                    (returns KERN_RESOURCE_SHORTAGE instead)
 *                KMA_KOBJECT          use kernel_object
 *                KMA_LOMEM            support for 32 bit devices in a 64 bit world
 *                                     if set and a lomemory pool is available
 *                                     grab pages from it... this also implies
 *                                     KMA_NOPAGEWAIT
 //   And also:
 //               KMA_NOENCRYPT         Do not encrypt the pages (calls
 //               pmap_set_noencrypt())
 //               KMA_GUARD_[FIRST|LAST] Place guard pages before or after the
 //               allocation
 */

kern_return_t
kernel_memory_allocate(
        register vm_map_t       map,
        register vm_offset_t    *addrp,
        register vm_size_t      size,
        register vm_offset_t    mask,
        int                     flags);
```

This function finds a large enough virtual address space in the `vm_map` it is handed, and takes memory from the wired list to satisfy the allocation. In some cases (specifically, calls from `stack_alloc()`), flags to `kernel_memory_allocate()` may specify a request for guard pages — before or after the actual allocation. These are similar in principle to those of user mode's `libgmalloc.dylib` — and are virtual-only pages marked non-accessible, so as to trigger a page fault on access. Getting guard pages therefore only requires space in the `vm_map`, but no physical backing (and hence no pmem).

A simplified flow of `kernel_memory_allocate()` is shown in Figure 12-5:

The actual allocation of the physical page is done by looking at one of two free lists: the per-processor free list (using `vm_page_grab()`, which uses the `PROCESSOR_DATA` macro to get a page from `free_pages` list), or the low memory free list (using `vm_page_grablo()`, which queries the `vm_lopage_queue_free` list). The latter case is rarely encountered, only when specific physical memory regions (less than 16MB) are required. The `vm_page_grablo()` function calls on `cpm_allocate()`, which is used to allocate contiguous physical memory by stealing pages directly from the free list. The `cpm_allocate()` function (from `osfmk/vm/vm_resident.c`) is rarely called on: It is otherwise only called from `kmem_alloc_contig()`, `vm_map_enter()` (for superpages) or `vm_map_enter_cpm()`.

The `kernel_memory_allocate()` function is also seldom called directly. Exceptions include early startup (when there is little choice), kernel stack allocations, and IOKit's `IOMallocAligned()`, which requires specific aligned memory. In all other cases, wrappers are used, the most significant of which is `kmem_alloc()`.



Stack allocations may request additional guard pages before or after the allocations.These are fictitious pages (i.e. only PTEs) and require no physical backing – only virtual space.

Grab pages one by one, and link to `wired_page_list`, until `wired_page_count` is satisfied. Pages are grabbed from per CPU free list (`vm_page_grab`) or global low page queue (`vm_page_grab_lo`) if `KMA_LOMEM` was requested.

If unsuccessful, this can block indefinitely (using a call to `vm_page_wait(THREAD_UNINT)`, until the page is obtained).

If `KMA_NOPAGEWAIT` was specified, the function will not block, and fails with `KERN_RESOURCE_SHORTAGE` immediately.

Call `vm_object_allocate()` to alloc a new object, unless we can use the `kernel_object`.

Find space to insert all the pages in target's `vm_map`.

While pages added have not satisfied, and we have remaining pages in wired_page_list: insert them one by one to the target `vm_map` and the `kernel_pmap` (using the `PMAP_ENTER` macro). If we run out of pages, panic.

A similar loop also handles the insertion of the guard pages (but does not call `PMAP_ENTER` for them, as they have no physical backing).

FIGURE 12-5: Simplified flow of kernel_memory_allocate()

kmem_alloc() and Friends

The most common memory allocator in Mach is provided by the `kmem_alloc()` family of functions in `osfmk/kern/vm_kern.c`, which wrap `kernel_memory_allocate()`, as shown in Figure 12-6.

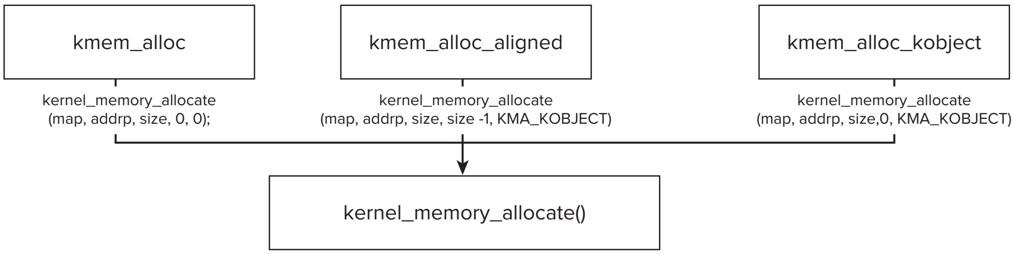


FIGURE 12-6: The Kmem_malloc family of functions.

All the `kmem_alloc` types shown in Figure 12-6 share the same prototype, taking as their three arguments a map, an in/out address pointer, and a size argument. The map argument in these functions is commonly the `kernel_map vm_map`, unless pageable memory is requested. As shown in the figure, these functions are layered on top of `kernel_memory_allocate()`, discussed previously.

Other `kmem_alloc_*` functions exist, which are not implemented over `kernel_memory_allocate()`. These functions are:

- `kmem_alloc_contig()` — for contiguous physical memory (implemented over `cpm_allocate()`).
- `kmem_alloc_pageable()` (allocated over `vm_map_enter()`), which allocates non-wired memory. Non-wired memory, however, may be paged out without warning.
- `kmem_alloc_pages()` can be used to allocate new pages in an existing object, and wraps `vm_page_alloc()` (which itself is just a wrapper over the `vm_page_grab()`/`vm_page_insert()` of `kernel_memory_allocate()`.

Using `kmem_alloc()` is quite expensive, particularly due to physical map backing: Recall, the underlying implementation of `kernel_memory_allocate()` may block indefinitely. More often, then, the faster `kalloc()` alternative (built over the more efficient mechanism of zones) is used.

kalloc

Once Mach zones are initialized, they may be used for quick kernel internal allocations, as is provided by the `kalloc_()` family of functions. These functions are all defined in `osfmk/kern/kalloc.h` as shown in Listing 12-9.

LISTING 12-9: Some of the kalloc functions in osfmk/kern/kalloc.h

```
extern void *kalloc(vm_size_t   size);
extern void *kalloc_noblock(vm_size_t   size);
extern void  kfree(void         *data,                vm_size_t     size);
```

These functions are functionally equivalent to user-mode `malloc()` and `free()`, but utilize zones and can thus offer nonblocking functionality, as in the `kalloc_noblock()` function. Because the zone memory is pre-allocated, `kalloc()` allocation is simply a call through to `zalloc_canblock()` on the corresponding zone (one of the `kalloc.nn` zones, shown in Table 12-4). The zones themselves are set up by `kalloc_init()`, which is called from `vm_mem_bootstrap()` during system startup (as shown in Chapter 6). If `kalloc()` is called with a size larger than the maximum zone, it calls `kmem_alloc()` instead (and must block). Likewise, if `kfree()` detects the size of the block freed does not match one of the zones, it calls `kmem_free(`instead of `zfree()`). The `kalloc()` function keeps track of the largest block size it is required to allocate in a global, and `kfree()` ignores attempts to free blocks larger than that size. Internally, a `krealloc()` function is defined as well, but neither it nor a `kget()` function is used.

Overall, this mechanism is quite similar to Linux's `kmalloc()`, which also allocates memory in a fast, potentially non-blocking manner. Also like it, `kalloc()` sizes are rounded to the nearest power of two, which can be quite wasteful (for example, 4,098 bytes actually consume 8,192 bytes).

In iOS 5, `kalloc` zones are also available in sizes which are not powers of 2. Listing 12-10 shows the output of `zprint` from an iOS 5.0 host:

LISTING 12-10: kalloc zones. The bold zones are iOS specific

```
root@podicum (/)# zprint kalloc
zone name                 size   size   size  #elts  #elts inuse  size count
------------------------------------------------------------------------------
kalloc.8                     8    60K    60K   7680   7776  7392    4K   512 C
kalloc.16                   16    88K   121K   5632   7776  5332    4K   256 C
kalloc.24                   24   334K   410K  14280  17496 14034    4K   170 C
kalloc.32                   32   124K   128K   3968   4096  3541    4K   128 C
kalloc.40                   40   255K   360K   6528   9216  6374    4K   102 C
kalloc.48                   48    87K   192K   1870   4096  1408    4K    85 C
kalloc.64                   64   120K   256K   1920   4096  1612    4K    64 C
kalloc.88                   88   229K   352K   2668   4096  2382    4K    46 C
kalloc.112                 112   118K   448K   1080   4096   884    4K    36 C
kalloc.128                 128   168K   512K   1344   4096  1133    4K    32 C
kalloc.192                 192    94K   768K    504   4096   454    4K    21 C
kalloc.256                 256   168K  1024K    672   4096   580    4K    16 C
kalloc.384                 384   551K  1536K   1470   4096  1253    4K    10 C
kalloc.512                 512    40K   512K     80   1024    42    4K     8 C
kalloc.768                 768    82K   768K    110   1024   101    4K     5 C
kalloc.1024               1024   104K  1024K    104   1024    79    4K     4 C
kalloc.1536               1536    99K  1536K     66   1024    55   12K     8 C
kalloc.2048               2048    84K  2048K     42   1024    41    4K     2 C
kalloc.3072               3072    72K  3072K     24   1024    18   12K     4 C
```

```
kalloc.4096              4096   136K  4096K     34   1024     32    4K     1 C
kalloc.6144              6144   258K   576K     43     96     41   12K     2 C
kalloc.8192              8192   144K 32768K     18   4096     16    8K     1 C
kalloc.large            59163  2657K  2906K     46     50     46   57K     1
```

The `kalloc` function is the most widely used memory allocator in XNU, with many wrappers, including:

- **IOKit's `IOMalloc` (`iokit/Kernel/IOLib.cpp`)**: Directly wrapping `kalloc()` but also adding a call to IOStatisticsAlloc macro, which records the allocations (for `ioalloccount(8)`, as discussed in chapter 18)
- **Libkern's `kern_os_malloc`(`libkern/c++/OSRuntime.cpp`)**: A direct wrapper over `kalloc()`, which prepends the block size to the allocation. This function is itself wrapped by the `new` operator.
- **BSD's `_MALLOC` (`bsd/kern/kern_malloc.c`):** used for various allocations in the BSD layer, discussed in Chapter 13. Similar to `kern_os_malloc()`, it also prepends the block size to the allocation.

OSMalloc

Mach exports yet another family of memory allocation functions, `OSMalloc`. The `OSMalloc` sorority, though implemented alongside `kalloc` in `osfmk/kern/kalloc.c`, is actually defined in `libkern/libkern/OSMalloc.h` as shown in Listing 12-11.

LISTING 12-11: OSMalloc functions, as defined in libkern/libkern/OSMalloc.h

```
typedef struct __OSMallocTag__ * OSMallocTag;

// First get a tag — this actually uses kalloc()
extern OSMallocTag OSMalloc_Tagalloc(const char * name,
                                     uint32_t    flags);
// Then allocate with it:
extern void * OSMalloc(uint32_t    size, OSMallocTag tag);
// The following two are equivalent:
extern void * OSMalloc_nowait(uint32_t    size,  OSMallocTag tag);
extern void * OSMalloc_noblock (uint32_t    size, OSMallocTag tag);
// Freeing memory requires the tag, as well:
extern void OSFree(void      * addr, uint32_t    size, OSMallocTag tag);
// Finally, free tag
extern void OSMalloc_Tagfree(OSMallocTag tag);
```

The key concept in `OSMalloc` is that of the *tag*, an opaque type, which must be allocated first. Once the caller is in possession of the tag, it can be passed to one of the `OSMalloc` functions (either the blocking or non-blocking varieties) to allocate the memory. The memory can be freed (using `OSFree()`), and when the tag is no longer required, it, too, can be freed. The `OSMalloc` memory is allocated with `kmem_alloc_pageable`, if the tag flags allow it (specifying `OSMT_PAGEABLE`). Otherwise, it is allocated with `kalloc()`, from wired memory. Alternatively, the `noblock/nowait` functions (which are functionally equivalent) call on `kalloc_noblock()` for wired memory.

The tag itself is part of a linked list of tags, each with a reference count. Allocations increment the reference count of the tag. Listing 12-12 shows the structure of a tag.

LISTING 12-12: OSMalloc tags

```
typedef struct _OSMallocTag_ {
    queue_chain_t   OSMT_link;
    uint32_t        OSMT_refcnt;
    uint32_t        OSMT_state;
    uint32_t        OSMT_attr;
    char            OSMT_name[OSMT_MAX_NAME];
} * OSMallocTag;
```

MACH PAGERS

Sooner or later, it happens to the best: The memory requirements of processes exceed the available amount of RAM, and the system has to find a way to back up inactive pages and remove them from RAM, at least temporarily, to make more RAM available for active ones.

In other operating systems, this is the role of dedicated kernel threads. Linux, for example, has `pdflush` and `kswapd`. In Mach, these dedicated tasks are called pagers, and may be in-kernel threads, or even external user mode (or remote) servers.

A Mach pager is a memory manager, charged with the task of backing up virtual memory to a backing store of a particular type. The backing store holds the content of the memory pages when they need to be swapped out, due to insufficient RAM, and recovered, when RAM becomes available again. This is required only for these pages which are "dirty," i.e. have changed in RAM, and therefore must be saved to prevent data loss.

Note, that the pagers listed here merely implement the paging operation of the memory objects they are tied to. They do not manage or control the system's paging policy. Doing so is the role of the `vm_pageout` daemon, which is the role that `kernel_bootstrap_thread()` assumes once it completes (as discussed in Chapter 8). The `vm_pageout` daemon is discussed in more detail at the end of this chapter.

The Mach Pager interface

Although there are several types of pagers, all present the same interface to the kernel. The pagers all expose particular routines, and perform operations on memory objects. Mach's original design treated pagers as fully external entities, and defined the External Memory Manager Interface (EMMI), to specify the types of Mach messages pagers use to communicate with the kernel. The MIG specifications for pagers can still be found in `osfmk/mach`, as shown in Table 12-5:

TABLE 12-5: MIG Files in osfmk/mach Specifying Mach Pager Interfaces

FILE	SPECIFIES
`memory_object.defs`	Subsystem 2200, specifying initialization, termination and the core routines involved in the object lifecycle, all of which operate on a `memory_object_t`.
`memory_object_control.defs`	Subsystem 2000, specifying additional memory object operations, operating on a `memory_object_control_t` argument.
`memory_object_default.defs`	Subsystem 2250, consisting of a single routine, `memory_object_create()`, which is used to construct a new memory object.
`memory_object_name.defs`	Unused.

In practice, however, you have seen that XNU takes significant shortcuts and deviations from the microkernel design of Mach, in order to achieve greater efficiency. The pagers in XNU are therefore implemented in-kernel, and instead of over messages, the pager interface is implemented as function calls. Much like the Mach thread schedulers, the Mach pagers are defined as objects and implement a set of well-known methods, or operations. These operations correspond to the MIG routines in `memory_object.defs`, and are defined in `osfmk/mach/memory_object_types.h` in a struct `memory_object_pager_ops` as shown in Table 12-6.

TABLE 12-6: Pager Operations

PAGER METHOD	USED FOR
`memory_object_reference` `(memory_object_t mem_obj)`	Marks `mem_obj` as referenced. This is required for the LRU of the `vm_pageout` daemon, discussed later.
`memory_object_deallocate` `(memory_object_t mem_obj)`	Deallocates the memory object `mem_obj`.
`memory_object_init` `(memory_object_t mem_obj, memory_object_control_t mem_control, memory_object_cluster_size_t size))`	Initializes a new memory object of *`size`* bytes, with `mem_control` data. The pager is expected to set the object's IPC class (`IKOT_MEMORY_OBJECT`) and tie its operations to it (as function pointers).
`memory_object_terminate` `(memory_object_t mem_obj);`	Terminates (destroys) memory object `mem_obj`.

continues

TABLE 12-6 *(continued)*

PAGER METHOD	USED FOR
`memory_object_data_request` `(memory_object_t mem_obj, memory_object_offset_t offset, memory_object_cluster_size_t length, vm_prot_t desired_access, memory_object_fault_info_t fault_info);`	Handles a page-in request (a request for `mem_obj` at address offset of *`length`* bytes). The kernel is requesting the pager to provide a page from the backing store.
`memory_object_data_return` `(memory_object_t mem_obj, memory_object_offset_t offset, memory_object_cluster_size_t size, memory_object_offset_t *resid_offset, int *io_error, boolean_t dirty, boolean_t kernel_copy, int upl_flags);`	Handles a page-out request (a request for `mem_obj` at address offset of *`length`* bytes). The kernel is "returning" the dirty page to the pager, which is expected to commit it to the backing store.
`memory_object_data_initialize` `(memory_object_t mem_obj, memory_object_offset_t offset, memory_object_cluster_size_t size);`	Similar to `data_return`, but allows initialization of `mem_obj`. In practice, unimplemented in pagers (results in panic).
`memory_object_data_unlock` `(memory_object_t mem_obj, memory_object_offset_t offset, memory_object_size_t size, vm_prot_t desired_access);`	Change permissions on `mem_obj` to *`desired_access`*.
`memory_object_synchronize` `(memory_object_t mem_obj, memory_object_offset_t offset, memory_object_size_t size, vm_sync_t sync_flags);`	Synchronize `mem_obj` to backing store according to sync_flags (equivalent to flushing a page).
`memory_object_map``( memory_object_t mem_obj, vm_prot_t prot);`	Map pages in the `mem_obj` with the protections specified.
`memory_object_last_unmap` `(memory_object_t mem_obj);`	Called when the last mapping of `mem_obj` is removed.
`memory_object_data_reclaim` `(memory_object_t mem_obj, boolean_t reclaim);`	Request pager to reclaim page. In practice, left NULL by most pagers.

In the preceding table, the two most important operations are `data_request` (for swap in) and `data_return` (for swap out). A pager does not have to implement all the methods listed in the table. In fact, some memory managers panic if certain methods are called.

Additional memory object operations are defined on an opaque `memory_object_control_t` type. These include getting/changing attributes, locking, and UPL related requests (more on UPLs later). Both types, the `memory_object_t` and the `memory_object_control_t`, are defined in `osfmk/mach/memory_objects_types.h`, as shown in Listing 12-13:

LISTING 12-13: Memory objects, as defined in osfmk/memory_object_types.h

```
/*
 * Temporary until real EMMI version gets re-implemented
 */

#ifdef  KERNEL_PRIVATE

struct memory_object_pager_ops; /* forward declaration */

typedef struct          memory_object {
        unsigned int    _pad1; /* struct ipc_object_header */
#ifdef __LP64__
        unsigned int    _pad2; /* pad to natural boundary */
#endif
        const struct memory_object_pager_ops    *mo_pager_ops;
} *memory_object_t;

typedef struct          memory_object_control {
        unsigned int    moc_ikot;  /* struct ipc_object_header. Must be
                                   /* IKOT_MEM_OBJ_CONTROL */
#ifdef __LP64__
        unsigned int    _pad; /* pad to natural boundary */
#endif
        struct vm_object *moc_object;
} *memory_object_control_t;
```

As an old adage goes, the most permanent things in life start out as "temporary," and so, apparently, is the implementation of memory objects: Operations on a `memory_object_t` in Table 12-6 are redirected to the implementing pager (via the `mo_pager_ops` field of the structure). Other operations, which require a `memory_object_control_t` argument, convert their argument into a struct `vm_object` (described earlier in this chapter), by means of a `memory_object_control_to_vm_object()` call, which really just returns the `moc_object` field of the control structure.

The different pagers implement their own memory objects by extending the memory object. Their pager object implementations must align with the `memory_object_t`, but the implementation is free to add more fields, as shown in Figure 12-7

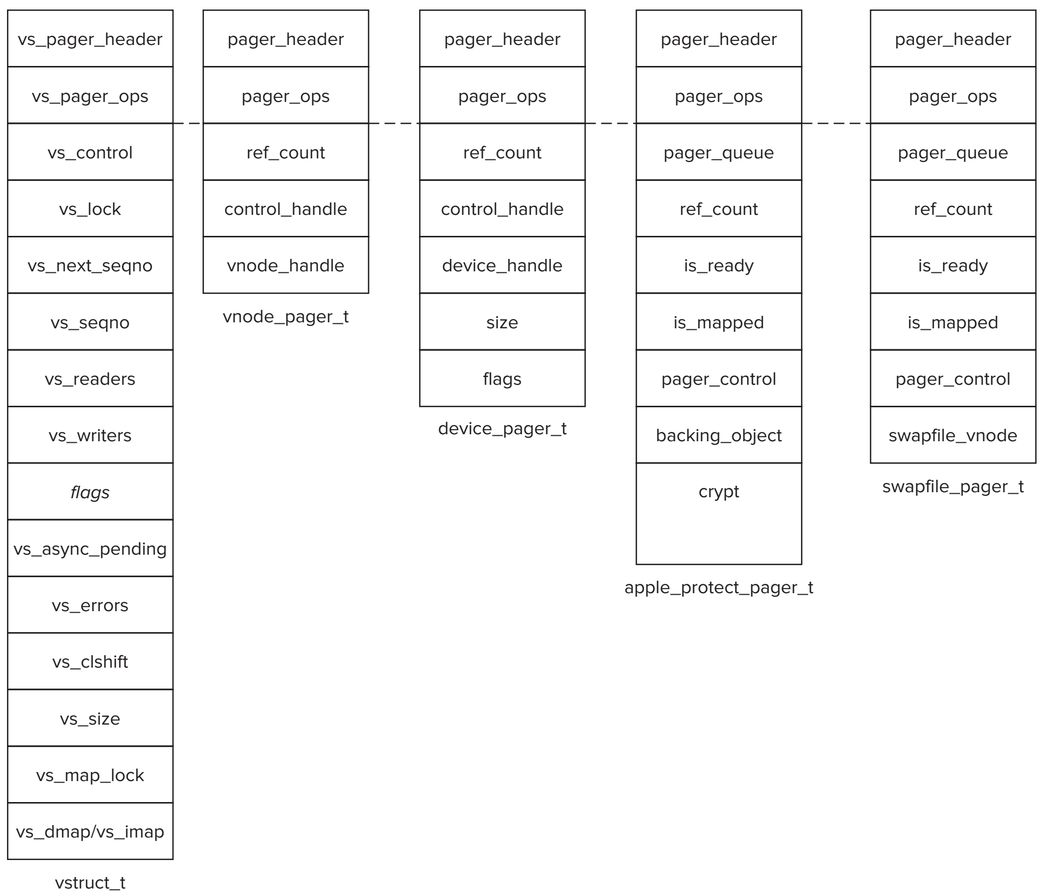


FIGURE 12-7

These pagers are all discussed shortly, but before we can turn to them, we must first consider another important data structure required for paging — the Universal Page List.

Universal Page Lists

Mach uses the Universal Page List (UPL) structure to maintain information about pages in implementation-agnostic lists. The "Universal" term implies the pages can be backed on any backing store type. The UPL structure is generally hidden from most other kernel components, with the exception of the pagers (primarily, the page out daemon) and some BSD components (notably, filesystems and the Unified Buffer Cache). It is defined as shown in Listing 12-14.

LISTING 12-14: The Universal Page List

```
struct upl {
        decl_lck_mtx_data(,     Lock)   /* Synchronization */
        int             ref_count;
        int             ext_ref_count;
        int             flags;
        vm_object_t     src_object; /* object derived from */
        vm_object_offset_t offset;
        upl_size_t      size;       /* size in bytes of the address space */
        vm_offset_t     kaddr;      /* secondary mapping in kernel */
        vm_object_t     map_object;
        ppnum_t         highest_page;
        void*           vector_upl;
#if     UPL_DEBUG
        uintptr_t       ubc_alias1;
        uintptr_t       ubc_alias2;
        queue_chain_t   uplq;       /* List of outstanding upls on an obj */

        thread_t        upl_creator;
        uint32_t        upl_state;
        uint32_t        upl_commit_index;
        void    *upl_create_retaddr[UPL_DEBUG_STACK_FRAMES];

        struct  ucd     upl_commit_records[UPL_DEBUG_COMMIT_RECORDS];
#endif  /* UPL_DEBUG */
};
```

The UPL serves to link the virtual addresses with the actual physical pages, somewhat like a Windows Memory Descriptor List (MDL), or IOKit's IOMemoryDescriptor. The corresponding physical page properties are recorded in the UPL. This API is not used directly, passing through several layers of abstraction, even for the few components, which are UPL-aware.

The MIG file `osfmk/mach/upl.defs` contains the definitions of some UPL operations. All the operations are implemented in `osfmk/vm/vm_pageout.c`, and shown in Table 12-7:

TABLE 12-7: UPL Operations

OPERATION	USED TO
`upl_create (int type, int flags, upl_size_t size);`	Create a new UPL. Usually wrapped by other functions.

continues

TABLE 12-7 *(continued)*

OPERATION	USED TO
`upl_deallocate``(upl_t upl);` `upl_destroy(upl_t upl);`	Decrement reference count of a UPL, destroying if count drops to 0.
`upl_clear_dirty``(upl_t upl,` `boolean_t value)`	Explicitly mark the UPL clear or dirty (according to *`value`*). Used by Apple Protect pager to prevent swap out of pages
`upl_abort[range]` `(upl_t upl,` `upl_offset_t offset,` `upl_size_t size,` `int error,` `boolean_t *empty);` **`upl_commit[_range]`** `(upl_t upl,` `upl_offset_t offset,` `upl_size_t size,` `int flags,` `upl_page_info_t *page_list,` `mach_msg_type_number_t cnt,` `boolean_t *empty);`	Abort or commit changes to a UPL or part thereof, from offset to size bytes (rounded to nearest page). The `upl_abort()` and `upl_commit` are wrappers over their *`corresponding _range`* counterparts, specifying an offset of 0 and a size of `upl->size`.

Pager Types

XNU contains the same pagers in iOS and OS X (this includes the swapfile pager, even though iOS has no real swap to speak of). iOS also contains an experimental new pager, called the Default Freezer. These pagers are shown in Table 12-8.

TABLE 12-8: Memory Pagers in XNU

MEMORY PAGER	DEFINED IN	USED FOR
Default pager	`default_pager/*`	Anonymous memory
VNode Pager	…`/bsd_vm.c`	Memory mapped files
Device pager	…`/device_vm.c`	Device backed I/O
Swapfile pager	…`/vm_swapfile_pager.c`	Handles specific swapfile mapping attempts to prevent reading swap file data by memory mappings
Apple-protected pager	…`/vm_apple_protect`	Apple-specific extension; Provides support for memory (and specifically, binary) encryption

MEMORY PAGER	DEFINED IN	USED FOR
Freezer (iOS, found in Lion kernel sources, but not enabled by default)	.../default_freezer.c	iOS specific extension to support "freezing" processes.

Although Mach allows for pagers to be defined externally using the EMMI, these pagers are all in-kernel threads.

The Default Pager

The default pager is, as its name implies, the basic pager in Mach and XNU. It is defined in `osfmk/default_pager/` in the following files, shown in Table 12-9:

TABLE 12-9: Default Pager Files

FILE	SPECIFIES
`default_pager.c`	Implementation
`default_pager_internal.h`	Data structures
`diag.h`	Diagnostics (statistics) lock
`default_pager_alerts.defs`	MIG subsystem 2295: containing one message (`default_pager_space_alert`) used to notify of high and low water mark events
`default_pager_object.defs`	MIG subsystem 2275: messages used to communicate with default server
`default_pager_types.defs`	Data types used in other MIG files
`dp_backing_store.c`	Backing store support
`dp_memory_object.c`	Implementation of default pager's operations

The default pager is started by one of two Mach traps (`macx_swapon()` or `macx_triggers()`, both discussed later). If either trap detects that the pager is not initialized (i.e. `default_pager_init_flag` is zero), it calls on `start_def_pager()`, which calls on `default_pager_initialize()` (both in `osfmk/default_pager/default_pager.c`).

When the default pager initializes, it creates a `vstruct_zone` for its pager objects, and registers a Mach port using `host_default_memory_manager()` (defined in `osfmk/vm/memory_object.c`). Clients wishing to communicate with it can call the same function to obtain its ports, and send it one of the messages (defined in `default_pager_objects.defs`). The port can also be obtained from user mode (via same Mach message, on the host's privileged port). The pager itself maintains communication with the `dynamic_pager(8)` (discussed towards the end of this chapter), a user mode accomplice which handles adding, deleting and adjusting swap files. This user mode daemon, however, communicates back with the `default_pager` using dedicated Mach traps, rather than messaging.

Although the default pager port is accessible from user mode, in most cases it is not meant to be used directly. Its only official user mode client is the `dynamic_pager(8)`. For those clients wishing to request information, the information message `default_pager_info_64` was wrapped by the `macx_swapinfo()` Mach trap. This trap, though, has since been wrapped as well, by the `sysctl(2)` interface and `kern.swapusage` MIB.

As a side effect of the port registration, a new kernel thread, `vm_pageout_iothread_internal`, is started by a call to `vm_pageout_internal_start()`. This is a dedicated thread which is used to page out `vm_objects` that are used internally by the kernel (discussed in the next section, under "The Pageout Daemon").

The Vnode Pager

The vnode pager is responsible for supporting the memory mapping of files. When files are memory mapped, their contents need to be read from the file system. When the memory mapped files are dirtied in memory, they need to be written back to the file system. The pager is implemented in `osfmk/vm/bsd_vm.c`.

When a vnode is created (using `vnode_create()`, as discussed in Chapter 15, "Files and Filesystems"), VFS calls on the Unified Buffer Cache `ubc_info_init()` function to handle the buffering required for the file's contents. This method, in turn, calls `vnode_pager_setup ()`, which simply calls `vnode_object_create()` to create a new pager memory object, and tie the supplied vnode handle to it. The vnode pager's `data_request` and `data_return` methods respectively wrap `vnode_pagein()` and `vnode_pageout()`.

The Device Pager

The device pager is responsible for supporting the memory mapping of devices. It is similar in concept to the vnode pager, but is closely integrated with IOKit. The `device_pager_setup()` (called from IOKit's `IOGeneralMemoryDescriptor::doMap()`) creates a new pager memory object, and ties the supplied device handle to it. The device pager's `data_request` and `data_return` methods then call `device_data_action()` (again implemented in IOKit's `iokit/Kernel/IOMemoryDescriptor.cpp`) to read or write data, respectively from or to the device. Similarly, `IOMemoryDescriptor::handleFault()` calls back on `device_pager_populate_object()`.

The Swapfile Pager

The swapfile pager's name is misleading — this is not the pager charged with swapping (the default pager is). In fact, it is meant to discourage attempts to directly map the swap file. If a user process does try to map a swap file, the mapping is associated with the swapfile pager, rather than the default, as shown in Listing 12-15:

LISTING 12-15: Redirection of swap mmap(2) requests, from bsd/kern/kern_mman.c:

```
int mmap(proc_t p, struct mmap_args *uap, user_addr_t *retval)
{
   struct fileproc *fp;
   register struct         vnode *vp;
   // ...
   int fd = uap->fd;
   // ...
   err = fp_lookup(p, fd, &fp, 0);
   // ...
   vp = (struct vnode *)fp->f_fglob->fg_data;
   // ...
            if (vnode_isswap(vp)) {
                        /*
                         * Map swap files with a special pager
                         * that returns obfuscated contents.
                         */
                        control = NULL;
                        pager = swapfile_pager_setup(vp);
                        if (pager != MEMORY_OBJECT_NULL) {
                                control = swapfile_pager_control(pager);
                        }
           ...
}
```

The swapfile pager implements the `swapfile_pager_data_request()` method, which just returns zeroed pages (by explicitly `memset()` using), as Listing 12-16 shows:

LISTING 12-16: The implementation of the swapfile pager's data request (osfmk/vm/vm_swapfile_pager.c)

```
kern_return_t
swapfile_pager_data_request(
        memory_object_t         mem_obj,
        memory_object_offset_t  offset,
        memory_object_cluster_size_t            length,
#if !DEBUG
        __unused
#endif
        vm_prot_t               protection_required,
        __unused memory_object_fault_info_t mo_fault_info)
{
            //...

        /*
         * Reserve a virtual page in the kernel address space to map each
         * destination physical page when it's its turn to be processed.
         */
        vm_object_reference(kernel_object);     /* ref. for mapping */
```

continues

LISTING 12-16 *(continued)*

```
        kr = vm_map_find_space(kernel_map,
                               &kernel_mapping,
                               PAGE_SIZE_64,
                               0,
                               0,
                               &map_entry);
        // ...
        dst_vaddr = CAST_DOWN(vm_offset_t, kernel_mapping);
        dst_ptr = (char *) dst_vaddr;
        /*
         * Gather in a UPL all the VM pages requested by VM.
         */
        mo_control = pager->pager_control;

        upl_size = length;
        upl_flags =
                UPL_RET_ONLY_ABSENT |
                UPL_SET_LITE |
                UPL_NO_SYNC |
                UPL_CLEAN_IN_PLACE |    /* triggers UPL_CLEAR_DIRTY */
                UPL_SET_INTERNAL;
        pl_count = 0;
        kr = memory_object_upl_request(mo_control,
                                       offset, upl_size,
                                       &upl, NULL, NULL, upl_flags);
        // ...
        /*
         * Fill in the contents of the pages requested by VM.
         */
        upl_pl = UPL_GET_INTERNAL_PAGE_LIST(upl);
        pl_count = length / PAGE_SIZE;
        for (cur_offset = 0; cur_offset < length; cur_offset += PAGE_SIZE) {
                ppnum_t dst_pnum;

                if (!upl_page_present(upl_pl, (int)(cur_offset / PAGE_SIZE))) {
                        /* this page is not in the UPL: skip it */
                        continue;
                }

                /*
                 * Establish an explicit pmap mapping of the destination
                 * physical page.
                 * We can't do a regular VM mapping because the VM page
                 * is "busy".
                 */
                dst_pnum = (ppnum_t)
                        upl_phys_page(upl_pl, (int)(cur_offset / PAGE_SIZE));
                assert(dst_pnum != 0);
                pmap_enter(kernel_pmap,
                           kernel_mapping,
                           dst_pnum,
                           VM_PROT_READ | VM_PROT_WRITE,
                           0,
```

```
                        TRUE);

                memset(dst_ptr, '\0', PAGE_SIZE);  // explicit zeroing of pages
                /* add an end-of-line to keep line counters happy */
                dst_ptr[PAGE_SIZE-1] = '\n';


}
```

The pager cannot handle page-out requests, and will panic if its `data_return` function is called.

The Apple Protect Pager

A specific external memory manager of great importance is the Apple Protect pager. This is the memory pager responsible for implementing Apple's code encryption mechanism. This pager is somewhat similar to the swapfile pager (having likely been copied from it), but instead of zeroed out pages, it returns pages after invoking a decryption function on them. The pager contains an additional field, a `pager_crypt_info` structure, defined in `<osfmk/kern/page_decrypt.h>` as shown in Listing 12-17:

LISTING 12-17: page_crypt_info structure from osfmk/kern/page_decrypt.h

```
/*
 *Interface for text decryption family
 */
struct pager_crypt_info {
        /* Decrypt one page */
        int     (*page_decrypt)(const void *src_vaddr, void *dst_vaddr,
                                unsigned long long src_offset, void *crypt_ops);
        /* Pager using this crypter terminates - crypt module not needed anymore */
        void    (*crypt_end)(void *crypt_ops);
        /* Private data for the crypter */
        void    *crypt_ops;
};
```

The `page_decrypt` field is a function pointer, a hook, which can be externally set for various decryption modules. This mechanism enables Apple to plug-in encryption modules in order to decrypt memory that is declared as "protected." OS X's XNU has a default module, the DSMOS, kernel extension.[*] In iOS the corresponding modules are FairPlayIOKit and TextEncryptionFamily, which links to it. In either case, the Apple Protect pager is totally oblivious of the decryption logic: When a data request arrives, it calls on `page_decrypt()` function to do all the work, as shown in Listing 12-18.

LISTING 12-18: Apple Protect data request

```
kern_return_t apple_protect_pager_data_request(
        memory_object_t         mem_obj,
        memory_object_offset_t  offset,
        memory_object_cluster_size_t            length,
#if !DEBUG
        __unused
#endif
```

[*]DSMOS is an acronym for "Don't Steal Mac OS X." This module has a very rigid (and threatening!) license, preventing any reverse engineering of it. Therefore, the detail of memory decryption stops here.

continues

LISTING 12-18 *(continued)*

```
        vm_prot_t               protection_required,
        memory_object_fault_info_t mo_fault_info)
{
...
                /*
                 * Decrypt the encrypted contents of the source page
                 * into the destination page.
                 */
                ret = pager->crypt.page_decrypt((const void *) src_vaddr,
                                                (void *) dst_vaddr,
                                                offset+cur_offset,
                                                pager->crypt.crypt_ops);

if (ret) {
                        /*
                         * Decryption failed.  Abort the fault.
                         */
                        retval = KERN_ABORTED;
                } else {
                        /*
                         * Validate the original page...
                         */
                        if (src_page->object->code_signed) {
                                vm_page_validate_cs_mapped(
                                        src_page,
                                        (const void *) src_vaddr);
                        }
                        /*
                         * ... and transfer the results to the destination page.
                         */
                        UPL_SET_CS_VALIDATED(upl_pl, cur_offset / PAGE_SIZE,
                                             src_page->cs_validated);
                        UPL_SET_CS_TAINTED(upl_pl, cur_offset / PAGE_SIZE,
                                           src_page->cs_tainted);
                }
```

Decrypted pages are never marked dirty, and therefore never swapped out to disk (which would defeat the entire purpose of the encryption, if a plaintext copy could be excavated from the swap file!). In fact, the Apple Protect pager cannot handle data return (read, page-out) requests and `panic()`s if this method is called.

Although this mechanism can be used for various kinds of encrypted memory, Apple currently uses it for encrypting binaries. Recall (from Chapter 3) that Mach-O segments can be protected. The kernel's Mach-O handler, `load_segment()`, checks whether the `SG_PROTECTED_VERSION_1` flag is set for a segment. If it is, it calls `unprotect_segment()`.

If XNU is compiled with `CONFIG_CODE_DECRYPTION`, as it is by default, then `unprotect_segment()` calls the Apple protect pager, as shown in Listing 12-19.

LISTING 12-19: unprotect_segment() from bsd/kern/mach_loader.c

```
#if CONFIG_CODE_DECRYPTION

#define APPLE_UNPROTECTED_HEADER_SIZE   (3 * PAGE_SIZE_64)

static load_return_t
unprotect_segment(
        uint64_t        file_off,
        uint64_t        file_size,
        struct vnode    *vp,
        off_t           macho_offset,
        vm_map_t        map,
        vm_map_offset_t map_addr,
        vm_map_size_t   map_size)

        struct pager_crypt_info crypt_info;

        crypt_info.page_decrypt = dsmos_page_transform;
        crypt_info.crypt_ops = NULL;
        crypt_info.crypt_end = NULL;
#pragma unused(vp, macho_offset)
        crypt_info.crypt_ops = (void *)0x2e69cf40;
        kr = vm_map_apple_protected(map,
                                    map_addr,
                                    map_addr + map_size,
                                    &crypt_info);

       }

        if (kr != KERN_SUCCESS) {
                return LOAD_FAILURE;
        }
        return LOAD_SUCCESS;
}
```

The `vm_map_apple_protected()` calls on `apple_protect_pager_setup()`, which iterates over the the AP pager's queue, and either looks for the object (if existing), or creates a new one. This way, when the `vm_map` is retrieved using a `data_request`, the AP pager can invoke the decryption function supplied.

As previously noted, while the effort in encrypting binaries in this way is a valiant one, it can be defeated quite easily. Mach's powerful vm_map APIs, which can be used outside the task, enable reading the task's memory directly, in which the memory is already decrypted — this is one of the things that the *corerupt* tool, presented in the chapter, can do. An even easier way is to force inject a library using `DYLD_INSERT_LIBRARIES` (as was discussed in Chapter 4), and just read the memory from inside the task. This is the reason why, despite App Store binaries being encrypted, iOS app piracy is thriving.

The Default Freezer (iOS)

The Default Freezer, a new addition in iOS, can be found in the Lion sources, though the compiled kernel does not use it (and, at this time of writing, it doesn't look like Mountain Lion will be using it, either). It will allow the system to selectively freeze a virtual memory image of a given task and restore it on demand. Note the use of future tense, "will" — this is still an evolving implementation.

> *The discussion in this subsection relies mostly on the open source of XNU, which (probably intentionally) leaks code segments dealing with hibernation, and some inspection of the kernel binary. The source, however, remains behind the iOS kernel version, and hibernation is virtually undocumented. The information herein is, therefore, subject to change, though the general ideas are likely to remain as described.*

The rationale for doing this can be found in mobile environments. Indeed, iOS's nemesis, Android, has this feature.† On systems with relatively low amounts of physical memory and no real swap, it is only a matter of time before a user, running too many applications, will also run out of memory. Applications in a mobile environment, however, most often have no real need to execute when not in the foreground. This is because the mobile platform normally only allows one app to be in foreground mode and use the screen. When the user switches between apps, the app can be "frozen," put in the background, then "thawed" as it resumes. Because the frozen app is not running in between the freeze and thaw operations, it can also, in theory, be killed altogether, then restored to the same register state and virtual memory image at a later time.

This ability is thus designed for iOS (think of all those times one switches away *Angry Birds* to answer a phone call, for example). Although Lion boasts a similar feature (resuming processes where the user left off), in OS X the implementation is done through the `CoreFoundation` framework, and is really a matter of saving the application state (in the Saved Application State directory). In iOS, the resumption of processes is performed by the the Default Freezer. The freezer is implemented in `osfmk/vm/default_freezer.c`, and is enabled if XNU is compiled with `CONFIG_FREEZE`. It is integrated into the kernel `memorystatus` mechanism (also known as Jetsam, discussed in Chapter 13), and provides new iOS specific system calls, such as `pid_suspend()` and `pid_resume()`. Note, that the current implementation of the freezer seems incomplete (for example, `pid_suspend()` cannot directly freeze a specific process) Chapter 13 discusses the mechanism in more detail.

PAGING POLICY MANAGEMENT

The Mach pager types discussed previously perform the dirty work of paging a memory object to or from its corresponding backing store, but they do not act on their own accord. They merely await callbacks (their published `data_request` and `data_return` methods). A separate entity must be able to direct them, and make the decision as to which pages should be committed.

†Note that Android's implementation is totally entirely different. Dalvik applications' programming model places the responsibility of saving state (as a "bundle") at the hands of the application, which responds to events. If the application is killed and restarted, its memory is reinitialized, not restored, but the application is passed the previous state, and may resume from it.

The Pageout Daemon

The pageout daemon isn't really a daemon, but a thread. Not just any thread: When `kernel_bootstrap_thread()` completes the kernel initialization and has nothing more to do, it literally becomes the pageout daemon, by a call to `vm_pageout()`, which never returns. The thread (with the help of a few others) manages the page swapping policy, deciding which pages need to be written back to their backing store.

vm_pageout thread:

The `vm_pageout()` function (in `osfmk/vm/vm_pageout.c`) converts the `kernel_bootstrap_thread` to the pageout daemon, by effectively resetting the thread. The function sets the thread's priority, initializes various paging statistics and parameters, and then spawns two more threads: The external iothread, and the garbage collector (a third, internal iothread, was started when the default pager is registered).

When the set up is done, `vm_pageout()` finally calls `vm_pageout_continue()`, which periodically wakes up to perform the `vm_pageout_scan()`. This is a massive, entangled function, which maintains four page lists (referred to as page queues). Every `vm_page` in the system is tied to one of these four by means of its `pageq` field:

- **`vm_page_queue_active`:** Pages recently active, and resident.
- **`vm_page_queue_inactive`:** Pages not recently active, and therefore candidates for paging out. These pages may be paged out, or reactivated, depending on their usage.
- **`vm_page_queue_free`:** The free page list. These are pages that were inactive, but have been laundered (page out).
- **`vm_page_queue_speculative`:** Pages which were speculatively mapped, as the result of a read-ahead. These are inactive, but are likely to be used very soon. This queue is composed of many "bins" (from `VM_PAGE_MIN_SPECULATIVE_AGE_Q) VM_PAGE_MAX_SPECULATIVE_AGE_Q`), and will generally be shielded from `vm_pageout_scan()` for a like number of milliseconds. Pages gradually age until they fall to inactive status, and join the `vm_page_queue_inactive`.

The function works to meet target values for all queues, maintained in the `vm_page_[active|inactive|free|speculative]_target` variables, and then blocks the thread. If the current values (maintained in similarly named `count` variables) fall below the targets, the thread is woken up. The check is usually performed as the last stage of a `vm_page_grab()` or other page operation.

The pageout daemon's statistics can be obtained by a call to `host_statistics[64]`, (`osfmk/kern/host.c`) with the `HOST_VMINFO[64]` request, as is shown in the next experiment:

Experiment: Virtual Memory Statistics

Recall from Chapter 4 the discussion of the `vm_stat(1)` command, used to display kernel virtual memory statistics. The kernel keeps these statistics in a `vm_statistics` struct, defined in `osfmk/mach/vm_statistics.h` as shown in Listing 12-20:

LISTING 12-20: vm_statistics64 struct, from vm_statistics.h

```
struct vm_statistics64 {
        natural_t       free_count;             /* # of pages free */
        natural_t       active_count;           /* # of pages active */
        natural_t       inactive_count;         /* # of pages inactive */
        natural_t       wire_count;             /* # of pages wired down */
        uint64_t        zero_fill_count;        /* # of zero fill pages */
        uint64_t        reactivations;          /* # of pages reactivated */
        uint64_t        pageins;                /* # of pageins */
        uint64_t        pageouts;               /* # of pageouts */
        uint64_t        faults;                 /* # of faults */
        uint64_t        cow_faults;             /* # of copy-on-writes */
        uint64_t        lookups;                /* object cache lookups */
        uint64_t        hits;                   /* object cache hits */

        /* added for rev1 */
        uint64_t        purges;                 /* # of pages purged */
        natural_t       purgeable_count;        /* # of pages purgeable */

        /* added for rev2 */
        /*
         * NB: speculative pages are already accounted for in "free_count",
         * so "speculative_count" is the number of "free" pages that are
         * used to hold data that was read speculatively from disk but
         * haven't actually been used by anyone so far.
         */
        natural_t       speculative_count;      /* # of pages speculative */

} __attribute__((aligned(8)));
```

The `vm_stat(1)` command therefore has very little work — just get the statistics using a `host_statistics64` call on `mach_host_self()`, and print it out. The code (which is part of Darwin's `system-cmds` package) has been little changed from Avadis Tevanian's original Mach code, having just been ported to Mac OS X and expanded to 64 bits. This is shown in Listing 12-21:

LISTING 12-21: Using vm_statistics64 in vm_stat (from system_cmds-541/vm_stat.tproj/vm_stat.c)

```
void get_stats(vm_statistics64_t stat)
{
     unsigned int count = HOST_VM_INFO64_COUNT;
     kern_return_t ret;
     if ((ret = host_statistics64 (mach_host_self(),
                                   HOST_VM_INFO64,
                                   (host_info64_t) stat,
                                   &count) != KERN_SUCCESS)) {
          fprintf(stderr, "%s: failed to get statistics. Error %d\n", pgname,ret);
          exit(EXIT_FAILURE);
          }
}
```

Taking this code and embedding it in your own `main()` is straightforward. A simple `printf()` of the structure fields from Listing 12-4, and there you have it — a quick implementation of `vm_stat(1)`.

vm_pageout iothreads

The internal and external iothreads each look at a corresponding `vm_pageout_queue_ts`, which are initialized by `vm_pageout()` as well. The `vm_pageout_queue_internal` is reserved for internal VM objects (i.e. those created by the kernel, are maintained by default pager, and have their `internal` flag set to `true`), and the `vm_pageout_queue_external` is used for all other VM objects.

Both threads employ the same thread function, `vm_pageout_iothread_continue()`, but on different queues. This function (technically, a continuation), loops over its queue, dequeueing each page, getting its corresponding pager (from its `vm_object` reference), and calling the pager's `memory_object_data_return()` function. This enables the pageout threads to be decoupled from the actual paging implementation, for which the pager is solely responsible.

Garbage Collection Thread:

The garbage collection thread (`vm_pageout_garbage_collect()`) is occasionally woken up on its continuation by `vm_pageout_scan()`. It handles garbage collection in three areas:

- **`stack_collect()`**: Pages from the kernel stack (implemented in `osfmk/kern/stack.c`)
- **`consider_machine_collect()`**: For machine dependent pages. In OS X, this is a null function (implemented in `osfmk/i386/pcb.c`)
- **`consider_buffer_cache_collect()`**: if the function is indeed defined. To define the function, the caller uses `vm_set_buffer_cleanup_callout()`. The BSD layer registers the `buffer_cache_gc()` in the `bufinit()` function. (Both are defined `bsd/vfs/vfs_bio.c`).
- **`consider_zone_gc()`**: For zone garbage collection, as discussed earlier in this chapter (This function is implemented in `osfmk/kern/zalloc.c`)

The garbage collection thread also calls `consider_machine_adjust()` (again, a null function in OS X). Finally, just before blocking on its continuation, it calls `consider_pressure_events()` (defined in `bsd/kern/vm_pressure.c`), which falls through to `vm_dispatch_memory_pressure()` (in the same file). This mechanism is tied into the BSD layer's Jetsam mechanism (somewhat akin to Linux's low memory killer), which is explored in Chapter 13.

XNU's paging code contains calls to `VM_CHECK_MEMORYSTATUS`, especially in the `osfmk/vm/vm_resident.c` functions (`vm_page_release()`, `vm_page_grab()`, and friends). In OS X, this is just an empty macro. In iOS, where physical memory is scarce and there is no swap, this macro calls `vm_check_memorystatus()`, which wakes up the `kernel_memorystatus` thread, also part of Jetsam.

Handling Page Faults

The `vm_pageout()` daemon only handles one direction of swapping — from the physical memory out to the backing store. The other direction, paging in, is handled when a page fault occurs. The logic is quite complicated, but can be simplified as follows:

- The machine level trap handler (Intel: `user/kernel_trap()`, ARM: `sleh_abort`) calls `vm_fault()` if the trap reason is a page fault.
- The `vm_fault()` function calls `vm_page_fault()` to handle the actual faulting page, and retrieve it from the backing store. This is done, as can be expected, by looking up the `vm_page`'s corresponding `vm_object`, and obtaining the pager port from it. The pager's `data_request` function then does the work of paging in the contents from the backing store. A page-in operation also decrypts the page (if it resides on encrypted swap) as well as validates its code signature, if any.
- `PMAP_ENTER()` inserts the page into the task's pmap.

Note, that there can be many types of page faults, and the behavior described above can be anticipated only when the fault is of a non-resident page type — that is, cases where the page is in the `vm_map`, but not in the `pmap`. Other cases of page faults include:

- **Invalid access:** Access to an address which is not mapped into the process address space (read: in the task's `vm_map`). This is what usually happens when a stray pointer is dereferenced. This results in a SIGSEGV to the process.
- **Page protection fault:** Access to an address which is mapped, but whose page protection mask forbids the requested access. This is generally the case with trying to jump to an address in a data segment (enforced by NX/XD in Intel, or the XN bit in ARM), or when trying to write (or read) to a non-writable (or non-readable) page. This results in a SIGBUS to the process (Debuggers use this mechanisms to insert watchpoints).
- **Copy-On-Write:** A page may also be marked read-only, so that if a task attempts to write to it, the fault is trapped, and the page may then be copied before the write operation is retried. This is a very common tactic to allow sharing of memory in a way that enables saving RAM. Most of the task's `vm_map` is shared in this way (as the process loads many shared libraries). The fault in this case is because of the kernel's "laziness" in not having pre-allocated a private copy of the page. The page fault handling code therefore handles this transparently in a manner similar to the above, and the task remains unaware that anything even happened.

  Pre-Leopard, the page fault logic also contained mechanisms for detection of the "task working set," used to pre-fetch non-contiguous pages related to the faulting task. This was meant as a read-ahead mechanism, to reduce subsequent page faults which result when a task is brought in from swap. This is no longer the case.

The dynamic_pager(8) (OS X)

Recall the `dynamic_pager`, discussed in Chapter 4. The `dynamic_pager(8)` is a user mode daemon, which maintains the system swap file, by default `/private/var/vm/swapfile`. The name is somewhat misleading, as this daemon isn't one of the actual pagers from Table 12-9, and therefore does not directly control paging operations. Rather, when the kernel's `default_pager` needs to resize or otherwise modify swap file settings in ways which require user mode intervention, it is called upon from kernel space.

The daemon communicates with the `default_pager` over Mach messages, and uses Mach traps to control system swapping. Specifically, when the daemon starts, it registers the `HOST_DYNAMIC_PAGER_PORT` (a host special port). It can also register a port as an alert port (using the `macx_triggers` trap) to get messages from the kernel. The kernel can then send messages to the daemon, which performs the required support operations in user mode (namely, creating, resizing or removing a file), and can invoke Mach traps to inform the kernel. These traps are actually defined as part of the BSD layer, in `bsd/vm/dp_backing_file.c`, as shown in Table 12-10.

TABLE 12-10: Mach Traps Used By the dynamic_pager(8) Program

MACH TRAP	USAGE
`macx_swapon` `(uint64_t filename,` ` int flags,` ` int size,` ` int priority);`	Starts swapping to a given file. Mach interface for BSD's `swapon()`. This is a wrapper, which communicates with `default_pager`. Calls `default_pager_backing_store _create()` and `default_pager_add_file()`.
`macx_swapoff` `(uint64_t filename` ` int flags);`	Stops swapping to the given file. Calls `default_pager_backing_store_delete()`.
`macx_triggers` `(int hi_water,` ` int low_water,` ` int flags,` ` mach_port_t alert_port);`	Sets callbacks for high and low water marks (used for the `-H` and `-L` switches, respectively). This is a fall through to `mach_macx_triggers()`. Also used to set encryption on swap, if UseEncryptedSwap is set in the `dynamic_pager`'s plist. The `dynamic_pager` also uses this to registers its port as the *`alert_port`*, to which the kernel will send messages on high/low water marks.

SUMMARY

This chapter focused on one of Mach's (and, by extension, XNU's) most important and complicated, yet least understood systems — virtual memory. In particular, we elaborated on the machine-independent virtual memory layer, which enables the Mach core to adapt to multiple architectures, and the machine-specific physical memory, pmap, which binds to them. Through the high-level abstraction of `vm_map`, which represents the task address space, virtual memory regions may be allocated, adjusted, shared, and freed according to need.

Additionally, we discussed kernel memory allocator mechanisms, especially those based on Mach zones, which allow a higher level of abstraction, akin to the user mode's `malloc(3)`.

The chapter then turned to paging, with an exploration of Mach's pagers, which allow to extend the backing store of virtual memory onto swap, memory mapped files, devices or even remote hosts. All five pagers, common to OS X and iOS, were discussed, as well as iOS's new Default Freezer. We concluded with an explanation of the workings of the pageout daemon and the dynamic pager, both performing important operations despite misleading names.

As this chapter concludes, so does the detailed subsection of this book dealing with Mach. The next chapters focus on the various components of the BSD layer (Chapter 13), advanced BSD primitives (Chapter 14), and then the subsystems of files (VFS, Chapter 15) and networking (Chapter 17).

REFERENCES

1. Rashid, Tevanian, Young, Golub, Baron, Black, Bolosky, and Chew, CMU. "Machine-Independent Virtual Memory Management of Paged Uniprocessor and Multiprocessor Architectures," ACM October, 1987

13

BS”D — The BSD Layer

Mach is merely a microkernel. Although some of its application programming interfaces (APIs) are exposed to user mode, developers mainly use the much more popular API of POSIX, which is implemented by the BSD layer of Mach.

This chapter discusses the BSD layer in considerable depth. “Considerable” because BSD by itself is a complicated design spanning many implementations, notably FreeBSD and its various sister operating systems. XNU largely conforms to 4.4BSD, and so, in places where this book leaves off for brevity, refer to the BSD documents[1] listed in the references for this chapter.

This chapter starts with the discussion of the standards that BSD implements. It then discusses, in order, the fundamental objects of BSD: processes, threads, and the executable programs that create them. It then continues to talk about process control calls, in particular `ptrace(2)`, and the undocumented policy control functions.

The chapter concludes by discussing UNIX signals, and how they correspond with the processor traps and Mach exceptions discussed in Chapter 11. Discussion of more advanced topics, or features that are Apple proprietary, is left for the next chapter.

INTRODUCING BSD

Even before its incarnation in XNU, Mach was closely integrated with BSD. Mach traps and services alone cannot provide for a full operating system, and by design are not meant to. After all, they do not include something as fundamental as a file system. Another layer needs to build on top of these primitives the well-known abstractions of files, devices, users, groups, and more. The layer originally chosen in Mach, and kept in XNU, is BSD.

BSD and POSIX user mode developers in OS X can remain blissfully ignorant of the Mach layers. Even though the Mach APIs are still accessible in user mode via the Mach traps discussed Chapters 11 and 12, XNU’s primary “personality” is that of BSD, and the system exposes the full set of POSIX system calls. Though the fact is little known, Mac OS X received official

UNIX03[2] certification in Leopard, something that most UNIX-like systems, including Linux, cannot really claim. (Apple received this certification from The Open Group in May 2007 and is due for renewal as this book goes to print).

One Ring to Bind Them

The UNIX03 certification means that OS X conforms to the Single UNIX specification, commonly referred to as SUS. Following the great divide, UNIX has proliferated into so many versions and flavors that developers could no longer write portable code without having to consider OS idiosyncrasies.

FIGURE 13-1: The logo of The Open Group, holders of the UNIX trademark (with apologies to NH)

The need for a reuniting standard emerged to once more allow portability, enabling developers to write code they can deploy on multiple operating systems, conforming to said standard. Portability is of two types:

- **Source-level compatibility**: This type implies that, even though the underlying architecture might be different, all the common system APIs are identical. As such, compiling code cleanly on the operating system–compatible compiler must be possible so as to create a binary that executes with the exact expected behavior.
- **Binary compatibility**: This type is a stronger requirement than source-level compatibility and implies that the program, once compiled, could be moved from one standards-compliant operating system to the other (assuming the same underlying machine architecture) and would run seamlessly.

Somewhat surprisingly, OS X makes no attempt for binary compatibility. In fact, at the time of this writing, binary compatibility is impossible by design because the native binary format of OS X is still the venerable Mach-O executable, which is yet another legacy of OS X's NextSTEP roots. Indeed, other UNIX-like systems, such as BSD, Linux, and Solaris, are somewhat closer to this in that they all agree on the Executable and Library Format (ELF), which is the de facto standard in UNIX-like environments, save OS X.

UNIX03 demands only source-level compatibility, however. With OS X declared compliant, SUS-conforming sources, which rely on common and standardized APIs, are guaranteed to be able to compile neatly on OS X.

Note that the standards compliance ensures only compatibility for the minimum approved standard. It does not imply the compliant system cannot expose its own idiosyncratic APIs, at the cost of breaking compatibility with other operating systems. Indeed, OS X has many such APIs that don't even begin to compile on other operating systems. Mach-O is just one. It is therefore going to be a long time before non-Apple operating systems can execute OS X binaries.

What's in the POSIX Standard?

SUS v3 is aligned with another standard, POSIX (known also by another name, IEEE Std 1003.1-2001). Table 13-1 shows some of what the standard includes.

TABLE 13-1: Single UNIX specification components

SUS PART	MAN SECTION	CONTAINS
Base definitions (XBD)	4, 5, 7	Conventions that are expected of a UNIX system. This lengthy tome contains 13 chapters describing everything from environment variables and regular expression syntax through the common file system, devices, and tty specifications found on UNIX. Additionally, the last chapter lists the constants, macros, and data structures exposed by the operating system. These are available to the developer as the familiar `#include` files in `/usr/include`. The well-known `<unistd.h>` and `<stdlib.h>`, alongside programmatic lynchpins such as `<stdio.h>`, `<string.h>`, and nearly 100 other header files are included in this part of the standard.
System Interfaces (XSH)	2, 3	The APIs exposed by the system. Drawing on the standard data structures and constants from XBD, this specification defines the system calls (section 2 of the manual) and library calls (section 3 of the manual).
Base Utilities (XCU)	1, 6, 8	The shell (the familiar bash, ksh, and csh, at a bare minimum) with some 150 command-line utilities making up the familiar contents of the `bin` and `sbin` directories. From the man perspective, XCU contains sections 1 (user commands) and 8 (system administration commands).

Implementing BSD

To expose the BSD APIs, XNU actually borrows code from the BSD code-base itself. Much of the kernel code in the `bsd/` directory is the original BSD code, which still contains the required copyright of the BSD license. The BSD license is considered to be very permissive, which allows Apple to close off its operating system on a whim, as it has indeed done in iOS.

Like the original NeXTSTEP ancestor, which was Mach 2.5 tied to 4.3 BSD, so is xnu now based on Mach 3.0, and tied to 4.4 BSD (and sharing a common code base ancestry with FreeBSD).

XNU Is *Not* Fully BSD

Although XNU exports a fully functional BSD layer and API, it is not a full BSD implementation. Parts of it, such as the Virtual Filesystem Switch (VFS) and network architecture, were copied fully, but others were either partially ported or completely omitted. A few of the well-known BSD APIs, such as `sbrk()` and `swapon()`, are missing. Additionally, XNU's kexts (kernel extensions) are incompatible with BSD's kmods (kernel modules), and I/O Kit is entirely unique in XNU. As a consequence, OS X remains a BSD-like system (and, in the UNIX genealogy, clearly sides with the BSD branch, rather than AT&T's), but cannot be considered fully BSD.

PROCESSES AND THREADS

The primitives and algorithms of Mach scheduling — tasks and threads — are discussed in great detail in Chapter 10. As mentioned, Mach provides these primitives as low-level abstractions with a deliberately basic and incomplete API, on top of which the upper layers are expected to implement the full functionality.

BSD takes the two primitives and structures them into the well-known concepts of *process* and *thread* from the UNIX landscape. This section goes on to discuss the specific BSD implementation of processes and threads, and how it ties to the underlying Mach layer. Note that this builds on the basic concepts of processes in UNIX, which were introduced in Chapter 4. If you are somewhat unfamiliar with these concepts, you might want to review Chapter 3 before going on with this chapter.

BSD Process Structs

Mach provides a rich abstraction of tasks and threads, but is still incomplete and leaves much to be desired. A BSD *process* can be uniquely mapped to a Mach task, but it contains more than the basic scheduling and statistics information the Mach task offers. Most notably, BSD processes contain file descriptors and signal handlers. Processes also support the complex genealogy linking them with their parents, siblings, and children.

BSD maintains these features of a process and many more by means of a `struct proc`, which is yet another mammoth structure, defined in `bsd/sys/proc_internal.h`. XNU's version of the `struct proc` is similar to that of BSD, but contains many idiosyncratic fields, relating to DTrace support, code signing, work queues, and other specific features. Rather than fill page after page with a listing of this huge structure, Table 13-2 highlights the important fields (shaded rows denote parameters which copy over on process `fork()`:

TABLE 13-2: Important fields of the struct proc (not in order)

FIELD	PURPOSE
`LIST_ENTRY(proc) p_list;`	Ties proc to list of all running processes.
`pid_t p_pid, p_ppid, p_pgrpid;`	PID, PPID, and PGRP of this process.
`uid_t p_uid, p_ruid, p_svuid,` `gid_t p_gid, p_rgid, p_svgid;`	UIDs and GIDs (current, real and saved) of process.

FIELD	PURPOSE
`void * task;`	Pointer to underlying Mach task.
`char p_stat;`	Process status (letter shown in `PS`).
`struct proc * p_pptr;`	Pointer to parent process (`this->p_pptr->p_pid == this->ppid`).
`LIST_ENTRY(proc) p_pglist;` `LIST_ENTRY(proc) p_sibling;` `LIST_ENTRY(proc) p_children;`	Fellow members in same PGRP, siblings (other processes which are children of same ppid), and children of this process (which are all siblings to one another).
`LIST_ENTRY(proc) p_hash;`	Pointer to process hash chain entry.
`TAILQ_HEAD(, uthread) p_uthlist;`	All of the BSD threads in to this process.
`TAILQ_HEAD( ,eventqelt) p_evlist;`	Events associated with this process.
`struct filedesc *p_fd;`	Open file descriptors. The `int fd` from user space is an index into this `p_fd` array.
`struct sigacts *p_sigacts;`	Signal behaviors.
`struct plimit *p_limit;` `struct timeval p_rlim_cpu;`	Process resource limits (from `setrlimit(2)`). The remaining CPU time is maintained separately.
`pid_t si_pid;` `u_int si_status;` `u_int si_code;` `uid_t si_uid;`	Fields initialized from last SIGCHLD in case this process has spawned children and needs to collect their exit code.
`u_int p_argslen;` **`int p_argc;`**	Length and number of command-line arguments.
`char p_comm[MAXCOMLEN+1];` **`char p_name[(2*MAXCOMLEN)+1];`**	Command line and process name.
`user_addr_t *user_stack;`	Address of user mode stack.

continues

TABLE 13-2 *(continued)*

FIELD	PURPOSE
`u_char p_priority;` **`u_char p_resv0;`** **`char p_nice;`** **`u_char p_resv1;`**	BSD priority and nice fields, as well as calculated fields.
`struct vnode *p_textvp;` **`off_t p_textoff;`** **`uint8_t p_uuid[16];`**	Pointer to vnode of executable that is making up this process image and the offset in it. The UUID is copied from the Mach-O **`LC_UUID`**.
`sigset_t p_sigmask;` **`sigset_t p_sigignore;`** **`sigset_t p_sigcatch;`**	Signals masked, ignored and caught by this process. (sigmask is deprecated).
`int p_mac_enforce;`	Is process subject to MAC enforcement?
`uint32_t p_csflags;`	Code-signing flags (discussed later).
`int p_iopol_disk;`	In iOS controls process I/O policy for disk.
`int p_aio_total_count; int p_aio_active_count;` `TAILQ_HEAD` `( , aio_workq_entry )` `p_aio_activeq;` `TAILQ_HEAD` `( , aio_workq_entry )` `p_aio_doneq;`	Asynchronous I/O support: Counts and lists of AIO requests.
`struct lctx *p_lctx; LIST_ ENTRY(proc) p_lclist;`	Support for login contexts: pointer to current login context, and processes in that context.
`user_addr_t p_threadstart;` `int p_pthsize;` `void * p_pthhash;`	Pthread support. Size of thread, thread function, and pointer to `pthread waitqueue` hash.
`user_addr_t p_wqthread;` `void *p_wqptr;` `int p_wqsize;` `boolean_t p_wqiniting;` `lck_spin_t p_wqlock;`	Work queue support (discussed in more detail in the next chapter).

*Bold rows imply parameters that copy over on process `fork()`

The structure is so massive it requires several disjoint locks to protect access to its various fields, and the lists it participates in. The process lock (PL) locks the entire structure, but there exist a process spin lock (PSL), a file descriptor lock (PFDL), and others that lock the groups and siblings.

Process Lists and Groups

XNU maintains processes in `struct proclist` variables, which are really nothing more than linked lists of `struct proc`. There are two such lists and a special iterator function to traverse them, as shown in Listing 13-1.

LISTING 13-1: proclists in XNU, from bsd/sys/proc_internal.h (implementation in bsd/kern/kern_proc.c)

```
LIST_HEAD(proclist, proc);

/* defns for proc_iterate */
#define PROC_ALLPROCLIST        1   /* walk the allproc list (procs not exited yet) */
#define PROC_ZOMBPROCLIST       2   /*  walk the zombie list */
#define PROC_NOWAITTRANS        4   /* do not wait for transitions (checkdirs only)  */

extern struct proclist allproc;     /* List of all processes. */
extern struct proclist zombproc;    /* List of zombie processes. */
...
int proc_iterate(int flags,                      //  PROC_* flags, above
                 int (*callout)(proc_t,void *), // funciton to execute on each item
                 void *arg,                     // 2nd argument to callout
                 int (*filterfn)(proc_t,void *),// function to decide callout execution
                 void *filterarg);              // 2nd argument to be passed to filterfn
```

Processes may also belong to a process group, in which case an additional `struct pgrp` is used, as shown in Listing 13-2:

LISTING 13-2: Process group declaration in bsd/sys/proc_internal.h (implemented in bsd/kern/kern_proc.c)

```
// In the following, LL implies LIST_LOCK, and PGL implies Process Group Lock, which
// are system wide locks used to protect structure fields against concurrent access

struct  pgrp {
   LIST_ENTRY(pgrp)  pg_hash;      /* Hash chain. (LL) */
   LIST_HEAD(, proc) pg_members;   /* Pointer to pgrp members. (PGL) */
   struct  session * pg_session;   /* Pointer to session. (LL ) */
   pid_t             pg_id;        /* Pgrp id. (static) */
   int               pg_jobc;      /* # procs qualifying pgrp for job control (PGL) */
   int               pg_membercnt; /* Number of processes in the pgrocess group (PGL) */
   int               pg_refcount;  /* number of current iterators (LL) */
   unsigned int      pg_listflags; /* (LL) */
   lck_mtx_t         pg_mlock;     /* mutex lock to protect pgrp */
};
..
..
/* defns for pgrp_iterate */
#define PGRP_DROPREF            1
```

continues

LISTING 13-2 *(continued)*

```
#define PGRP_BLOCKITERATE       2
..
..
// pgrp_iterate is used to iterate over the pgrp->pg_members list
extern int pgrp_iterate(struct pgrp * pgrp, // pgrp to iterate over
               int flags,
               int (*callout)(proc_t , void *), // function to execute on each item
               void *arg,                        // 2nd argument to be passed to callout
               int (*filterfn)(proc_t , void *),// function to decide callout execution
               void *filterarg);                 // 2nd argument to be passed to filterfn
```

The iterator functions, both `proc_iterate()` and `pgrp_iterate()`, operate very similarly, as they both traverse linked lists. The former function looks at the `allproclist` (if `PROC_ALLPROCLIST` is set in flags) and at the `zombproclist` (if `PROC_ZOMBPROCLIST` is set in flags), whereas the latter looks at the `pg_members` field of the `pgrp`.

The iterators both accept a `filterfn`, a pointer to a function, which, if set, will be called for each process in the list, along with an optional `filterarg`. If the function returns a non-zero value (or no function exists to begin with), the callout function will be applied on the process in question, with an optional `calloutarg`. A good example of how this mechanism is used can be found in the process-killing logic, implemented by `killpg1() bsd/kern/kern_proc.c`, which is also described in the "Signals" section of this chapter.

Threads

Processes serve as containers, but the actual execution units of a binary are threads. Mach provides the thread primitive, but — yet again — it is insufficient for the requirements of higher level operating systems. A richer, more standardized API therefore needs to be provided by XNU.

The BSD Thread Object

BSD thread objects are defined as instances of a `struct uthread`, which is defined in `bsd/sys/user.h`. Again, we are dealing with an overwhelming, large structure with inline structures that further inhibit readability. Listing 13-3 attempts to simplify as much as possible, by highlighting the important fields:

LISTING 13-3: The struct uthread, from bsd/sys/user.h

```
struct  uthread {
   /* syscall parameters, results and catches */
   u_int64_t uu_arg[8]; /* arguments to current system call */
   int      *uu_ap;      /* pointer to arglist */
   int uu_rval[2];

   /* thread exception handling */
   int      uu_exception;
   mach_exception_code_t uu_code;  /* ``code'' to trap */
   mach_exception_subcode_t uu_subcode;
   char uu_cursig;               /* p_cursig for exc. */

   /* support for syscalls which use continuations */
   struct _select { .. } uu_select;
```

```
    union {
     struct _kqueue_scan { } ss_kqueue_scan; /* saved state for kevent_scan() */
     struct _kevent { }   ss_kevent;         /* saved state for kevent()      */
     } uu_kevent;
     struct _kauth { } uu_kauth;
           ..
     /* internal support for continuation framework */
     int (*uu_continuation)(int);
     int uu_pri;
     int uu_timo;
     caddr_t uu_wchan;                /* sleeping thread wait channel */
     const char *uu_wmesg;            /* ... wait message             */
     int uu_flag;

     int uu_iopol_disk;               /* disk I/O policy */ // iOS only

     struct proc * uu_proc;          // parent to owning process
     void * uu_userstate;
     // ...
     // signal stuff (uu_sig* fields)
     struct vfs_context uu_context;   /* thread + cred */
     sigset_t  uu_vforkmask;          /* saved signal mask during vfork */

     TAILQ_ENTRY(uthread) uu_list;   /* List of uthreads in proc */
     struct kaudit_record    *uu_ar; /* audit record */
     struct task*    uu_aio_task;    /* target task for async io */

     lck_mtx_t       *uu_mtx;

     // throttled I/O support…

     struct kern_sigaltstack uu_sigstk;
     int             uu_defer_reclaims;
     int             uu_notrigger;    // should this thread trigger automount?
     vnode_t         uu_cdir;         /* per thread CWD */
     int             uu_dupfd;        /* fd in fdesc_open/dupfdopen */

     // JOE_DEBUG's stuff..

     // DTRACE support ..

     void *          uu_threadlist;
     char *          pth_name;     // used for pthread_setname_np (over proc_info)
     struct ksyn_waitq_element  uu_kwe;     // use*d* for pthread synch
};
```

A mysterious developer, forever known as JOE laced BSD thread handling code all over XNU with conditional logic for debugging. If you peek at `bsd/sys/user.h`, `bsd/vfs/vfs_subr.c`, *and* `bsd/vfs_bio.c`, *you will see quite a few* `#ifdef JOE_DEBUG` *statements. None of them are in the release kernel, because* `JOE_DEBUG` *is* `#defined` *to 0 in* `osfmk/i386/loose_ends.c`. *Nonetheless, the* `#ifdefs` *have been around for a while now (at least since XNU 792), and are still in the Lion kernel sources.*

User mode threads begin with a call to `pthread_create`. This function doesn't do too much, as its main functionality provided by the `bsdthread_create` system call, whose implementation is in `bsd/kern/pthread_synch.c`. `bsdthread_create()` is basically a long wrapper over Mach's thread create. It is the underlying Mach layer that creates the thread object. `bsdthread_create()` merely goes on to set up its stack, if a custom stack is specified, its (machine-specific) thread state, and custom scheduling parameters, if any. Figure 13-2 shows this flow in more detail.

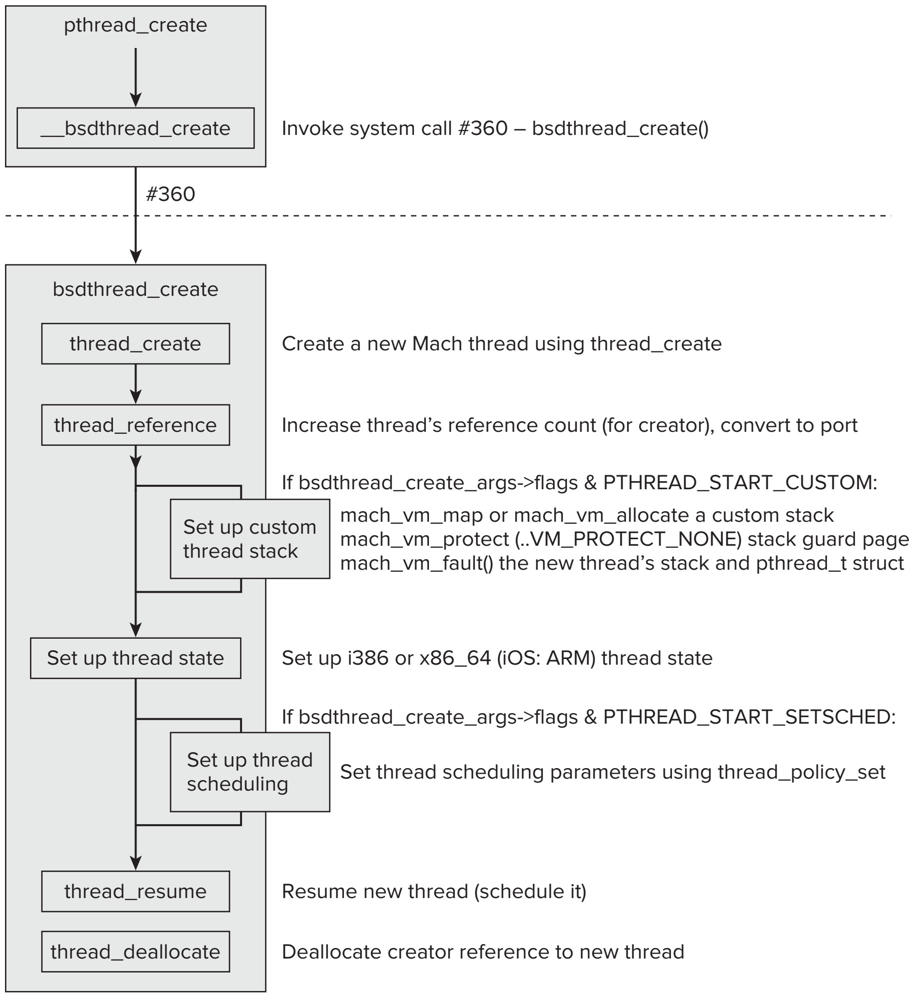


FIGURE 13-2: Flow of thread creation

Mapping to Mach

As you saw in Chapter 11, the underlying Mach microkernel is what actually implements the primitives for the massive process and thread structures. Every Mach task contains a `bsd_info` pointer to its corresponding BSD `proc` structure, and likewise, Mach threads contain a `uthread` field pointing to the corresponding `struct uthread`. These pointers are void, so Mach functions need not know the specifics of the BSD structures. Similarly, the BSD process points back to its corresponding task using a task field (again, a void *), and a BSD thread (`uthread`) points to the corresponding Mach thread using a `vc_thread *` field, which is itself a subthread of a field called `uu_context`. This is shown in Figure 13-3.

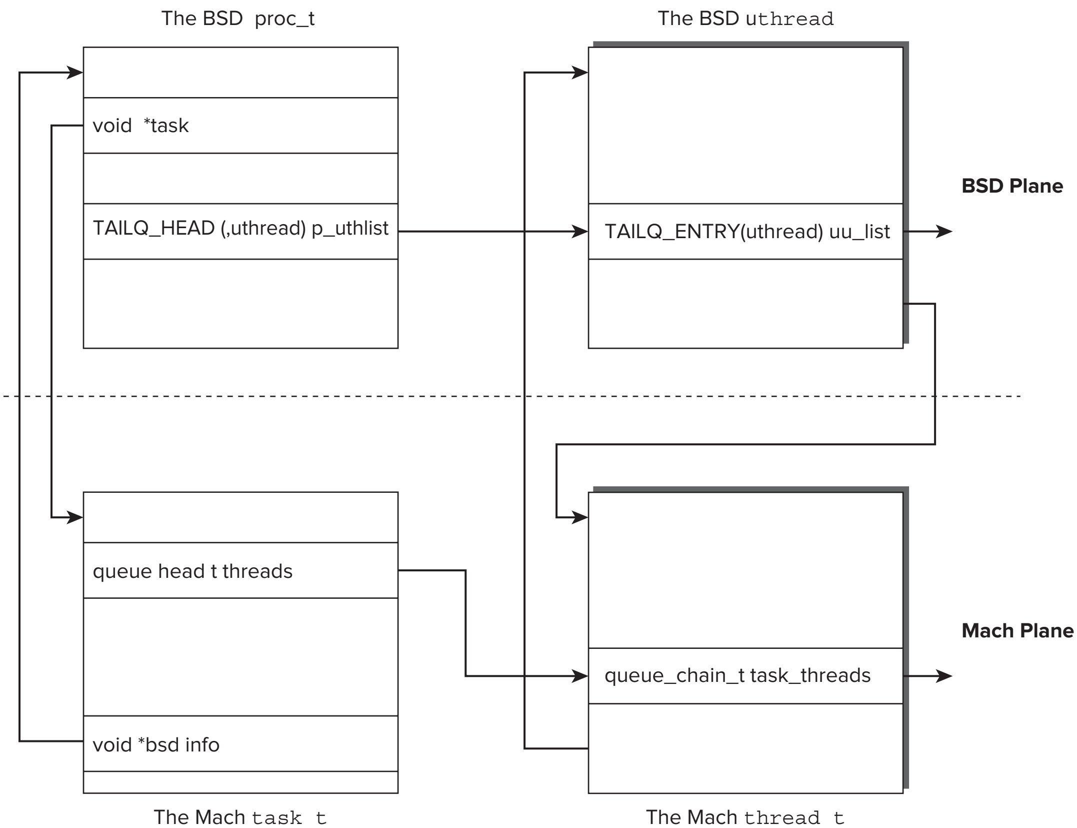


FIGURE 13-3: Mach processes and threads, mapped to BSD threads

Even though the pointers are straightforward to follow, helper functions, such as `get_bsdtask_info(task_t)` and `get_bsdthread_info(thread_t)`, which are both in `osfmk/kern/bsd_kern.c`), exist. They help preserve the implementation abstraction. On top of them, other functions, such as `current_proc()` in `bsd/kern/bsd_stubs.c`, can be implemented (essentially by wrapping `get_bsdtask_info()` on the `current_task`).

From the Mach side, the Mach call of `task_for_pid()` (`bsd/vm/vm_unix.c`) exists for mapping a BSD PID to the underlying Mach task port. This call used to include PID 0 (the Mach `kernel_task`), but now rejects this argument as invalid. The `task_for_pid()` call is deprecated, and in iOS also requires special entitlements (and therefore requires code-signing the binary, *and* root permissions for a process not owned by you). This is for (obvio us) security reasons: Getting the task port of an arbitrary PID opens a Pandora's box of mischief and malice, enabling (among other things) one to read and modify that task's memory image. The coreruption tool, presented in Chapter 12, demonstrates just how powerful these abilities are. As noted earlier in this book, obtaining the kernel_task's port (for PID 0) is tantamount to omnipotence, which is why jailbreakers patch the call and re-enable PID 0.

In XNU, all kernel threads are Mach threads and have no corresponding BSD processes. That is, their uthread * is NULL, and they are contained in the kernel_task. Likewise, the kernel_task has no BSD process identifier (save PID 0, as just described).

PROCESS CREATION

Chapter 4 discussed binary loading by the kernel and dyld fairly in depth, but did not go through the actual detail from the kernel perspective. This section picks up where Chapter 4 left off, by discussing this perspective in depth.

The User Mode Perspective

The UNIX model (with which OS X complies) does not support the concept of a "new" or "empty" process. In UNIX, a process cannot be created, only duplicated using the fork() system call. fork() is a special system call in that it is called once, but returns twice:

- In the child process, fork() returns 0.
- In the parent process, fork() returns the PID of the child.

If the fork() operation fails, fork() returns only in its calling process, with a return value of -1, and with errno set appropriately, usually EAGAIN or ENOMEM.

The child process is an exact duplicate of its parent, with a few notable exceptions:

- File descriptors, though having the same numbers and pointing to the same files, are copies of the original descriptors. This means that subsequent calls that modify the descriptors (e.g., lseek() or close()) affect only the process that made them.
- Resource limits, as per getrlimit(2) or ulimit(1), are inherited by the child, but utilization is set to zero.
- The memory image of the child seems (from the virtual memory perspective) private to the child but is, in fact (from the physical memory perspective), shared with the parent, using the same physical pages in memory. The virtual privacy is assured by setting the copy-on-write bit on the pages, so that either process — child or parent — attempting a write to a page triggers a page fault. In handling the page fault, the kernel duplicates the page, creating a separate physical copy of the same page, and breaking the mapping.

The last point, physically sharing the same memory pages, greatly facilitates process creation, as no memory is actually copied during the creation of the child, but does incur the overhead of duplicating the page tables and setting copy-on-write. A duplicate process, however, is seldom of any use. Most child processes continue to overwrite the entire memory space with a new memory image — that of the executable being loaded. A somewhat more efficient system call, vfork(), was created to take advantage of this fact by skipping any address space operations, essentially making any access to process memory in the child illegal. This is fine because this memory is overwritten with the new executable image anyway. vfork(), however, is largely considered deprecated.

A third system call, posix_spawn(), has been defined in the POSIX standard to facilitate process creation and subsequent image execution. This system call is defined in <spawn.h>, as shown in Listing 13-4.

LISTING 13-4: posix_spawn

```
int posix_spawn(pid_t *restrict pid,        // OUT pointer to spawned process pid
 const char *restrict path,                 // absolute or relative path to the image
 const posix_spawn_file_actions_t *file_act,// set up by posix_spawn_file_actions_init()
 const posix_spawnattr_t *restrict attrp,   // set up by posix_spawnattr_init()
 char *const argv[restrict],                // argv[0], or full argv[] command-line
 char *const envp[restrict]);               // environment pointer (same as in exec*e)
```

There are several advantages in using posix_spawn over the traditional fork()/exec() model, including that it enables using one system call, rather than two. Additionally, posix_spawn() allows fine-grained control over attribute and file descriptor inheritance, achieved via the third and fourth parameters: file_actions and the spawn attributes, as shown in Listing 13-5.

LISTING 13-5: posix_spawn_file_actions_t and posix_spawnattr_t manipulation

```
int posix_spawn_file_actions_init(posix_spawn_file_actions_t *file_actions);
int posix_spawn_file_actions_addopen
    (posix_spawn_file_actions_t *restrict file_actions,
     int filedes, const char *restrict path,
    int oflag, mode_t mode);
int posix_spawn_file_actions_adddup2(posix_spawn_file_actions_t *file_actions,
                                     int filedes, int newfiledes);
int posix_spawn_file_actions_addclose(posix_spawn_file_actions_t *file_actions,
                                      int filedes);
int posix_spawn_file_actions_destroy (posix_spawn_file_actions_t *file_actions);
int posix_spawnattr_init(posix_spawnattr_t *attr);
int posix_spawnattr_getflags(const posix_spawnattr_t *restrict attr,
                             short *restrict flags);
int posix_spawnattr_getpgroup(const posix_spawnattr_t *restrict attr,
                               pid_t *restrict pgroup);
int posix_spawnattr_getsigmask(const posix_spawnattr_t *restrict attr,
                               sigset_t *restrict sigmask);
int posix_spawnattr_setflags(posix_spawnattr_t *attr, short flags);
int posix_spawnattr_setpgroup(posix_spawnattr_t *attr, pid_t pgroup);
int posix_spawnattr_setsigmask(posix_spawnattr_t *restrict attr,
                               const sigset_t *restrict sigmask);
int posix_spawnattr_destroy(posix_spawnattr_t *attr);
```

The Kernel Mode Perspective

Regardless of the system call used — fork(), vfork(), or posix_spawn() — all paths in the kernel converge in the same underlying implementation, called fork1(), as shown in Figure 13-4. Its behavior, however, differs based on its third parameter — kind — for which each function passes a different value. These values are shown in Table 13-3:

```
int fork1 (proc_t parent_proc, thread_t *child_threadp, int kind);
```

TABLE 13-3: fork1() "kinds" and their behavior

KIND	PROCESS CREATED	ADDRESS SPACE
`PROC_CREATE_FORK`	Complete	Copied (on write)
`PROC_CREATE_VFORK`	Partial	Newly created
`PROC_CREATE_SPAWN`	Complete	Lazy (Invalid)

It is `fork1()` that eventually creates the new process by creating a new Mach task for the process. Though it serves as a focal point for the three functions it quickly splits back into the three distinct cases by `switch()`ing on its `kind` argument, which indicates which one of the three called it, as shown in Figure 13-5. For `vfork`, it calls `forkproc()`, discussed in the following section. Otherwise, `cloneproc()` is preferred. The latter wraps over `forkproc()`, but performs many more tasks, as will be discussed.

`posix_spawn()` and `fork()` calls are handled in the same way, save `dup`'ing the parent process's thread state into the `child_thread`, which is done only in `fork` by `thread_dup()`. Following the call to `clone`/`forkproc`, `fork1()` marks the child as `fork`ed, but not `exec()`ed (using the `AFORK` setting on its `p_acflag` field), and if not `posix_spawn()`ed, handles DTrace.

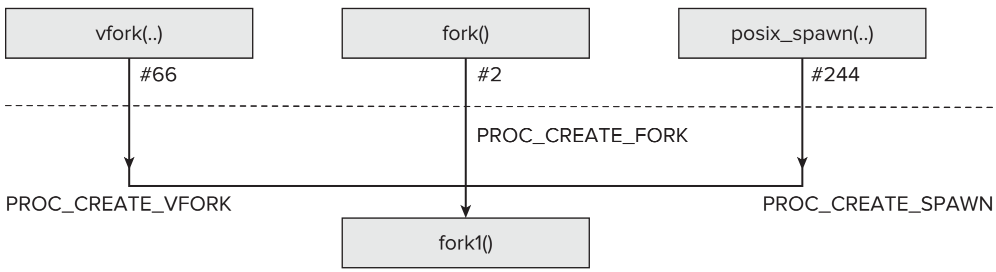


FIGURE 13-4 All paths leads to fork1()

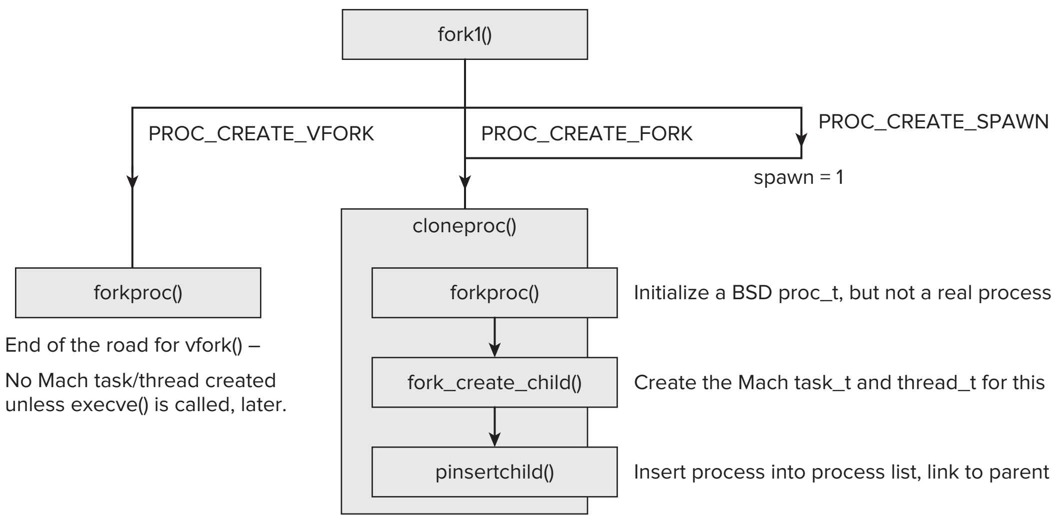


FIGURE 13-5: Fork() and demultiplexing the various process creation calls

The forkproc() Function

The `forkproc()` function is in charge of doing the work of initializing the new process's `proc_t` structure, whether from `fork()`, `vfork()`, or `posix_spawn()`. It proceeds in the following way:

- Allocates the `child_proc proc_t` from the `M_PROC` zone, and `bzero`s it.
- Allocates the child's statistics (`p_stats`) and signal actions (`p_sigacts`).
- Allocates the interval timer callout (`p_rcall`).
- Gets a PID for the child, accommodating for possible wrapping of the PID past `PID_MAX` `(99999)`. Inserts in the PID hash table.
- Initializes other process fields. Most of these are `bcopy()`ed directly from the parent, from in between the parent's `p_startcopy` (set to `p_argslen`) and `p_endcopy` pointers (`p_aio_totalcount`). Some are filtered out. For example, the only `p_flags` inherited are `P_LP64`, `P_TRANSLATED`, `P_AFFINITY`, `P_DISABLE_ASLR`, and `P_PROFIL`.
- Copies all the parent's file descriptors, using `fdcopy()`.
- Copies System V shared memory from the parent (`#if SYSV_SHM`), using `shmfork()`.
- Copies the parent's resource limits (as in `ulimit(1)` or `setrlimit(2)`) using `proc_limitfork()`.
- `Memset`s the `p_stats` from `pstat_startzero` (`p_ru`) to `endzero` (`p_start`) using `bzero()`, and `record p_start` (the process start time) to be now.
- If the parent has defined signal actions (`p_sigacts`), copies them over, or else initializes the child's to be all `NULL`.
- Sets child's controlling terminal, if any.
- Blocks all signals by `proc_signalstart (child_proc,0)` and marks as in transition (using `proc_transstart(child_proc,0)`).
- Initializes the child's thread list (`p_uthlist`) and asynchronous I/O queues.
- Inherits the parent's code-signing flags.
- Copies the parent's work queue information.
- If the parent is in the login context, (and `#if CONFIG_LCTX`), adds the child as well, using `enterlctx();`.

Note that one very important aspect is missing from this function — the creation of the actual process and thread at the Mach level. This is not done in the case of a `vfork()`, but only in `fork()` and `posix_spawn()`. This is why `forkproc()` is only called directly from `vfork()`, and is otherwise wrapped by `cloneproc()` (discussed next), which also creates the required Mach constructs. A `vfork()`ed process has no corresponding Mach task or thread. Only if it is followed by an `execve()` will those items be created for it. In fact, a `vfork()` process has no *raison d'etre* other than next calling `execve()`, because this system call was originally designed for this purpose. Its task_t and thread_t (as can be obtained with mach_task_self() and mach_thread_self(), respectively) are exactly those of its parent, as is the vm_map. Only if a later call to execve() results in a Mach-O image activation will a Mach task and thread eventually be created.

The cloneproc() function:

The cloneproc() function is called only on PROC_CREATE_SPAWN or PROC_CREATE_FORK. Because we are interested in a "real" fork, rather than vfork(), it calls forkproc(), but then performs other operations, as well. It proceeds as follows:

- Calls forkproc() on the parent_proc. This function, discussed earlier, returns a child_proc proc_t, which will eventually become the child process's fully populated control block.
- Calls fork_create_child() to create the child process's uthread.

  This function creates the new Mach task (using task_create_internal()) and Mach thread (using thread_create), performs housekeeping (such as setting or clearing the vm_map 32-/64-bitness), and ties the bsd proc_t to the Mach task. The memory_inherit flag is handled by task_create_internal(). If, for some reason this fails, it calls forkproc_free() on the child_proc to deconstruct the new child, effectively a stillborn. Otherwise, the Mach thread_t created will eventually be returned to the caller. These tasks were all previously carried out by procdup(), which has been removed in recent kernels.
- Sets the 64-bitness of the child according to the parent's P_LP64.
- Calls pinsertchild() on the parent_proc and the newly born child_proc. This function ties the two by inserting the child process into the parent's p_children list and also announces the child to the world by inserting it into the allproc list. It has an additional side effect of clearing the P_LIST_INCREATE flag from the child's p_listflag. This flag, set during forkproc(), hides the child from proc_ref_locked().

Loading and Executing Binaries

If a process can be likened to a body, then the binary executing in it can be likened to a brain. Simply giving birth to a new process by fork() would hardly be useful, unless the executing image could be replaced with another, by means of an exec(). The heart of process creation, therefore, lies in loading and executing the binary.

Executable Formats

Somewhat like Linux, the kernel contains designated handlers for various executable formats it supports. Whereas Linux calls these binary formats (or binfmt), OS X calls them execsw. Though very similar in function, in Linux these handlers are more powerful, primarily in that they can be dynamically registered using register_binfmt. Even more powerful in Linux is that registration can be done from within a kernel module, in effect making Linux able to handle any executable format, at least in theory. Figure 13-6 compares the Linux binfmt with the OS X execsw:

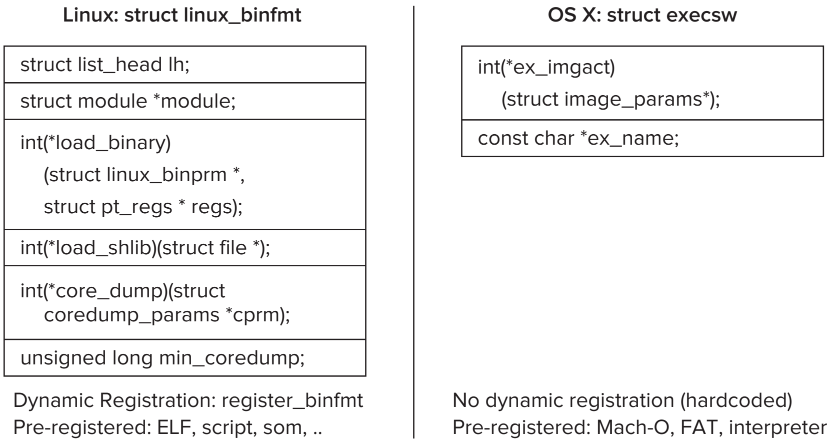


FIGURE 13-6: Comparison of Linux and OS X binary format handlers

By contrast, OS X `execsw` structs are hard-coded. In `bsd/kern/kern_exec.c`, you can find the definition shown in Listing 13-6.

LISTING 13-6: "Image activators" for executable formats in bsd/kern/kern_exec.c

```
/*
 * Our image activator table; this is the table of the image types we are
 * capable of loading.  We list them in order of preference to ensure the
 * fastest image load speed.
 *
 * XXX hardcoded, for now; should use linker sets
 */
struct execsw {
        int (*ex_imgact)(struct image_params *);
        const char *ex_name;
} execsw[] = {
        { exec_mach_imgact,             "Mach-o Binary" },
        { exec_fat_imgact,              "Fat Binary" },
#ifdef IMGPF_POWERPC   /* Deprecated as of Leopard, unsupported in Lion */
        { exec_powerpc32_imgact,        "PowerPC binary" },
#endif  /* IMGPF_POWERPC */
        { exec_shell_imgact,            "Interpreter Script" },
        { NULL, NULL}
};
```

So, although the code does hint at Apple's eventual intent to make executable formats extensible, at present — unlike Linux — they are very much set, offering only the native Mach-O, fat binaries, and the generic script interpreter (all of which were discussed in Chapter 4). This architecture is still fairly extensible; all it takes to extend a binary format is to add another `execsw` entry, but this would mandate kernel recompilation.

Image Parameters

The `image_params` expected by an `execsw` image activator are defined in `bsd/sys/imgact.h` as shown in Listing 13-7.

LISTING 13-7: Image_params for execsw image activators

```
struct image_params {
   user_addr_t     ip_username_fname;          /* argument */
   user_addr_t     ip_user_argv;           /* argument */
   user_addr_t     ip_user_envv;           /* argument */
   int             ip_seg;                 /* segment for arguments */
   struct vnode    *ip_vp;                 /* file */
   struct vnode_attr       *ip_vattr;      /* run file attributes */
   struct vnode_attr       *ip_origvattr;  /* invocation file attributes */
   cpu_type_t      ip_origcputype;         /* cputype of invocation file */
   cpu_subtype_t   ip_origcpusubtype;      /* subtype of invocation file */
   char            *ip_vdata;              /* file data (up to one page) */
   int             ip_flags;               /* IMGPF_* bit flags specifying options */
   int             ip_argc;                /* argument count */
   int             ip_envc;                /* environment count */
   int             ip_applec;              /* apple vector count */
   char            *ip_startargv;          /* argument vector beginning */
   char            *ip_endargv;            /* end of argv/start of envv */
   char            *ip_endenvv;            /* end of envv/start of applev */
   char            *ip_strings;            /* base address for strings */
   char            *ip_strendp;         /* current end pointer */
   int             ip_argspace;         /* remaining space of NCARGS limit(argv+envv) */
   int             ip_strspace;         /* remaining total string space */
   // The following are used for fat binaries
   user_size_t     ip_arch_offset;         /* subfile offset in ip_vp */
   user_size_t     ip_arch_size;           /* subfile length in ip_vp */
   // The following two context;        /* VFS context */
        struct nameidata *ip_ndp;              /* are used for interpreters (!#)
   char            ip_interp_buffer[IMG_SHSIZE];   /* interpreter buffer space */
   int             ip_interp_sugid_fd;             /* fd for sugid script */

   /* Next two fields are for support of architecture translation... */
   char            *ip_p_comm;             /* optional alt p->p_comm */
        struct vfs_context      *ip_vfs_
  current nameidata */
        thread_t        ip_new_thread;          /* thread for spawn/vfork */

        struct label    *ip_execlabelp;         /* label of the executable */
        struct label    *ip_scriptlabelp;       /* label of the script */
        unsigned int    ip_csflags;             /* code signing flags */
        void            *ip_px_sa;
        void            *ip_px_sfa;
        void            *ip_px_spa;
};
```

Architecture Handlers

Up until the release of Lion, OS X still had limited support for multiple architectures — both Intel (i386/x86_64) and PowerPC. This was required for backward compatibility with PPC, which was — until its fall from grace in Tiger and later extinction in Lion — the native architecture of OS X.

During the transition period, support for PPC was handled somewhat similarly to the way interpreters are: When a PPC binary was detected, it was replaced by its corresponding handler — in this case, a binary originally called translate, and then renamed Rosetta.

From the kernel perspective, this meant utilizing a struct `exec_archhandler`, defined in `bsd/machine/exec.h` as follows:

```
struct exec_archhandler {
        char path[MAXPATHLEN];
        uint32_t fsid;
        uint64_t fileid; };
```

The only handler defined in the kernel was Rosetta, defined in `bsd/kern/bsd_init.c` as follows:

```
struct exec_archhandler exec_archhandler_ppc = {
        .path = "/usr/libexec/oah/RosettaNonGrata",
};
```

Support for PPC is now removed, but, in theory, the `exec_archhandler` could be reused some time in the future by Apple. One clever use of it would be to introduce ARM architecture support to OS X, which could enable (with a great deal of translation) running iOS binaries on OS X or vice versa.

Sequence of Steps in Executing an Image

Armed with all this information, we can now piece together, step by step, the process of executing an image, as shown in Figure 13-7.

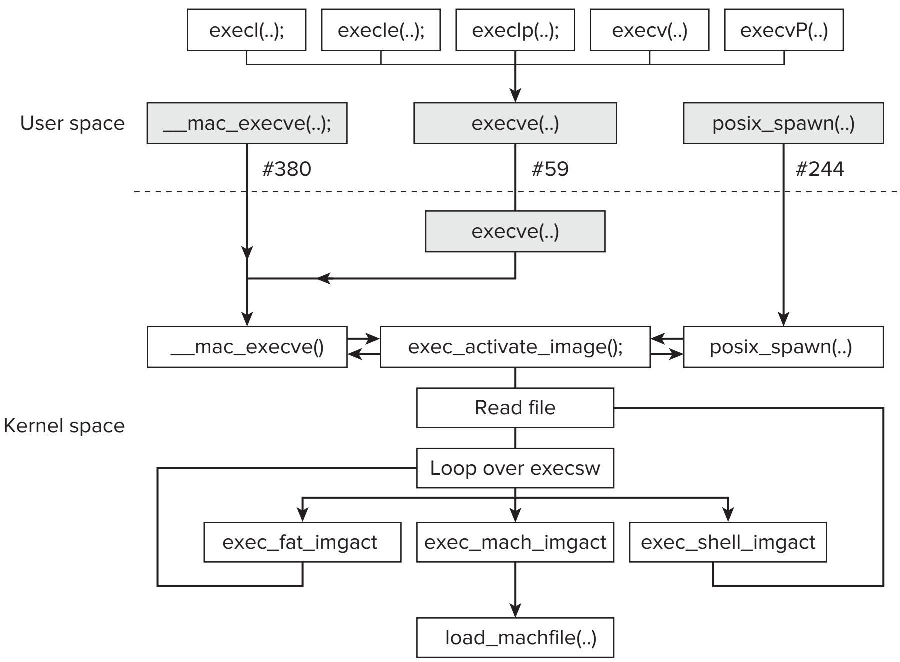


FIGURE 13-7: Flow of the various process execution functions in OS X

User mode has several options in launching a new executable:

- Using the `exec*` family of functions, as listed in Table 13-4.

TABLE 13-4: exec* variants

EXEC* SUFFIX LETTER	DENOTES
`l` (list)	Arguments to the executed program are passed one by one, in a list, with the end of the list specified by a `NULL` argument. Because arguments are passed left to right, the first argument will be at the top of the stack (or, alternatively, in the first register), and the library call can keep inspecting the stack until it finds `NULL`.
`v` (vector)	Arguments to the executed program are passed in a vector — a `char *argv[]`, much like the standard `argv[]` found in C programs.
`exec*` **SUFFIX LETTER**	**DENOTES**
`e` (environment)	The set of environment variables is also passed to the program, as a `char *envp[]`. The program can access these either by declaring `envp[]` as an additional parameter or calling `getenv(3)/setenv(3)`.
`p` (path)	The program name — the first argument — can be specified as a relative name (i.e., with no path separators), in which case the library call will search for the program in the directories listed in the `PATH` environment variable.
`P` (path)	This option is similar to the lowercase `p`, but the library function accepts a second parameter, a `char *` specifying the search path (thereby overriding any setting of the `PATH` environment variable).

All the `exec*` variants in Table 13-4 are really just library function wrappers over the system call, `execve()`, which is why there is no need for an `execve()` library function.

- Calling the `execve()` system call directly, if there is no need for argument list setup code. The `execve()` function, however, is itself only a pass through to `__mac_execve()`.
- Calling `__mac_execve()` directly. This is, as one can guess, an extension, which is not POSIX compliant. `__mac_execve()` differs from the standard `execve()` by only one parameter — an additional field in its second argument, `macp`, which is a mandatory access control (MAC) label. Normally, `execve()` falls right through to it, specifying this label to be `USER_ADDR_NULL`, as shown in Listing 13-8.

LISTING 13-8: execve

```
int
execve(proc_t p, struct execve_args *uap, int32_t *retval)
{
        struct __mac_execve_args muap;
        int err;

        muap.fname = uap->fname;
        muap.argp = uap->argp;
```

```
        muap.envp = uap->envp;
        muap.mac_p = USER_ADDR_NULL;
        err = __mac_execve(p, &muap, retval);

        return(err);
}
```

mac_execve, despite the misleading name, is not an OS X–specific call. It is a part of BSD's MAC architecture, which forms the basis for the seatbelt/sandbox mechanism, as discussed in Chapter 3, and elaborated on from the kernel perspective in Chapter 14.

- Calling posix_spawn() takes care of the fork() operation as well. This system call allows finer granularity of process attribute inheritance from the parent to the child — namely, file descriptors, process group ID, user and group ID, signal masking/behavior, and scheduling.

Eventually, all the image-loading work is performed by exec_activate_image(). This function takes an image_params pointer as an argument and proceeds in the following way:

1. Gets the proc_t structure from the saved VFS context field.
2. execargs_alloc allocates kernel memory for user-space arguments and the first page of image.
3. exec_save_path saves the program path (and fixes up arguments).
4. Gets the image's inode file using the NDINIT macro (in bsd/sys/namei.h) and namei().
5. Ensures thread safety by making sure no other thread in the calling process is calling exit() or the like. It calls proc_transstart() (from bsd/kern/kern_proc.c) to raise the P_LINTRANSIT flag, signifying an image transition is about to occur.
6. Checks the permissions on the inode about to be loaded. These are the standard +x permissions, along with any SetUID/SetGID, which we may need to allow (but not for interpreters).
7. Calls vn_rdwr on the inode to read its first page into memory.
8. Attempts to detect the image type by looping over the execsw[] array. The execsw handlers return one of the following error codes:
    - **Error 0**: The image was handled by the execsw[] handler and loaded. The only handler to return 0, at present, is exec_mach_imgact, the Mach-O image loader.
    - **Error -1**: The image is unrecognized. This is returned by all handlers if the handler cannot handle or does not recognize the image. The next execsw[] handler, if any, will be tried. Otherwise, exec_activate_image propagates the -1 to its caller, which returns an ENOEXEC to user mode.
    - **Error -2**: This error is returned only by the exec_fat_imgact and is returned if image is encapsulated (i.e., a fat binary). In this case, exec_fat_imgact also retrieves the preferred binary architecture from the fat archive, and this step is retried.
    - **Error -3**: This error is reserved for the exec_shell_imgact and is returned if the image is an interpreter. In this case, exec_shell_imgact redirects to the inode of the interpreter file (that is, it loads the path specified after the !#), and the process is retried from step 6.

Looking at Figure 13-7, you can clearly see that all image-loading paths either terminate with an error or eventually result in a Mach image. Fat binaries are merely treated as archives of other images, and interpreters would redirect to load the interpreter first, which again brings us to the Mach image case. The following section covers this case in depth, picking up where Chapter 4 left off.

The book's website has a detailed experiment on extending XNU to recognize other types of binaries.

Mach-O Binaries

The Mach-O loading logic in XNU is still largely the same as it was in its inception back in 1988 in NeXT. Apple has made a few changes over the years, most notably for code decryption, but the base of the Mach-O file format has changed very little over the years.

Apple has wrapped that logic by means of `exec_mach_imgact()`, which as the previous section described, is the registered handler for Mach binaries. This function first reads the Mach header, and then parses its architecture (32-bit or 64-bit) and flags. The function refuses `DYLIB` and `BUNDLE` files — those are maintained by `dyld(1)` in user mode. It then goes on to `apply posix_spawn()` arguments, if any. After this, it makes sure the binary is right for the current architecture by grading the binary.

Before the actual loading of the Mach file commences, the function checks its `imgp` flags for `IMGPF_SPAWN` and the `bsdthread_info uu_flag` for `UT_VFORK`. If any of these are true, it calls `fork_create_child()` (discussed earlier in this chapter, as part of the fork operation) to create a new Mach task and thread for this process. This is required because neither of these is created in a `vfork()`.

The main function handling the loading of Mach-O is `load_machfile()` in `bsd/kern/mach_loader.c`.

This function is defined as shown in Listing 13-9.

LISTING 13-9: load_machfile() , from bsd/kern/mach_loader.c

```
load_return_t load_machfile(
    struct image_params     *imgp,    // Image parameters as set by exec_mach_imgact
    struct mach_header      *header,  // Mach-O header (overlaid on imgp->ip_vdata)
    thread_t                thread,   // current_thread();
    vm_map_t                new_map,  // get_task_map() for vfexec or spawn, else NULL
    load_result_t           *result); // out parameter, returning load operation data
```

The `load_machfile()` function is responsible for setting up the memory map that will eventually be loaded by the various `LC_SEGMENT` commands. It proceeds as follows:

1. If `new_map` is a `NULL_MAP` or the `ipgp` flags state `IMGPF_SPAWN`, `load_machfile()` creates a new `vm_map` by first creating a new `pmap` using `pmap_create()`, and then `vm_map_create()`. Otherwise, use the `new_map` parameter as the `vm_map`.
2. Harden virtual memory security first. This is done in two steps:
    a. Disallow the execution of data segments. This step is similar to Window's Data Execution Prevention (DEP) and is set if the Mach header flags state `MH_NO_HEAP_EXECUTION` and unless the `imgp` flags specifically set `IMGPF_ALLOW_DATA_EXEC`.

   b. Set up address space layout randomization. This step generates a random `aslr_offset` slide value for the image unless the `imgp` flags specifically set `IMGPF_DISABLE_ASLR`.

3. Call `parse_machfile`, which does the hard work of actually parsing the load commands.
4. If parsing fails, forget it — `vm_map_deallocate()` the map, if created. Return with failure.
5. Otherwise, if a new map has been created, commit to the new map, using `swap_task_map()`, which places the new map as the active one, and then `vm_map_deallocate()` the previous map. This step also involves terminating the old task and any threads it might contain (because their memory is invalid, anyway).

The heart of `load_machfile` is `parse_machfile`. This function is defined as shown in Listing 13-10.

LISTING 13-10: parse_machfile

```
load_return_t
parse_machfile(
        struct vnode       *vp,     // vnode pointer from imgp
        vm_map_t            map,    // map, as initialized by load_machfile
        thread_t            thread, // thread, from load_machfile
        struct mach_header *header, // header, from load_machfile
        off_t               file_offset, // Architecture offset
        off_t               macho_size,  // Architecture binary size
        int                 depth, // recursion level. Started at 0.
        int64_t             aslr_offset, // generated by load_..
        load_result_t       *result);
```

`load_machfile()` calls `parse_machfile`, with most of the parameters copied directly from its own arguments (`thread` and `header`), from its `imgp` (`vp`, `file_offset`, and `macho_size`), or from values it sets up (`map`, `depth` set to 0, and `slide`).

The parsing operation is a potentially recursive one, which is why it is started with depth set to 0, and incremented on subsequent calls. The maximum depth allowed is 6, after which a `LOAD_FAILURE` is returned. The `parse_machfile()` function proceeds as follows:

1. Checks header to determine 64-bitness.
2. Fails if depth is greater than 6.
3. Validates architecture mask, or return `LOAD_BADARCH`.
4. Switches on the header's filetype field:
    - Allows `MH_OBJECT`, `EXECUTE`, or `PRELOAD` only for depth of 1.
    - Allows `MH_FVMLIB` or `MH_DYLIB` only for a depth greater than 1.
    - Allows `MH_DYLINKER` only for a depth of exactly 2.
    - Otherwise, fails (return `LOAD_FAILURE`).
5. Maps all the load commands into memory by rounding to page size and by calling `vn_rdwr()`, or fail with `LOAD_IOERROR`.
6. If the header flags state `MH_PIE`, or `dyld` is being loaded, applies the `aslr_offset`.

7. Performs three passes. In each, while there are still load commands to execute, switches on each load command, and act on it:
   - On LC_SEGMENT/LC_SEGMENT_64, load_segment(), mapping the segment directly into memory according to the segment directions.
   - On LC_UNIXTHREAD, load_unixthread(), which itself calls load_threadentry() and load_threadstate().
   - On LC_LOAD_DYLINKER, if in pass 3 and depth is exactly 1, saves it (in the dlp variable).
   - On LC_UUID, copy the UUID into the result.
   - On LC_CODE_SIGNATURE, if in pass 1, load_code_signature() but do not validate yet.
   - On LC_ENCRYPTION_INFO, set_code_unprotect() (using the Apple Protect Pager, discussed in Chapter 11). If the decryption is unsuccessful, kill the poor process.
   - All other load commands are ignored, being the responsibility of the DYLINKER (dyld).
8. If, after the three passes, there is a saved dynamic linker command (in dlp), load the dynamic linker into the new map, possibly adjusting by the ASLR offset. The load_dylinker() function recursively calls parse_machfile().

When parse_machfile() is successful, it sets its load_result_t parameter, which is then passed back to load_machfile and, eventually, to the caller, as shown in Listing 13-11.

LISTING 13-11: load_result returned from load_machfile

```
typedef struct _load_result {
        user_addr_t             mach_header;
        user_addr_t             entry_point;          // set by load_unixthread()
        user_addr_t             user_stack;           // set by load_unixthread()
        mach_vm_address_t       all_image_info_addr;
        mach_vm_size_t          all_image_info_size;
        int                     thread_count;
        unsigned int
                /* boolean_t */ unixproc        :1, // by load_unixthread()
                                dynlinker       :1, // by load_dylinker()
                                customstack     :1, // by load_unixthread()
                                validentry      :1, // by load_segment()
                         /* unused */           :0;
        unsigned int          csflags;  // code-signing flags, by load_code_signature();
        unsigned char   uuid[16];       // parse_machfile, on LC_UUID
        mach_vm_address_t       min_vm_addr;
        mach_vm_address_t       max_vm_addr;
} load_result_t;
```

If load_machfile() returns success, exec_mach_imgact picks up after it and does additional housekeeping. Specifically, it performs the following actions:

- Sets the ulimit -m (MEM_LOCK) by calling vm_map_set_user_wire_limit.
- Sets code-signing flags:
  - CS_HARD: Refuse to load invalid pages
  - CS_KILL: Kill process if any pages are invalid
  - CS_EXEC_*: Same as previous, but follow execve(2)

    (This does not enforce anything yet: The actual code-signing enforcement is called from Mach's VM page fault handler, which calls cs_invalid_page (bsd/sys/kern_proc.c) to enforce the policy)
- Sets up system memory areas and a custom stack, if any
- Sets the entry point (the register state from LC_UNIXTHREAD)
- Sets the process new name (p->comm)
- Delivers any delayed signals

PROCESS CONTROL AND TRACING

As discussed in Chapter 5, Mach offers extensive tracing facilities, first and foremost of them being DTrace. Chapter 5 discounted another mechanism, ptrace(2), which is (deliberately) only partially functional in OS X and iOS.

ptrace (#26)

BSD and other UNIX systems offer a one-stop system call called ptrace(2) to support process tracing and debugging. Much like an ioctl(2), it is a highly generic call that you can use for multiple operations. It is defined as follows:

```
int     ptrace(int request, pid_t pid, caddr_t addr, int data);
```

The caller needs to specify a request (one of the values in Table 13-5) and a process ID to which this request will apply. The caller may also specify two additional arguments — addr and data — that are dependent on the request.

This system call is highly useful for both debugging and reverse engineering, and in Linux, for example, is used by gdb, the system call tracer (strace) and the library call tracer (ltrace).

Although ptrace(2) is available on XNU and its prototype is the same as in other systems, its functionality is greatly reduced, not to say crippled. <sys/ptrace.h> defines the standard request codes (which are slightly different from those you may know from Linux), but XNU only supports those you see in Table 13-5, which are used for debugger program tracing.

TABLE 13-5: ptrace request codes supported by XNU

PTRACE REQUEST (LINUX EQUIVALENT)	USED FOR
`PT_TRACE_TRACEME` (TRACEME)	Declaring tracing by the process's parent.
`PT_CONTINUE` `(CONT)`	Continuing on `next (addr == 1)` or other (specify `addr`) instruction. Also, optionally deliver signal specified by *`data`*.
`PT_KILL` `(KILL)`	Killing the target process. Equivalent to `PT_CONTINUE(...., SIG_KILL).`
`PT_STEP` `(SINGLESTEP)`	Single-stepping the target process.
`PT_ATTACH` `(ATTACH)`	Specifying the target PID to attach to in order to start tracing. Must be process owner (same UID) or root.
`PT_DETACH` `(DETACH)`	Specifying target PID to detach from in order to stop tracing. Traced process is freed to continue on its own.
`PT_DENY_ATTACH` (N/A)	Apple proprietary: Specified by a process that does not want to be meddled with (all arguments are ignored). iTunes and other Apple processes use this.

Unlike Linux, wherein the true power of `ptrace` lies in being able to read (and write) a foreign process memory, XNU's `ptrace` implementation (in `bsd/kern/mach_process.c`) silently ignores these options. Thanks to the Mach APIs, however, achieving comparable functionality is possible, as shown in Table 13-6.

TABLE 13-6: ptrace request codes that are unavailable, but can be emulated using Mach APIs

PTRACE REQUEST (LINUX EQUIVALENT)	USED FOR	EMULATED BY
`PT_READ_I` `(PEEKTEXT)`	Reading an integer from the process `I` (instruction) space.	
`PT_READ_D` `(PEEKDATA)`	Reading an integer from the process `D` (data) space.	`vm_map_read()`
`PT_READ_U` `(PEEKUSER)`	Reading from the process `U` (user) space (registers).	

PTRACE REQUEST (LINUX EQUIVALENT)	USED FOR	EMULATED BY
`PT_WRITE_I` (`POKETEXT`)	Writing an integer from the process `I` (instruction) space.	
`PT_WRITE_D` (`POKEDATA`)	Writing an integer from the process `D` (instruction) space.	`vm_map_write()`
`PT_WRITE_U` (`POKEREG`)	Writing to the process `U` (user) space.	
`PT_GETREGS` (`GETREGS`)	Obtaining thread register state.	`thread_get_state()`
`PT_SETREGS` (`SETREGS`)	Modifying thread register state .	`thread_set_state()`

proc_info (#336)

The undocumented `proc_info` system call was described in Chapter 5, and is mentioned here again for the random access reader. The system call, well deserving of its own file (`bsd/kern/proc_info.c`), is a wonderfully useful one, providing an amalgam of many diagnostic and control functions. Most of these functions indeed relate to process and thread information, yet it seems that Apple's developers decided to throw in some additional functionality. One such example is call number 4, `proc_kernmsgbuf` (available from user mode's `libproc` as `proc_kmsgbuf)`, which displays the kernel's message buffer, thereby having little to do with processes and threads. User mode's `libproc` exports most, but not all of `proc_info`'s functionality. Nifty features like setting process and thread names (akin to Linux's `prctl(2) PR_SET_NAME`), remain virtually undocumented (though available via `LibC`'s `pthread_setname_np`).

Policies

OS X and iOS support the notion of I/O and execution policies. This is somewhat of a difficult choice of word, however, since the main use of policies is in the context of the Mandatory Access Control Framework (MACF), discussed previously in Chapter 3, and re-examined in the Chapter 14. In the context of this discussion, however, a policy is a set of execution rules relating primarily to performance, and not to security.

iopolicysys (#322)

The proprietary `iopolicysys` system call has been available since Leopard, but remains hidden among the many system calls of XNU. It is used by LibSystem's (technically, `libC`'s) `get/set_iopolicy_np(3)`, and the manual page provides ample documentation.

The only I/O policy Apple provides at this time is `IOPOL_TYPE_DISK`, for local device I/O, and the scope a policy can be applied on is either that of the thread, or the entire process. The policy can have values of `NORMAL` (best-effort), `THROTTLE` (bandwidth-restricted), or `PASSIVE` (on behalf of other processes).

process_policy (#323)

Another virtually undocumented system call is `process_policy`. This is a new addition in Lion and iOS that allows the enforcement of execution policies on processes. The currently defined policies, from `bsd/sys/process_policy.h`, are shown in Table 13-7, but the implementation in Lion is partial. Unlike other header files in `bsd/sys`, this header is not exported to user mode. The main client of the system call is (as with `proc_info`) `libproc`. The various functions, however, are not publicly declared in `<libproc.h>` which concentrates on the `proc_info` wrappers, and instead declared in the non-exported `libproc_internal.h`.

You can get a good idea of the system call's usage by looking at `bsd/kern/process_policy.c`, or downloading Darwin's `LibC` and looking at `Darwin/libproc.c` and the `libproc_internal.h` header. Doing so will reveal a discrepancy between `LibC` and XNU, as Apple has left out some of the iOS code (`#ifdef TARGET_OS_EMBEDDED`) hinting at features and flags not supported in OS X's XNU. The open source (and, therefore, OS X) implementation of this system call is woefully incomplete (and even includes a typo or two in function names!)

TABLE 13-7: Process policies

PROCESS POLICY	SCOPE
`PROC_POLICY_BACKGROUND`	Handles background execution of App. Naturally more applicable in iOS, where SpringBoard uses this for applications when the home button is pressed.
`PROC_POLICY_HARDWARE_ACCESS`	Controls access to disk, GPU, network, and CPU. Inert on OS X.
`PROC_POLICY_RESOURCE_STARVATION`	Controls process behavior when the system is extremely low on resources (e.g. VM Pressure).
`PROC_POLICY_RESOURCE_USAGE`	Sets limits on resource usage. The code hints at resources like wired and virtual memory, network, disk, and even power, but in practice the only resource enabled is CPU utilization.
`PROC_POLICY_APP_LIFECYCLE`	Sets various attributes of the lifecycle, such as PID binding, device state, and others. Non-existent in OS X's XNU.
`PROC_POLICY_APPTYPE`	Type of app — Active, Inactive, background, non-UI.

Process Suspension/Resumption

Mac OS and iOS occasionally depart from the POSIX APIs to offer specific systems calls. Process suspension and resumption are excellent (system calls #433 and 434) examples of this (The system calls have been renumbered from #430, #431 in Snow Leopard to their present numbers in Lion and iOS).

The idea of suspending a process, effectively stopping it for an indefinite amount of time during its execution until resumed, is not new to UNIX users, who are likely familiar with the STOP and TSTP signals (the former more commonly known to users as Ctrl-Z). This, however, is not what suspension is about in OS X and iOS: As early as Snow Leopard, XNU offered — in addition to the signals — the custom system calls to enable this feature.

Initially, these system calls were no more than simple wrappers over the Mach APIs of `task_suspend()` and `task_resume()`. In iOS 5, however, they were integrated with the Mach `default_freezer` (discussed in the Mach VM chapter) and the process hibernation mechanism (discussed in Chapter 14). This enables a process to be selectively frozen and thawed by means of the system calls, which is a decision usually left up to iOS's launcher, SpringBoard. In Lion the integration is still `#ifdef`'ed out, as it requires the `CONFIG_FREEZE` option. Disassembly of iOS 5 and later shows this feature is very much enabled in it.

SIGNALS

Mach already provides low-level handling of traps by means of the exception mechanism, which was previously discussed in Chapter 11. The BSD layer builds its signal handling on top of the exception primitives. Hardware-generated signals are caught by the Mach layer and translated into their corresponding UNIX signals. In order to maintain a unified mechanism, operating system and user-generated signals are actually converted into Mach exceptions first, and then into signals.

The UNIX Exception Handler

When the first BSD (and user mode process) is started (by `bsdinit_task()` in `bsd/kern/bsd_init.c`) the function also sets up a special Mach kernel thread called `ux_handler` by calling `ux_handler_init` from `bsd/uxkern/ux_exception.c`, as shown in Listing 13-12.

LISTING 13-12: ux_handler_init in bsd/uxkern/ux_exception.c

```
void
ux_handler_init(void)
{
        thread_t        thread = THREAD_NULL;
        ux_exception_port = MACH_PORT_NULL;  // global, defined ibid.

       // spin off ux_handler in a new Mach thread
        (void) kernel_thread_start((thread_continue_t)ux_handler, NULL, &thread);
        thread_deallocate(thread);

       // Lock the process list (not allowing any processes to be created,
```

continues

LISTING 13-12 *(continued)*

```
        // including bsdinit_task(), which called us) until ux_exception_port
        //is registered by ux_handler
         proc_list_lock();
         if (ux_exception_port == MACH_PORT_NULL)  {
           (void)msleep(&ux_exception_port, proc_list_mlock, 0, "ux_handler_wait", 0);
         }

         proc_list_unlock();

}
```

Only after `ux_handler_init` returns does `bsdinit_task()` go on to register the `ux_exception_port`, as shown in Listing 13-13.

LISTING 13-13: bsdinit_task() exception handling

```
void bsdinit_task(void)
{
    proc_t p = current_proc();
    struct uthread *ut;
    thread_t thread;

    process_name("init", p); // set our process name to "init" (this gets changed later
                             // in load_init_program() to launchd)

    ux_handler_init();       // spin off Unix exception handler thread

    thread = current_thread();

   // when ux_handler_init() returns, ux_handler() is executing in a separate thread
   // and registers the ux_exception_port.

    (void) host_set_exception_ports(host_priv_self(),
                                    EXC_MASK_ALL & ~(EXC_MASK_RPC_ALERT),
                                    (mach_port_t) ux_exception_port,
                                    EXCEPTION_DEFAULT| MACH_EXCEPTION_CODES,
                                    0);

    ut = (uthread_t)get_bsdthread_info(thread);

    bsd_init_task = get_threadtask(thread);
    init_task_failure_data[0] = 0;

#if CONFIG_MACF
        mac_cred_label_associate_user(p->p_ucred);
        mac_task_label_update_cred (p->p_ucred, (struct task *) p->task);
#endif

    // go on to load the init program, launchD.
    load_init_program(p);

}
```

By calling host_set_exception_ports, the bsdinit_task() redirects all Mach exception messages to ux_exception_port, which is held by the ux_handler() thread. True to the Mach paradigm, exception handling for PID 1 will be handled out of process by ux_handler(). Because all subsequent user mode processes are descendants of PID 1, they will automatically inherit the exception port, thereby assigning ux_handler() responsibility for every Mach exception that occurs in a UNIX process on the system.

ux_handler() is a fairly simple function, which makes sense given the amount of exceptions it needs to process. As one would expect, it sets up the ux_handler_port on entry, and then enters an endless Mach message loop. The message loop receives the Mach exception messages, and then calls mach_exc_server() to handle the exception, as shown in Listing 13-14.

LISTING 13-14: ux_handler(), in bsd/uxkern/ux_exception.c

```
void
ux_handler(void)
{
    task_t              self = current_task();
    mach_port_name_t    exc_port_name;
    mach_port_name_t    exc_set_name;

    /* self->kernel_vm_space = TRUE; */
    ux_handler_self = self;


    /*
     *  Allocate a port set that we will receive on.
     */
    if (mach_port_allocate(get_task_ipcspace(ux_handler_self),
        MACH_PORT_RIGHT_PORT_SET,
        &exc_set_name) != MACH_MSG_SUCCESS)
            panic("ux_handler: port_set_allocate failed");

    /*
     *  Allocate an exception port and use object_copyin to
     *  translate it to the global name.  Put it into the set.
     */
    if (mach_port_allocate(get_task_ipcspace(ux_handler_self),
        MACH_PORT_RIGHT_RECEIVE,
        &exc_port_name) != MACH_MSG_SUCCESS)
        panic("ux_handler: port_allocate failed");
    if (mach_port_move_member(get_task_ipcspace(ux_handler_self),
                        exc_port_name,  exc_set_name) != MACH_MSG_SUCCESS)
        panic("ux_handler: port_set_add failed");
    if (ipc_object_copyin(get_task_ipcspace(self), exc_port_name,
                        MACH_MSG_TYPE_MAKE_SEND,
                        (void *) &ux_exception_port) != MACH_MSG_SUCCESS)
                panic("ux_handler: object_copyin(ux_exception_port) failed");

    proc_list_lock();
    thread_wakeup(&ux_exception_port);
    proc_list_unlock();

    /* Message handling loop. */
```

continues

LISTING 13-14 *(continued)*

```
    // No problem with getting into an endless loop here, since ux_handler() runs in its
    // own thread, and the mach_msg_receive() function blocks anyway.
    for (;;) {
        // inline structure definitions make for great readability.. This
        // is likely a vestige of MIG's automatic code generation
        struct rep_msg {
                mach_msg_header_t Head;
                NDR_record_t NDR;
                kern_return_t RetCode;
        } rep_msg;
        struct exc_msg {
                mach_msg_header_t Head;
                /* start of the kernel processed data */
                mach_msg_body_t msgh_body;
                mach_msg_port_descriptor_t thread;
                mach_msg_port_descriptor_t task;
                /* end of the kernel processed data */
                NDR_record_t NDR;
                exception_type_t exception;
                mach_msg_type_number_t codeCnt;
                mach_exception_data_t code;
             /* some times RCV_TO_LARGE probs */
                char pad[512];
        } exc_msg;
        mach_port_name_t        reply_port;
        kern_return_t    result;

        exc_msg.Head.msgh_local_port = CAST_MACH_NAME_TO_PORT(exc_set_name);
        exc_msg.Head.msgh_size = sizeof (exc_msg);

        result = mach_msg_receive(&exc_msg.Head, MACH_RCV_MSG,
                             sizeof (exc_msg), exc_set_name,
                             MACH_MSG_TIMEOUT_NONE, MACH_PORT_NULL,
                             0);

        if (result == MACH_MSG_SUCCESS) {
            reply_port = CAST_MACH_PORT_TO_NAME(exc_msg.Head.msgh_remote_port);
          // mach_exc_server will call mach_exception_raise(), which will be caught
          // by mach_catch_exception_raise() - where the signal handling logic is.
            if (mach_exc_server(&exc_msg.Head, &rep_msg.Head)) {
                result = mach_msg_send(&rep_msg.Head, MACH_SEND_MSG,
                   sizeof (rep_msg),MACH_MSG_TIMEOUT_NONE,MACH_PORT_NULL);
                if (reply_port != 0 && result != MACH_MSG_SUCCESS)
                   mach_port_deallocate(get_task_ipcspace(ux_handler_self), reply_port);
            }

        }
        else if (result == MACH_RCV_TOO_LARGE)
                /* ignore oversized messages */;
      else // any other result is unexpected, and thereby constitutes a panic
                panic("exception_handler");
    } // end message loop
} // end ux_handler()
```

The messages are caught by `catch_mach_exception_raise()`, defined in the same file as shown in Listing 13-15

LISTING 13-15: catch_mach_exception_raise, in bsd/uxkern/ux_exception.c

```
kern_return_t
catch_mach_exception_raise(
        __unused mach_port_t exception_port,
        mach_port_t thread,
        mach_port_t task,
        exception_type_t exception,
        mach_exception_data_t code,
        __unused mach_msg_type_number_t codeCnt
)
{
    mach_port_name_t thread_name = CAST_MACH_PORT_TO_NAME(thread);
    mach_port_name_t task_name = CAST_MACH_PORT_TO_NAME(task);
   ..
if (th_act != THREAD_NULL) {
            /*
             *  Convert exception to unix signal and code.
             */
            ux_exception(exception, code[0], code[1], &ux_signal, &ucode);

            ut = get_bsdthread_info(th_act);
            sig_task = get_threadtask(th_act);
            p = (struct proc *) get_bsdtask_info(sig_task);

            /* Can't deliver a signal without a bsd process */
            if (p == NULL) {
                    ux_signal = 0;
                    result = KERN_FAILURE;
            }
  if (code[0] == KERN_PROTECTION_FAILURE &&
                ux_signal == SIGBUS) {
               // handle specifically stack overflow
               …
       }
/*
 *  Send signal.
             */
            if (ux_signal != 0) {
                        ut->uu_exception = exception;
                        //ut->uu_code = code[0]; // filled in by threadsignal
                        ut->uu_subcode = code[1];
                        threadsignal(th_act, ux_signal, code[0]);
            }

            thread_deallocate(th_act);
     ..
     /*
    *  Delete our send rights to the task port.
     */
    (void)mach_port_deallocate(get_task_ipcspace(ux_handler_self), task_name);
..
}
```

At a higher level, the flow can be pictured roughly as shown in Figure 13-8.

Hardware-Generated Signals

Hardware-generated signals begin their life as processor traps. These are, naturally, platform specific. `ux_exception` (`bsd/uxkern/ux_exception.c`) is responsible for translating traps into signals. To handle the machine-specific cases, it tries `machine_exception` (`bsd/dev/i386/unix_signal.c`). If the function cannot convert the signal, `ux_exception` handles generic cases.

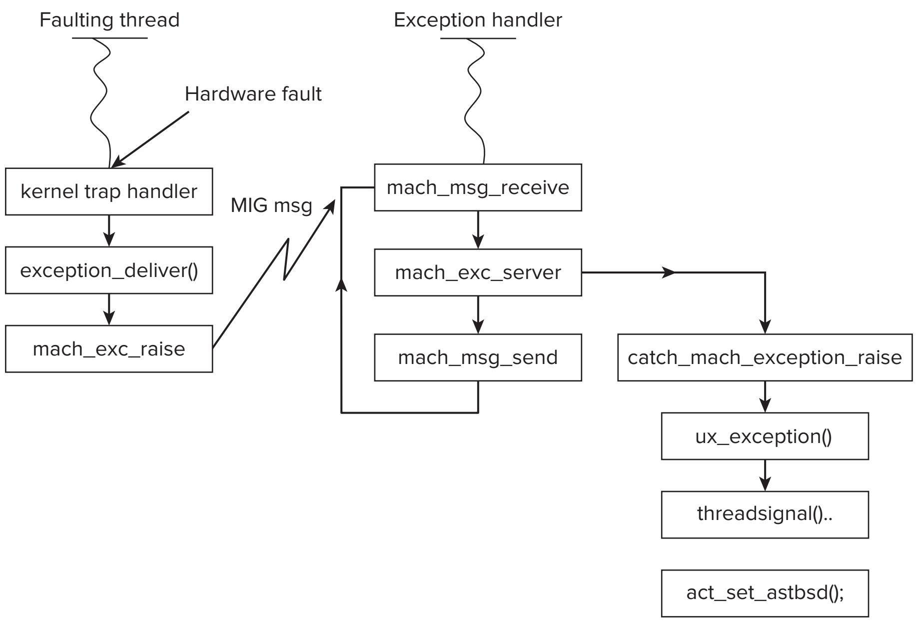


FIGURE 13-8: Mach Exception handling and conversion to UNIX signals

The Mach exceptions previously discussed in Chapter 11 are mapped to UNIX signals as shown in Table 13-8:

TABLE 13-8: Mapping Mach exceptions to UNIX S

MACH EXCEPTION	UNIX SIGNAL
`EXC_BAD_INSTRUCTION`	`ILL`
`EXC_EMULATION`	`EMT`
`EXC_BREAKPOINT`	`TRAP`
`EXC_ARITHMETIC`	`FPE`
`KERN_BAD_ACCESS`	`SEGV(KERN_INVALID_ADDRESS)` `BUS (else)`
`EXC_SOFTWARE`	`SYS (EXC_UNIX_BAD_SYSCALL)` `PIPE (EXC_UNIX_BAD_PIPE)` `ABRT (EXC_UNIX_ABORT)` `KILL (EXC_SOFT_SIGNAL)`

Software-Generated Signals

When the signal is not generated by hardware, it actually begins its life as a signal generated by one of two APIs: `kill(2)` or `pthread_kill(2)`. These functions send a signal to a process or a thread, respectively. `kill(2)` accepts a PID argument, which is interpreted as shown in Table 13-9:

TABLE 13-9 Kill arguments and their meanings

KILL ARGUMENT	MEANING
Greater than 0	Process identifier. Kill invokes `psignal(p,signum)`
0	Current process group. Kill invokes `killpg1()` with `pgid = 0`
-1	All processes (broadcast). Kill invokes `killpg1()` with `pgid = 0` and `all = 1`
Less than -1	Process group. Kill invokes `killpg1()` with `pgid = -(pid)` (i.e., value flipped to positive)

`killpg1()` uses the process list iteration functions (described previously in this chapter) to walk either the global process list, or the one associated with the `pgrp`. The filter function employed is `killpg1_pgrpfilt`, which filters out PIDs less than 2 (thus making the `init` process, `launchd`, unsignalable) , any zombie processes or processes marked as system. The callout function used is `killpg1_callback()`, which calls `cansignal()` to check kill permissions, and then goes on to `call psignal()` if `cansignal()` returns `TRUE` on the process in question. This flow is depicted in Figure 13-9:

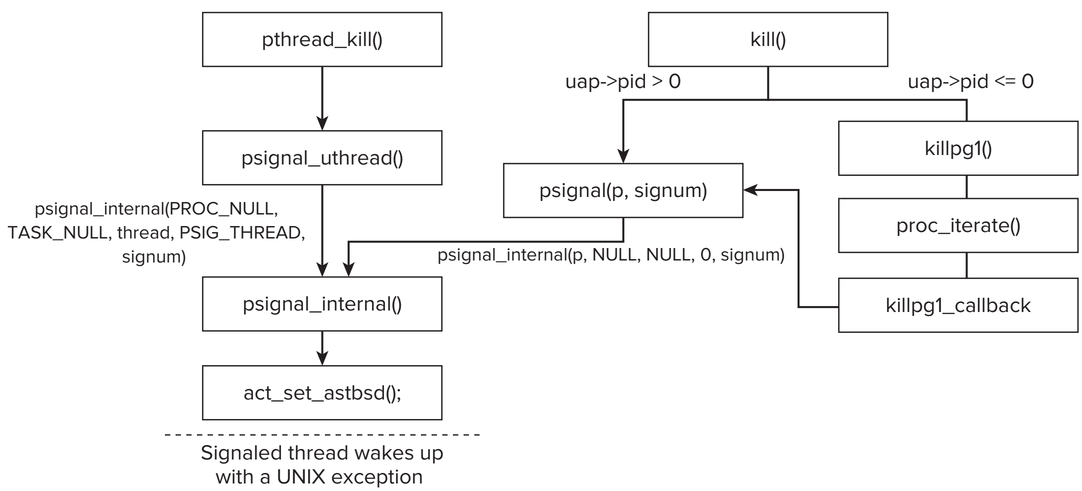


FIGURE 13-9: Handling signals from user mode

Signal Handling by the Victim

Whether it's a hardware-generated or other signal, both execution paths end in `act_set_bsdast()`. This causes the process being signaled to wake up (more accurately, one of its threads does) with its execution redirected to `ast_taken()` (see Chapter 11), which in turn calls the `bsd_ast()`. The flow of `bsd_ast` is shown in Figure 13-10.

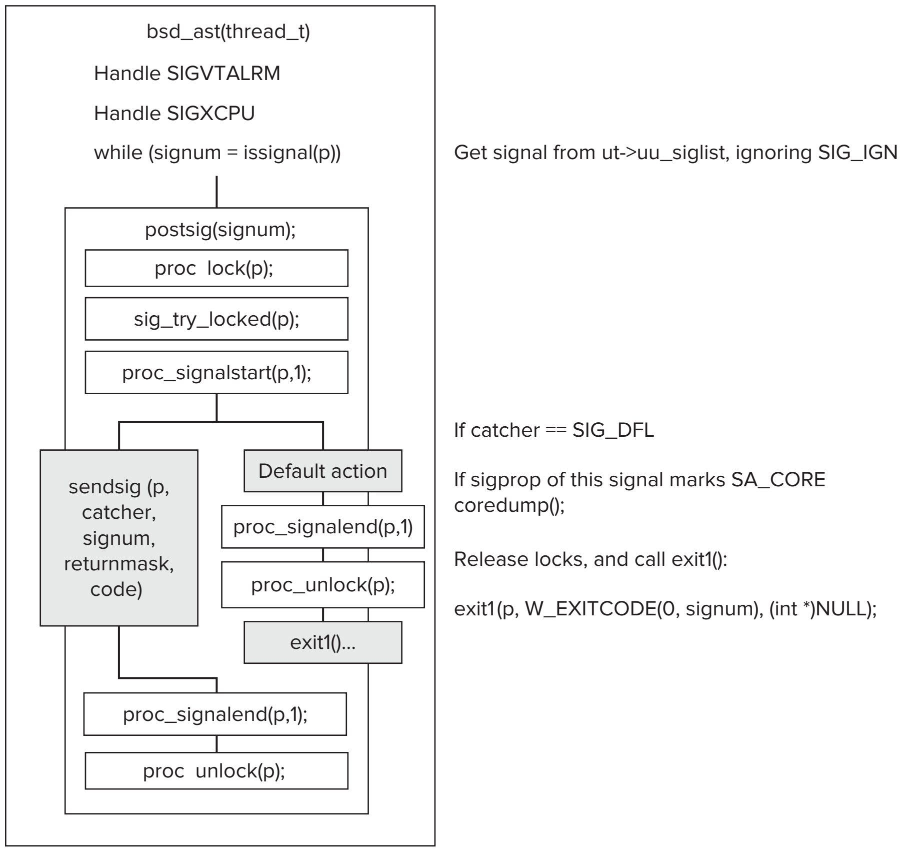


FIGURE 13-10: Signal handling by the signaled process/thread, from bsd_ast()

SUMMARY

This chapter described in depth the BSD layer, which serves as XNU's primary interface to user mode. This layer presents a standardized POSIX-compliant interface, and a developer can expect to find everything present in other UNIX SUSv3 systems. Although OS X implements BSD on top of Mach, the developer remains blissfully unaware of the Mach internals, and instead deals with the higher-level abstractions of processes and threads, rather than the low-level primitives. The next chapter will further discuss signals, IPC objects, and devices.

REFERENCES

1. McKusick, Marshall Kirk, Keith Bostic, Michael J. Karels, and John S. Quarterman, *Design & Implementation of the 4.3BSD UNIX Operating System* (old, but still very comprehensive). AW UNIX and Open Systems Series. ISBN: 978-0132317924
2. UNIX03 specification, `http://www.unix.org/unix03.html`

14

Something Old, Something New: Advanced BSD Aspects

XNU inherits much more than process and threads objects from BSD. The user mode POSIX APIs for shared memory and memory management, as well as signals, all wrap the underlying Mach abstractions covered in the previous chapters.

Apple has made significant improvements to BSD in certain areas, most notably TrustedBSD's Mandatory Access Control framework, which (as discussed in Chapter 3) serves as the substrate for Apple's sandbox and policy control modules.

This chapter picks up where its predecessor left off. We examine first BSD's memory management, as well as Apple's unique Memorystatus mechanism (known as Jetsam). We then focus on the kernel perspective of those features previously touched on in Chapter 3: Sysctl, work queues, and the Mandatory Access Control Framework. The chapter explains what happens behind the scenes in all these OS X and iOS specific technologies that are used from user mode.

MEMORY MANAGEMENT

As you saw in Chapter 12, virtual memory management is carried out by the Mach layer, which controls the pagers and exports the various `vm_` and `mach_vm_` messages to user mode. User mode developers, however, mostly know the standard POSIX calls, so the Mach calls need to be encapsulated. Likewise, the BSD layer itself uses its own memory management functions.

POSIX Memory and Page Management System Calls

POSIX offers the programmer several APIs for managing and maintaining tighter control over virtual memory pages. XNU implements the calls shown in Table 14-1, which are all implemented in `bsd/kern/kern_mman.c` (corresponding to `<sys/mman.h>`).

TABLE 14-1: Page Management System Calls in POSIX

#	SYSTEM CALL	USE
197	`void * mmap(void *addr,` `size_t len,` `int prot,` `int flags,` `int fd,` `off_t offset);`	Maps a region of memory Calls `vm_map_enter_mem_object()` for anonymous (*flags* `|= MAP_ANON`) or `vm_map_enter_mem_object_control()` for file (`flags |= MAP_FILE`) mapping
73	`int munmap(void *addr, size_t len);`	Calls `mach_vm_deallocate()`
75	`int madvise(void *addr,` `size_t len,` `int advice);` `(also: posix_madvise)`	Provides non-obligating advice to OS as to how the memory pages from *addr* to *addr+len* will be accessed: Invokes `mach_vm_behavior_set` and translates advice. The `POSIX MADV_*` constants are changed to corresponding `VM_BEHAVIOR_*` constants.
78	`int mincore` `(caddr_t addr,` `size_t len,` `char *vec);`	Returns vector *vec* specifying residency flags of pages containing *addr* to *addr+len*. Flags are: `MINCORE_INCORE` — resident `MINCORE_REFERENCED` — referenced by process `MINCORE_MODIFIED` — modified by process `MINCORE_REFERENCED_OTHER` — referenced externally `MINCORE_MODIFIED_OTHER` — modified externally Calls `mach_vm_page_query()`
250	`int minherit` `(caddr_t addr,` `size_t len,` `int inherit);`	Sets inheritance of pages containing *addr* to *addr+len* to `VM_INHERIT_NONE`, `_COPY`, or `_SHARE` Calls `mach_vm_inherit()`

#	SYSTEM CALL	USE
203 204	`int mlock (const void *addr, size_t len);` `int munlock (const void *addr, size_t len);`	Locks/unlocks virtual pages containing *addr* to *addr+len* in physical memory — that is, makes them resident (wired) Invokes `vm_map_wire()`
324 325	`int mlockall(void);` `int munlockall(void);`	Locks/unlocks all virtual pages of process. *Not* supported by OS X `(- ENOSYS)`
74	`int mprotect (void *addr, size_t len, int prot);`	Sets *prot* flags on virtual pages containing *addr* to *addr+len*. Flags can be: `PROT_NONE`: `---` `PROT_READ`: `r--` `PROT_WRITE`: `-w-` `PROT_EXEC`: `--x` Invokes `mach_vm_protect()`
65	`int msync(void *addr, size_t len, int flags);`	Flush/sync pages containing *addr* to *addr+len* according to flags: `MS_ASYNC`: asynchronously `MS_SYNC`: synchronously (block) `MS_INVALIDATE`: invalidating caches Invokes `mach_vm_msync()`

As shown in the table, all these functions are really wrappers over the Mach VM primitives discussed in Chapter 12, which deals with Mach Virtual Memory. The functions all perform basic sanity checks, and then go on to obtain the current Mach memory map (by a simple call to `current_map()`) and invoke the underlying Mach function.

BSD Internal Memory Functions

The BSD layer requires its own memory management functions, which are naturally layered over those of Mach. These functions used extensively in the BSD portion of XNU, but not exported to user mode.

BSD's MALLOC and Zones

BSD code uses functions which closely resemble user mode's `malloc()` and friends. (See Listing 14-1.)

LISTING 14-1: BSD malloc functions, from bsd/sys/malloc.h

```
extern void     *_MALLOC(size_t       size,
                          int          type,
                          int          flags); // M_NOWAIT or M_ZERO

extern void     _FREE(void            *addr,
                        int            type);

extern void     *_REALLOC(void        *addr,
                        size_t         size,
                        int            type,
                        int            flags);

extern void     *_MALLOC_ZONE(size_t  size,
                        int            type,
                        int            flags);

extern void     _FREE_ZONE(void       *elem,
                        size_t         size,
                        int            type);
```

Figure 12-4, which discussed the various memory allocation techniques in XNU, showed (among other things) the mappings between the BSD layer allocations and the underlying low-level functions.

The BSD zones built on top of Mach zones (see Chapter 12), defined in a `kmzones[]` array of `struct kmzones`. Lion has around 114 zones, defined in `sys/malloc.h` as shown in Listing 14-2:

LISTING 14-2: BSD kmzones defined in bsd/sys/malloc.h

```
/*
 * Types of memory to be allocated (not all are used by us)
 */
#define M_FREE          0       /* should be on free list */
#define M_MBUF          1       /* mbuf */
#define M_DEVBUF        2       /* device driver memory */
#define M_SOCKET        3       /* socket structure */
#define M_PCB           4       /* protocol control block */
#define M_RTABLE        5       /* routing tables */
#define M_HTABLE        6       /* IMP host tables */
#define M_FTABLE        7       /* fragment reassembly header */
#define M_ZOMBIE        8       /* zombie proc status */
#define M_IFADDR        9       /* interface address */
#define M_SOOPTS        10      /* socket options */
#define M_SONAME        11      /* socket name */
#define M_NAMEI         12      /* namei path name buffer */
#define M_GPROF         13      /* kernel profiling buffer */
#define M_IOCTLOPS      14      /* ioctl data buffer */
#define M_MAPMEM        15      /* mapped memory descriptors */
#define M_CRED          16      /* credentials */
#define M_PGRP          17      /* process group header */
```

```
#define M_SESSION       18      /* session header */
#define M_IOV32         19      /* large iov's for 32 bit process */
#define M_MOUNT         20      /* vfs mount struct */
#define M_FHANDLE       21      /* network file handle */
#define M_NFSREQ        22      /* NFS request header */
#define M_NFSMNT        23      /* NFS mount structure */
#define M_NFSNODE       24      /* NFS vnode private part */
#define M_VNODE         25      /* Dynamically allocated vnodes */
#define M_CACHE         26      /* Dynamically allocated cache entries */
#define M_DQUOT         27      /* UFS quota entries */
#define M_UFSMNT        28      /* UFS mount structure */
#define M_SHM           29      /* SVID compatible shared memory segments */
#define M_PLIMIT        30      /* plimit  structures */
#define M_SIGACTS       31      /* sigacts structures */
#define M_VMOBJ         32      /* VM object structure */
#define M_VMOBJHASH     33      /* VM object hash structure */
#define M_VMPMAP        34      /* VM pmap */
#define M_VMPVENT       35      /* VM phys-virt mapping entry */
#define M_VMPAGER       36      /* XXX: VM pager struct */
#define M_VMPGDATA      37      /* XXX: VM pager private data */
#define M_FILEPROC      38      /* Open file structure */
#define M_FILEDESC      39      /* Open file descriptor table */
#define M_LOCKF         40      /* Byte-range locking structures */
#define M_PROC          41      /* Proc structures */
#define M_PSTATS        42      /* pstats  proc sub-structures */
#define M_SEGMENT       43      /* Segment for LFS */
#define M_LFSNODE       44      /* LFS vnode private part */
#define M_FFSNODE       45      /* FFS vnode private part */
#define M_MFSNODE       46      /* MFS vnode private part */
#define M_NQLEASE       47      /* XXX: Nqnfs lease */
#define M_NQMHOST       48      /* XXX: Nqnfs host address table */
#define M_NETADDR       49      /* Export host address structure */
#define M_NFSSVC        50      /* NFS server structure */
#define M_NFSUID        51      /* XXX: NFS uid mapping structure */
#define M_NFSD          52      /* NFS server daemon structure */
#define M_IPMOPTS       53      /* internet multicast options */
#define M_IPMADDR       54      /* internet multicast address */
#define M_IFMADDR       55      /* link-level multicast address */
#define M_MRTABLE       56      /* multicast routing tables */
#define M_ISOFSMNT      57      /* ISOFS mount structure */
#define M_ISOFSNODE     58      /* ISOFS vnode private part */
#define M_NFSRVDESC     59      /* NFS server socket descriptor */
#define M_NFSDIROFF     60      /* NFS directory offset data */
#define M_NFSBIGFH      61      /* NFS version 3 file handle */
#define M_MSDOSFSMNT    62      /* MSDOS FS mount structure */
#define M_MSDOSFSFAT    63      /* MSDOS FS fat table */
#define M_MSDOSFSNODE   64      /* MSDOS FS vnode private part */
#define M_TTYS          65      /* allocated tty structures */
#define M_EXEC          66      /* argument lists & other mem used by exec */
#define M_MISCFSMNT     67      /* miscfs mount structures */
#define M_MISCFSNODE    68      /* miscfs vnode private part */
#define M_ADOSFSMNT     69      /* adosfs mount structures */
#define M_ADOSFSNODE    70      /* adosfs vnode private part */
#define M_ANODE         71      /* adosfs anode structures and tables. */
#define M_BUFHDR        72      /* File buffer cache headers */
```

continues

LISTING 14-2 *(continued)*

```
#define M_OFILETABL     73      /* Open file descriptor table */
#define M_MCLUST        74      /* mbuf cluster buffers */
#define M_HFSMNT        75      /* HFS mount structure */
#define M_HFSNODE       76      /* HFS catalog node */
#define M_HFSFORK       77      /* HFS file fork */
#define M_ZFSMNT        78      /* ZFS mount data */
#define M_ZFSNODE       79      /* ZFS inode */
#define M_TEMP          80      /* misc temporary data buffers */
#define M_SECA          81      /* security associations, key management */
#define M_DEVFS         82
#define M_IPFW          83      /* IP Forwarding/NAT */
#define M_UDFNODE       84      /* UDF inodes */
#define M_UDFMNT        85      /* UDF mount structures */
#define M_IP6NDP        86      /* IPv6 Neighbour Discovery*/
#define M_IP6OPT        87      /* IPv6 options management */
#define M_IP6MISC       88      /* IPv6 misc. memory */
#define M_TSEGQ         89      /* TCP segment queue entry, unused */
#define M_IGMP          90
#define M_JNL_JNL       91  /* Journaling: "struct journal" */
#define M_JNL_TR        92  /* Journaling: "struct transaction" */
#define M_SPECINFO      93      /* special file node */
#define M_KQUEUE        94      /* kqueue */
#define M_HFSDIRHINT    95      /* HFS directory hint */
#define M_CLRDAHEAD     96      /* storage for cluster read-ahead state */
#define M_CLWRBEHIND    97      /* storage for cluster write-behind state */
#define M_IOV64         98      /* large iov's for 64 bit process */
#define M_FILEGLOB      99      /* fileglobal */
#define M_KAUTH         100     /* kauth subsystem */
#define M_DUMMYNET      101     /* dummynet */
#ifndef __LP64__
#define M_UNSAFEFS      102     /* storage for vnode lock state for unsafe FS */
#endif /* __LP64__ */
#define M_MACPIPELABEL  103     /* MAC pipe labels */
#define M_MACTEMP       104     /* MAC framework */
#define M_SBUF          105     /* string buffers */
#define M_EXTATTR       106     /* extended attribute */
#define M_LCTX          107     /* process login context */
/* M_TRAFFIC_MGT 108 */
#if HFS_COMPRESSION
#define M_DECMPFS_CNODE 109     /* decmpfs cnode structures */
#endif /* HFS_COMPRESSION */
#define M_INMFILTER     110     /* IPv4 multicast PCB-layer source filter */
#define M_IPMSOURCE     111     /* IPv4 multicast IGMP-layer source filter */
#define M_IN6MFILTER    112     /* IPv6 multicast PCB-layer source filter */
#define M_IP6MOPTS      113     /* IPv6 multicast options */
#define M_IP6MSOURCE    114     /* IPv6 multicast MLD-layer source filter */
#define M_LAST          115     /* Must be last type + 1 */
```

The zones are set by `kmeminit()` (from `bsd_init()` during boot). For each zone, `kmeminit()` calls the underlying Mach `zinit()` and sets a 1 MB zone accountable to the caller (i.e. `Z_CALLERACCT`). `_MALLOC_ZONE` then calls `zalloc_noblock` (if the element size requested is exactly that of the zone's) or `zalloc()`. Likewise, `FREE_ZONE` calls through to `zfree()` or `kfree()`.

Mcache and Slab Allocators

BSD offers another very efficient method of memory allocation, based on caches. This mechanism is known as mcache, and its implementation is in `bsd/kern/mcache.c`. The default implementation is built on top Mach zones providing the pre-allocated cache memory, but it is extensible for use with any back end slab allocator. The main advantage of using the mcache mechanism is its speed: The memory is allocated and maintained in a per-CPU cache, which enables mapping to the CPU's physical cache, greatly speeding up access.

The main client of this allocation system is the `mbuf` logic in the kernel. The `mbufs` (or memory buffers, in their full name), are often-reusable buffers of virtual memory, which represent network data (i.e. packets). The logic and structures behind `mbufs` are explored in Chapter 17.

Memory Pressure

As noted in Chapter 12 in the discussion of the PageOut daemon, the Mach VM layer supports the notion of VM pressure, which is defined as the condition wherein the system is dangerously low on available RAM. The handling of VM pressure is delegated to the BSD layer, and the layer also offers a system call (`vm_pressure_monitor` (#296) in `bsd/vm/vm_unix.c`), which directly wraps that of Mach. The file also contains several `vm` namespace MIBs, including the pressure indicator (`vm.memory_pressure`) and the PageOut daemon's targets.

When `consider_pressure_events` is called (by the PageOut daemon's garbage collection thread), the BSD layer takes over, and calls on `vm_try_pressure_candidates` (also in `bsd/kern/vm_pressure.c`). Candidates are those processes that have requested pressure notifications, by specifying an `EVFILT_VM/NOTE_VM_PRESSURE` combination in a call to `kevent`, or have had that done for them (iOS Objective-C apps, for example, which do so in the low level initialization of `libdispatch`).

For each candidate on the list, the system queries the resident page count (using `task_info`), and sends a `NOTE_VM_PRESSURE` knote (which triggers a `kevent` on its `kqueue`, as discussed later in this chapter) to a process whose resident page count is the highest (and exceeds the minimum of `VM_PRESSURE_MINIMUM_RSIZE`, set at 10 MB).

A candidate process is expected to respond to the pressure notification, which iOS Objective-C apps also do. Objective-C's garbage collection makes use of `libauto`, which calls on `libdispatch` to create a VM pressure dispatch source. The handler for this source calls `malloc_zone_pressure_relief` (as discussed in Chapter 4 under "Heap Allocations"). The Objective-C runtime also calls the app's `didReceiveMemoryWarning` callback, allowing the application to purge caches (as `libcache` does) and other unnecessary, but nice-to-have RAM.

Sometimes, alas, all this is not enough. Processes can't always find memory to discard. When the cooperative approach fails, desperate times call for desperate measures. This is when Jetsam kicks in.

> *Jetsam and Hibernation are both moving targets: undocumented and internal Apple APIs, which are constantly undergoing modification by Apple.*

Jetsam (iOS)

OS X and iOS implement a low-memory condition handler called Jetsam, or by another name Memorystatus (in `bsd/kern/kern_memorystatus.c`). This mechanism, somewhat similar in concept to Linux's "Out-Of-Memory" killer (known as oom), was originally used to kill processes consuming too much memory. The Jetsam name refers to the act of killing top memory consuming processes and jettisoning their memory pages. It seems Apple is moving towards the "Memorystatus" nomenclature, so this section will adopt it, as well.

XNU exports Memorystatus to user mode apps through `<sys/kern_memorystatus.h>`, and it's interesting to see this header evolve through subsequent versions of OS X. Most iOS developers remain oblivious to its presence, but are still indirectly affected by it, as their apps as their apps may be subject to sudden termination.

Memorystatus is implemented in `bsd/kern/kern_memorystatus.c`, and offers the functions shown in Table 14-2. Note that, in the Lion sources, these are still named `jetsam_*`, but this might change in future releases.

TABLE 14-2: Memorystatus Functions, from bsd/kern/kern_memorystatus.c

FUNCTION	USAGE
`jetsam_task_page_count` `(task_t task)`	Helper function used to compute a count of pages used by *`task`* (calls `task_info` and returns `resident_size` divided by `PAGE_SIZE`)
`jetsam_flags_for_pid` `(pid_t pid)`	Returns flags for specified *`pid`* from the `jetsam_priority_list`
`jetsam_snapshot_procs``(void)`	Records all `vm` page counters and traverses all processes (`allproc`) to record a snapshot, with a count of pages (using `jetsam_task_page_count`) and flags (using `jetsam_flags_for_pid`)
`jetsam_kill_hiwat_proc``(void)`	Kills (or suspends) processes whose page count exceeds the high-water mark
`jetsam_kill_top_proc``(void)`	Kills (or suspends) top memory-consuming processes

Memorystatus maintains two lists: a snapshot list, which captures the state of all processes in the system and how many pages they consume, and a priority list, which holds the candidate processes to be killed. The lists can be queried (in iOS) from user mode via `sysctl(2)`[1], and the latter list can even be set from user mode. `launchd(1)` is one such process which uses this mechanism: jobs may contain a `<JetsamPriorities>` key, which can specify the `JetsamMemoryLimit` and `JetsamPriority` (this is apparently used at present only for `syslogd`).

[1] If XNU is compiled with `DEVELOPMENT` or `DEBUG` settings, a third exported `sysctl` enables jetsam diagnostic mode.

By any name you call it, Memorystatus/Jetsam is more critical for iOS, and iOS seems to be a few steps ahead in its implementation. It is likely that the next version of iOS will also improve on it, possibly adding more user mode control mechanisms, or improving on `sysctl(2)`.

Process Hibernation (iOS)

In iOS 5 (and Lion, but only `#if CONFIG_FREEZE`), Jetsam/Memorystatus is integrated with the default freezer, which enables it to freeze, rather than kill the process. This provides for a much better user experience, because no data is lost and the process may be safely resumed when memory conditions improve. If `CONFIG_FREEZE` is defined, it enables the compilation of the following functions, shown in Table 14-3.

TABLE 14-3: Freezer-related Function (iOS only)

FUNCTIONS	LOCATED IN	USED FOR
`default_freezer_*`	`osfmk/vm/default_freezer.c`	The default freezer implementation.
`vm_object_pack` `vm_object_pack_pages` `vm_object_unpack` `vm_object_pagein` `vm_object_pageout`	`osfmk/vm/vm_object.c`	Packing or unpacking individual pages, which involves calling the `default_freezer` pack/unpack functions.
`vm_map_freeze` `vm_map_thaw` `vm_map_freeze_walk`	`osfmk/vm/vm_map.c`	Freezing or thawing the memory pages of a given VM map. Walking just iterates over the pages and checks which ones can be frozen.
`task_freeze` `task_thaw`	`osfmk/kern/task.c`	Freezing and thawing a task (calling `vm_map_freeze` or `vm_map_thaw` on the `task->map`).
`jetsam_send_hibernation_note` `jetsam_hibernate_top_proc`	`bsd/kern/kern_memorystatus.c`	Enables jetsam to freeze, rather than kill processes that match a given criteria. The hibernation note is a kernel event notifying of the pending hibernation of a PID.

The `CONFIG_FREEZE` setting also enables a new thread, the `kernel_hibernation_thread`. Note that, in this context, *hibernation* refers to per-process hibernation, and not to system hibernation. This thread wakes up when signaled (by `kern_hibernation_wakeup`), and checks if it needs to perform hibernation for processes. Memorystatus checks are performed on most `vm_page_*` operations (in `osfmk/vm/vm_resident.c`), by calls to the `VM_CHECK_MEMORYSTATUS`, which is defined in `bsd/sys/kern_memorystatus.h` to be a no-op on OS X, and a call to `vm_check_memorystatus` (`osfmk/vm/vm_resident.c`) in iOS (i.e. `#if CONFIG_EMBEDDED`). This function body is also only defined for iOS, as can be seen in Listing 14-3:

LISTING 14-3: VM Memorystatus checks conducted on page operations

```
void vm_check_memorystatus()
{
#if CONFIG_EMBEDDED
        static boolean_t in_critical = FALSE;
        static unsigned int last_memorystatus = 0;
        unsigned int pages_avail;

        if (!kern_memorystatus_delta) {
            return;
        }

        pages_avail = (vm_page_active_count +
                      vm_page_inactive_count +
                      vm_page_speculative_count +
                      vm_page_free_count +
                      (VM_DYNAMIC_PAGING_ENABLED(memory_manager_default) ? 0 :
                                       vm_page_purgeable_count));
        if ( (!in_critical && (pages_avail < kern_memorystatus_delta)) ||
             (pages_avail >= (last_memorystatus + kern_memorystatus_delta)) ||
             (last_memorystatus >= (pages_avail + kern_memorystatus_delta)) ) {
            kern_memorystatus_level = pages_avail * 100 / atop_64(max_mem);
            last_memorystatus = pages_avail;

            // This wakes up the memorystatus thread (as does pid_hibernate)
            thread_wakeup((event_t)&kern_memorystatus_wakeup);

            in_critical = (pages_avail < kern_memorystatus_delta) ? TRUE : FALSE;
        }
#endif
}
```

Actual process hibernation is carried out by calling `jetsam_hibernate_top_proc`, which freezes the underlying task (by calling `task_freeze`). Freezing involves walking the `vm_map` of the task, and passing it to the default freezer. User mode can also control hibernation by calling `pid_suspend()` and/or `pid_resume` (both in `bsd/vm/vm_unix.c`). iOS also defines `pid_hibernate`, which currently ignores its argument, and only wakes up the hibernation thread (i.e. signals `kern_hibernation_wakeup`).

Kernel Address Space Layout Randomization

Mountain Lion contains a new feature that is likely to go unnoticed by most of its users: Kernel address space layout randomization. While irrelevant for most applications, it has some paramount consequences. If and when it is introduced into iOS (iOS 6, most likely), it might spell the end of jailbreaking.

The concept of user mode ASLR was described in detail in Chapter 4. Once unheard of, ASLR has become a prerequisite for any operating system attempting to defeat hackers and stop malware trying to perform code injection. This, by now almost trite, technique involves an attacker embedding readily executable binary code in the input of some unsuspecting program, then overwriting a function pointer (often, a function's return address), to divert the program flow into the injected code.

The leading defense against code injection was Data Execution Prevention (DEP, also referred to as W^X, XD in Intel, and XN in ARM), which has made code injection significantly more difficult, though not impossible, for hackers. As the bar for entry was raised, hackers adapted by revamping an old technique. As described in Shacham's Black Hat 2008 presentation[1], return oriented programming is now a de facto standard technique for malicious code execution, but on reusing existing program code (commonly, LibC), by emulating the stack layout of valid program calls. The term stems from the fact that, as far as the program is concerned, the injected code is a sequence of function calls, which return from one function into the other. The overwritable stack segment is used for directing this sequence of calls, but does not contain any code that gets executed. This method, therefore, effectively defeats DEP.

If the address space is properly randomized, it becomes next to impossible to find any code to return to. It also becomes unlikely the attacker can guess any specific kernel address to overwrite, even if an overflow or other vulnerability does enable such an overwrite. This is especially important in the kernel, where code injection can lead to total system compromise and, in iOS, to device jailbreaking. ASLR Mountain Lion is therefore the first operating system to introduce kernel mode ASLR, and it seems a sure bet that iOS 6 will follow.

The implications for the kernel code are minimal: Instead of using fixed addresses, the code can shift to relative addresses, which are based on the current location of the program, held in Intel's IP or ARM's PC. The kernel is loaded by EFI or iBoot with a `vm_kernel_slide` value, like `dyld`'s slide (described in Chapter 4), and everything proceeds normally. (Prelinked modules (kexts) are also subjected to the slide.)

The implications for malware or jailbreaking, however, are far reaching and more severe. At the time of writing, there is no clever workaround for proper ASLR. As a bonus, reverse engineering becomes somewhat harder (as the IP relative addresses can be set in several ways, instead of leaving fixed offsets for strings and function names).

Mountain Lion exports a new system call, `kas_info` (#439), which can be used to query the value of the kernel slide. This system call might not remain for too long, (especially in iOS) because leaking the value of the slide defeats the entire purpose of randomization.

> *Even with KASLR, pre-A5 devices will still be fully jailbreakable. This is because the vulnerability allowing the jailbreak is in iBoot itself, allowing the direct patching of the kernel. In this case, run-time addresses matter little, as jailbreakers can prepare a custom IPSW of a patched kernel. That said, it's only a matter of time before Apple removes support for those devices, the way it no longer supports the very first generation of the iPhone.*

WORK QUEUES

Work queues are a mechanism developed in OS X to facilitate multithread support for applications and scale to multiple CPUs. This mechanism is not exported directly to user mode (and hence was not mentioned in Chapter 3), but is nonetheless important, as it provides the foundation for Apple's Grand Central Dispatch (GCD). This section does not discuss how to use GCD (though a good reference exists in Apple Developer[2] and in a book devoted to multithreading[3]). Rather, it focuses on how GCD itself uses XNU's services[2]. Work queues are provided through two undocumented system calls: `workq_open` (#367) and `workq_kernreturn` (#368), both implemented (along with all other work queue functions) in `bsd/kern/pthread_synch.c`. The `workq_open` system call is used to create a work queue and is wrapped by LibC's `pthread_workqueue_create_np` (and further by GCD and libdispatch's `dispatch_get_global_queue`). It doesn't take any arguments. The `workq_kernreturn` system call is used for pretty much everything else, and can control the work queue, by specifying one of three currently defined options:

- **`WQOPS_QUEUE_ADD`** — The caller may specify an `item` (as the second argument) to be executed by the work queue. This item corresponds to the block or function to be executed (or *dispatched*, in GCD parlance). The caller may also request `affinity` (currently ignored), and specify a `prio` between up to `WORKQUEUE_NUMPRIOS` (currently 4), as well as an overcommit bit. These queues are listed in `bsd/sys/pthread_internal.h` as shown in Listing 14-3:

LISTING 14-3: Global work queues in XNU

```
#define WORKQUEUE_HIGH_PRIOQUEUE    0       /* high priority queue */
#define WORKQUEUE_DEFAULT_PRIOQUEUE 1       /* default priority queue */
#define WORKQUEUE_LOW_PRIOQUEUE     2       /* low priority queue */
#define WORKQUEUE_BG_PRIOQUEUE      3       /* background priority queue */
```

  If these seem somewhat familiar, it's for a good reason: They are the same global work queues offered by GCD (though with different `DISPATCH_QUEUE_PRIORITY_*` constants). Libdispatch actually creates two copies of each queue, with the additional copy set to overcommit, though these are not exported to callers directly. In this way, the application's main queue is really just a reference to the default queue, with overcommit set. The overcommit bit (which is also accessible via the undocumented `pthread_workqueue_attr_[get/set]overcommit_np`) denotes that new threads may be created for this queue. This strategy is generally discouraged, as more threads than the CPUs can handle slow down the program. GCD supports the idea of overcommit through the only valid flag for `dispatch_get_global_queue` (`DISPATCH_QUEUE_OVERCOMMIT`), but Apple's documentation hides that fact and claims the flag must be zero.

- **`WQOPS_THREAD_SETCONC`**: This controls work queue concurrency and is wrapped by `pthread_workqueue_requestconcurrency_np()`.

[2] GCD and libdispatch can also operate in the absence (or disablement) of work queues, in which case they fall to a thread pool model. This can be forced by setting the `LIBDISPATCH_DISABLE_KWQ` variable.

- **`WQOPS_THREAD_RETURN`**: This detaches from the work queue and terminates thread. It is wrapped by pthread's `workqueue_exit()`, in a call to the internal `_pthread_workq_return`.

The work queue set up logic (triggered as the result of item addition) is quite unique in XNU. The main work is performed by `wq_runitem`, which calls on `setup_wqthread` to manually construct the work queue thread's state, register by register. This is followed by waking up the thread in its new persona. The state setup is shown in Listing 14-4:

LISTING 14-4: Setting a work queue thread's state

```
int setup_wqthread(proc_t p, thread_t th, user_addr_t item, int reuse_thread,
                   struct threadlist *tl)
{
#if defined(__i386__) || defined(__x86_64__)
        int isLP64 = 0;

        isLP64 = IS_64BIT_PROCESS(p);
        /*
         * Set up i386 registers & function call.
         */
              // very similar to x86_64 case, so omitted
        } else {
            x86_thread_state64_t state64;
            x86_thread_state64_t *ts64 = &state64;

            ts64->rip = (uint64_t)p->p_wqthread; // Thread will resume from this point
            ts64->rdi = (uint64_t)(tl->th_stackaddr + PTH_DEFAULT_STACKSIZE +
                                                      PTH_DEFAULT_GUARDSIZE);
            ts64->rsi = (uint64_t)(tl->th_thport);
            ts64->rdx = (uint64_t)(tl->th_stackaddr + PTH_DEFAULT_GUARDSIZE);
            ts64->rcx = (uint64_t)item;
            ts64->r8 = (uint64_t)reuse_thread;
            ts64->r9 = (uint64_t)0;

             /*
              * set stack pointer aligned to 16 byte boundary
              */
             ts64->rsp = (uint64_t)((tl->th_stackaddr + PTH_DEFAULT_STACKSIZE +
                                     PTH_DEFAULT_GUARDSIZE) - C_64_REDZONE_LEN);

             // This had better work, or else..
             if ((reuse_thread != 0) && (ts64->rdi == (uint64_t)0))
                     panic("setup_wqthread: setting reuse thread with null pthread\n");

           // Call architecture specific thread state setting (osfmk/i386/pcb_native.c)
           thread_set_wq_state64(th, (thread_state_t)ts64);
        }
#else
#error setup_wqthread not defined for this architecture //unless you have iOS sources.
#endif
        return(0);
}
```

The `proc_info` system call (described in detail in Chapter 5 and in the previous chapter) provides the `PROC_PIDWORKQUEUEINFO` flavor, which displays work queues in a given process. This is also available through libproc's `proc_pidinfo()`, and returns information as shown in Listing 14-5:

LISTING 14-5: The structure returned for PROC_PIDWORKQUEUEINFO

```
struct proc_workqueueinfo {
        uint32_t     pwq_nthreads;       /* total number of workqueue threads */
        uint32_t     pwq_runthreads;     /* total number of running workqueue threads */
        uint32_t     pwq_blockedthreads; /* total number of blocked workqueue threads */
        uint32_t     pwq_state;          // new in Lion and later
};

/*
 *      workqueue state (pwq_state field)
 */
#define WQ_EXCEEDED_CONSTRAINED_THREAD_LIMIT    0x1
#define WQ_EXCEEDED_TOTAL_THREAD_LIMIT          0x2
```

BSD HEIRLOOMS REVISITED

Chapter 3 discussed the many technologies in OS X and iOS derived from and inspired by BSD, albeit from the user mode and administrator perspective. The rest of this chapter revisits these same technologies, but explores their kernel-level implementation in XNU.

Sysctl

BSD, like many other UNIX systems, offers a uniform interface for getting and setting kernel variables, called `sysctl(8)`. Unlike systems such as Linux, however, this is the only way to get access to the variables, for lack of a user-visible file representation in a `/proc` file system. The `sysctl` command was discussed in Chapter 3; this section discusses its implementation. As a reminder, the `sysctl` parameters are divided into the namespaces shown in the Table 14-4. With the exception of security, they are all defined in `bsd/sys/sysctl.h`, which is made available to user space as `<sys/sysctl.h>`:

TABLE 14-4: The sysctl Top-level Namespaces

SYSCTL NAMESPACE	USED FOR
CTL_KERN	Kernel variables and settings, such as the version string, process limits, and so on.
CTL_VM	Virtual memory manager settings and statistics.
CTL_VFS	Virtual file system switch settings. Discussed in Chapter 15, which deals with file systems.

SYSCTL NAMESPACE	USED FOR
CTL_NET	Network settings. Subdivided into net.link.*, net.inet.*, net.inet6.*, and further into transport layer protocols. Discussed in Chapter 17, which deals with networking.
CTL_DEBUG	Debug settings.
CTL_HW	Hardware settings: physmem, cpufrequency, and so on. Naturally, these are read-only.
CTL_MACHDEP	Machine-dependent settings. These differ greatly from OS X to iOS, and are further subdivided into cpu, pmap, memmap, and others.
CTL_USER	User-level identifiers.
_security (security/mac_internal.h)	Security settings. Currently only contains one sub-namespace, mac, which configures the MAC layer. Discussed in detail in this chapter.

XNU has two main files for dealing with sysctl(), bsd/kern/kern_newsysctl.c, which is the implementation of the architecture generic sysctls, and bsd/dev/<arch>/sysctl.c, which contains machine-specific ones (i.e. the machdep.* sysctls). Pre-SL kernels contained a ppc/ arch directory, and iOS likely contains an arm/ one, but the only one present in the open source version is i386/.

The sysctls are maintained in sysctl_oid structures, defined in bsd/sys/sysctl.h as shown in Listing 14-5.

LISTING 14-5: sysctl oid implementation

```
struct sysctl_oid {
        struct sysctl_oid_list *oid_parent;
        SLIST_ENTRY(sysctl_oid) oid_link;
        int             oid_number;
        int             oid_kind;
        void            *oid_arg1;
        int             oid_arg2;
        const char      *oid_name;
        int             (*oid_handler) SYSCTL_HANDLER_ARGS;
        const char      *oid_fmt;
        const char      *oid_descr; /* offsetof() field / long description */
        int             oid_version;
        int             oid_refcnt;
};
```

New sysctls may be constructed by calling a specialized macro, SYSCTL_OID, which defines the sysctl, initializes its fields, and informs the linker of it. Using one of the macros built on top of it, however, is easier (see Table 14-5):

TABLE 14-5: sysctl Type Declaration Macros

SYSCTL MACRO	USED FOR
`SYSCTL_DECL`	Declaring a top-level entry. XNU uses it for the types defined Table 13-sysc. Kernel extensions (for example, VMWare) use it for private namespaces.
`SYSCTL_OID`	Raw OIDs. Seldom used directly. May specify type as "N," "A," "I," "IU," "L," or "Q," corresponding to the `SYSCTL_*` constants shown in this table.
`SYSCTL_NODE`	Container nodes.
`SYSCTL_STRING`	Leaf nodes, containing `char *` data. `sysctl_handle_string()` is called.
`SYSCTL_COMPAT_INT` `SYSCTL_INT`	Leaf nodes, compatibility (old API) or preferred API for signed integer data
`SYSCTL_COMPAT_UINT` `SYSCTL_UINT`	Leaf nodes, compatibility (old API) or preferred API for unsigned integer data.
`SYSCTL_LONG`	Leaf nodes, with long integer data. `sysctl_handle_long()` called as handler.
`SYSCTL_QUAD`	Leaf nodes, with quad word data — i.e. 64-bit integers. `sysctl_handle_quad()` is called as handler.
`SYSCTL_OPAQUE`	Leaf nodes, with unspecified data. Some `void *` with given length. `sysctl_handle_opaque()` is called as handler.
`SYSCTL_STRUCT`	Leaf nodes, with structure data. `sysctl_handle_opaque()` is called as handler.
`SYSCTL_PROC`	Leaf nodes, but caller specifies own handler function.

An additional macro, `SYSCTL_PROC`, is used to declare leaf handlers, which are the callback functions that the kernel invokes when user space issues a `sysctl`. Defining your own handler thus becomes a fairly straightforward matter, involving two steps:

1. Define the `SYSCTL_NODE` by which your handler will be called:

```
SYSCTL_NODE(parent,   // _kern, _debug, or your own top level namespace..
      OID_AUTO, // request OID assignment by kernel
      myname,   // your name
      flags,    // access: CTLFLAG_*, bitwise OR'ed
      0,        // handler
      "sysctl description"); // some description
```

   Optionally, you may want to define a `SYSCTL_DECL` top-level namespace, as well:

```
SYSCTL_DECL(myname);
```

   You may skip this step altogether if you are only adding a leaf to an already-existing `sysctl` node.

2. Define the actual `sysctl` leaf your handle is supposed to implement. Here, you have two options:

a. Use one of the types from Table 14-5. This installs a default handler for you, and all you need to specify is the variable that holds the `sysctl` data. You lose, however, the ability to get a callback notification on value read or change. Almost all these macros are highly similar. For example, if you wanted an integer, you would specify the following:

```
SYSCTL_INT (parent,  // node created or used in step 1.
     nbr,     // OID_AUTO: so as not to worry about numbers
     name,    // name of leaf
     access,  //CTL_* flags: _RW, _ANYBODY… etc
     ptr,     // address of variable holding this data
     val,     // Used if ptr is NULL. Leaf is then read-only
     descr);  // textual description
```

b. Define the leaf as a `SYSCTL_PROC`, specifying the handler implementation. You then need to implement the handler as follows:

```
SYSCTL_PROC(parent,     // node created or used in step 1
     nbr,        // OID_AUTO, as usual
     name,       // name of leaf
     access,     // CTL_* flags, as above
     ptr,        // pointer to variable data
     arg,        // argument to handler
     handler,    // pointer to your own handler
     fmt,        // "A", "I", "IU", … as above
     descr);
```

The advantage of the latter approach is in getting the notification whenever some operation is attempted on the `sysctl`. This is somewhat like Linux, in which `/proc` and `/sys` file system handlers can listen in on access or changes to the exported data, and execute some operation when they occur.

Kqueues

Kqueues have been introduced into BSD, as an alternative to the `poll(2)/select(2)` model, which is deemed insufficiently scalable. Devised by Jonathan Lemon of the FreeBSD project[4], they are described as a "generic event delivery mechanism, which allows an application to select from a wide range of event sources, and be notified of activity on these sources in a scalable and efficient manner." An emphasis is placed on the extensibility of the interface, allowing the addition of any number of future event sources, without changes to the programming interface.

XNU exports two system calls for kqueues: The first, `kqueue` (#362) creates the kqueue, which is basically a file descriptor. The second, `kevent/kevent64` (#363 or #369, respectively) is used for setting event filters and reading from the kqueue. An example of their usage was presented in Listing 3-1.

The kernel implementation of kqueues is self-contained in a single file, `bsd/sys/kern_event.c`. The kqueue, as a file descriptor, is defined by its fileops, which are tied to the file descriptor when the kqueue is created. This is shown in the implementation of `kqueue(2)` in Listing 14-6.

LISTING 14-6: The implementation of kqueue(2), from bsd/sys/kern_event.c

```
int kqueue(struct proc *p, __unused struct kqueue_args *uap, int32_t *retval)
{
        struct kqueue *kq;
        struct fileproc *fp;
```

continues

LISTING 14-6 *(continued)*

```
        int fd, error;

        // allocate file structure fp as file descriptor fd
        error = falloc(p, &fp, &fd, vfs_context_current());
        if (error) {
                return (error);
        }
        // allocate actual kqueue
        kq = kqueue_alloc(p);
        if (kq == NULL) {
                fp_free(p, fd, fp);
                return (ENOMEM);
        }

        fp->f_flag = FREAD | FWRITE; // make descriptor readable/writable
        fp->f_type = DTYPE_KQUEUE;   // mark descriptor type as a queue
        fp->f_ops = &kqueueops;      // tie kqueue operations to file operations
        fp->f_data = (caddr_t)kq;    // tie kqueue to file structure

        // kqueue is not really backed by a file, so release unnecessary parts
        proc_fdlock(p);
        procfdtbl_releasefd(p, fd, NULL);
        fp_drop(p, fd, fp, 1);
        proc_fdunlock(p);

        *retval = fd;                // return fd to user
        return (error);
}
```

Both the `kevent(2)` and `kevent64(2)` calls end up using the same function, `kevent_internal`, which either sets the event filter (if supplied), or uses Mach continuations to block until an event arrives. The kernel event notifications themselves are known as *knotes*, and in that respect a kqueue can be seen as a linked list of knotes. A knote may belong to several kqueues, and the kqueues are the mechanism by means of which the user filtering is performed.

If XNU is compiled with socket support (which it is, by default), the `bsd/kern/kern_event.c` file also contains the implementation of kernel event sockets. These are referred to as `kev`s, but are actually part of a different mechanism, called system sockets (discussed in greater detail in Chapter 17). The corresponding user mode header file, `<sys/kern_event.h>`, refers to system sockets, and it is `<sys/event.h>`, which contains the exports for kevents.

Auditing (OS X)

Recall the discussion of auditing in Chapter 3, from the administrator's perspective. The chapter introduced the user commands of `praudit(1)` and the special audit device, `/dev/auditpipe`. From the kernel perspective, auditing is simply a matter of lacing the system call invocation logic (Listing 14-7) with several macros:

- **AUDIT_SYSCALL_ENTER**: Called right before the invocation of AUNIX system call from the sysent table. The macro takes three arguments: the system call *code* (number), the BSD process, and thread objects responsible for the call.
- **AUDIT_ARG**: Called inside the system call implementation. This takes the operation (argument typedef), and a variable number of arguments, corresponding to those of the system call.
- **AUDIT_SYSCALL_EXIT**: Called right after the system call implementation. Arguments are the same as those of ENTER, along with the return value of the system call.

LISTING 14-7: Auditing support in unix_syscall (bsd/dev/i386/systemcalls.c)

```
void unix_syscall(x86_saved_state_t *state)
{
   // ...
   AUDIT_SYSCALL_ENTER(code, p, uthread);
   error = (*(callp->sy_call))((void *) p, (void *) vt, &(uthread->uu_rval[0]));
   AUDIT_SYSCALL_EXIT(code, p, uthread, error);
   // ...
}
```

Additional macros exist for auditing Mach traps, but those are only used when a BSD call results in a Mach call and, even then, for only select Mach traps.

The auditing macros are defined in bsd/security/audit/audit.h. The macros check the value of the audit_enabled global variable, so as to avoid the need for any overhead if auditing is disabled. The administrator can toggle the value of this variable using the auditon(2) system call with the A_SETCOND command.

If auditing is indeed enabled, the macros either create a new kaudit_record (eventually calling audit_new), or use an existing audit record, if one can be found on the BSD thread's uu_ar field. An audit record is finalized by a call to audit_commit, which moves the audit record to an audit_q. Once the record is on the queue, the thread's uu_ar is reset.

In addition to placing the record in the audit_q, audit_commit also signals a condition variable, audit_worker_cv. Doing so wakes up the dedicated audit worker thread by continuation, and it processes the record (in audit_worker_process_record) by calling kaudit_to_bsm, which converts it into an OpenBSM-compatible format. The record can then be directly written (from the kernel) to the audit file, submitted to any audit pipes, and, as of Lion, to the audit session devices (by audit_sdev_submit, in audit_session.c). It is then freed. This is shown in Listing 14-8.

LISTING 14-8: Audit worker thread record processing

```
/*
 * Given a kernel audit record, process as required.  Kernel audit records
 * are converted to one, or possibly two, BSM records, depending on whether
 * there is a user audit record present also.  Kernel records need be
 * converted to BSM before they can be written out.  Both types will be
 * written to disk, and audit pipes.
 */
```

continues

LISTING 14-7 *(continued)*

```
static void audit_worker_process_record(struct kaudit_record *ar)
{
   // …

   // Convert to BSM record format
   error = kaudit_to_bsm(ar, &bsm);
   switch (error) {
     /// error handling on all codes is basically a goto out
   }

   //
   // Write directly to the file. The audit_vp is the vnode of the audit file
   //
   if (ar->k_ar_commit & AR_PRESELECT_TRAIL) {
          AUDIT_WORKER_SX_ASSERT();
          audit_record_write(audit_vp, &audit_ctx, bsm->data, bsm->len);
      }

   //
   // Send to any /dev/auditpipe instances
   //
   if (ar->k_ar_commit & AR_PRESELECT_PIPE)
          audit_pipe_submit(auid, event, class, sorf,
          ar->k_ar_commit & AR_PRESELECT_TRAIL, bsm->data,
          bsm->len);

   //
   // Send to any /dev/auditsessions device instances (new in Lion)
   //
   if (ar->k_ar_commit & AR_PRESELECT_FILTER) {
    /*
     *  XXXss - This needs to be generalized so new filters can
     *  be easily plugged in.
     */
     audit_sdev_submit(auid, ar->k_ar.ar_subj_asid, bsm->data,
      bsm->len);
   }

        kau_free(bsm);
out:
        if (trail_locked)
                AUDIT_WORKER_SX_XUNLOCK();
}
```

The `audit_vp` is an interesting example of kernel code writing directly to files, without user mode intervention. This is a necessary shortcut, due to the security sensitive nature of auditing.

Mandatory Access Control

Chapter 3 introduced the user mode view of the Mandatory Access Control (MAC), a powerful security feature Apple imported from TrustedBSD. That view, however, is extremely limited, as enforcement can be reliably carried out only by the kernel. This section discusses the implementation of MAC, delving deeper into its two main implementations: OS X's sandbox and iOS's entitlements.

MAC Policies

A MAC policy is visible to the user only as an opaque object. In the kernel, however, the policy is a `mac_policy_conf` structure, defined in `security/mac_policy.h`. A policy module is expected to register this structure on entry using `mac_policy_register`, and deregister (using `mac_policy_unregister`) on exit. A `MAC_POLICY_SET` macro is available to emit all this code automatically, as shown in Listing 14-9:

LISTING 14-9: the MAC_POLICY_SET macro from security/mac_policy.h

```
#define MAC_POLICY_SET(handle, mpops, mpname, mpfullname, lnames, lcount, slot, lfl
ags, rflags) \
  static struct mac_policy_conf mpname##_mac_policy_conf = {      \
   .mpc_name              = #mpname,     /* Policy name */      \
   .mpc_fullname          = mpfullname,  /* Policy official name */   \
   .mpc_labelnames        = lnames,  /* Label names (char **) */     \
   .mpc_labelname_count   = lcount,  /* Count of label names */   \
   .mpc_ops               = mpops,  /* Policy operations (see below) */ \
   .mpc_loadtime_flags    = lflags, /* MPC_LOADTIME_FLAG_* constants */  \
   .mpc_field_off         = slot,   /* int * holding policy slot, or NULL */     \
   .mpc_runtime_flags     = rflags  /* only MPC_RUNTIME_FLAG_REGISTERED defined */ \
    };                                                             \
                                                                   \
        static kern_return_t                                       \
        kmod_start(kmod_info_t *ki, void *xd)                      \
        {                                                          \
                return mac_policy_register(&mpname##_mac_policy_conf,   \
                    &handle, xd);                                  \
        }                                                          \
                                                                   \
        static kern_return_t                                       \
        kmod_stop(kmod_info_t *ki, void *xd)                       \
        {                                                          \
                return mac_policy_unregister(handle);              \
        }                                                          \
  extern kern_return_t _start(kmod_info_t *ki, void *data);       \
        extern kern_return_t _stop(kmod_info_t *ki, void *data);        \
                                                                   \
        KMOD_EXPLICIT_DECL(security.mpname, POLICY_VER, _start, _stop)  \
        kmod_start_func_t *_realmain = kmod_start;                 \
        kmod_stop_func_t *_antimain = kmod_stop;                   \
        int _kext_apple_cc = __APPLE_CC__
```

The key field in the `mac_policy_conf` structure is `mpc_ops`, which is a pointer the `mac_policy_ops structure`. This is a gargantuan struct of well over 300 function pointers, which each policy module is expected to either implement, or leave NULL. The function pointers cover virtually every operation in the system, following a naming convention of `mpo_`*`object_operation_call`*, where:

- ***object*** is the object type: file (really, descriptor), port, socket, sysvsem, proc, vnode (file)
- ***operation*** is either "label" or "check." The "label" operation corresponds to a label related operation. The "check" operation corresponds to authorizing a system call or trap.
- ***call*** is, for a check, usually the name of the system call (or Mach trap) the access check relates to. For label, one of the stages of the label lifecycle, usually `init`, `associate` and `destroy`, and sometimes other specific verbs.

When XNU calls on the MAC layer to validate an operation, the MAC layer calls on the policy modules, in turn, for validation. All MAC checks follow roughly the same template. As an example, consider a highly useful `mac_vnode_check_signature`, which is responsible for the enforcement of code signing. This is shown in listing 14-10:

LISTING 14-10: mac_vnode_check_signature, from security/mac_vfs.h

```
int
mac_vnode_check_signature(struct vnode *vp, unsigned char *sha1,
                          void * signature, size_t size)
{
        int error;

        // if either security.mac.vnode_enforce or security.mac.proc_enforce sysctls
        // are 0 (false), we just return 0 as well, never getting to the check.

        if (!mac_vnode_enforce || !mac_proc_enforce)
                return (0);

       // Otherwise, walk policy module list,execute mpo_vnode_check_signature for each
        MAC_CHECK(vnode_check_signature, vp, vp->v_label, sha1, signature, size);
        return (error);
}
```

The `MAC_CHECK` macro (defined in `security/mac_internal.h`) walks through the policy list to validate the operation by each of the registered modules. This walk, however, will be performed only if the global `mac_`*`xxx`*`_enforce` checks are true. Setting any of the `security.mac.`*`xxx`*`_enforce` variables (shown in Output 3-3) to 0 causes the resulting `mac_`*`xxx`*`_enforce` variable in the kernel to be false, and thus all the related checks of the subsystem to return 0 (i.e. a "go ahead"), rather than actually performing the check, which may result in an error.

Recall from Chapter 3, that the MAC layer exports `sysctl(2)` MIB variables, which allow the administrator to selectively disable enforcement. Looking back at the listing, it is easy to see how this is performed: If either `mac_vnode_enforce` or `mac_proc_enforce` are false, then the check is short circuited and returns 0 ("go ahead") on the operation.

APPLE'S POLICY MODULES

Even though the MAC framework is reasonably well documented and used by third-party software in FreeBSD, in OS X and iOS it mostly caters to Apple itself, due to the relative dearth of anti-malware and security software (a situation which is starting to change). MAC's primary use in OS X is for the sandbox mechanism (formerly seatbelt), and in iOS MAC enables the rigid code signing and entitlements which enable Apple to protect their precious from the horrors of third party code.

Sandbox.kext

The sandbox kernel extension for OS X has been reversed by Dionysus Blazakis, who has thoroughly documented his findings in a paper presented at BlackHat DC 2011[5]. His analysis, however, is for Snow Leopard's version (34.1), as Lion was not yet released at the time. Lion's version is considerably newer (177.3), and Mountain Lion's newer still, at 189. The iOS 5.1 version seems to be an almost direct port of the OS X one, with several differences:

- The iOS sandbox reports a slightly older version (154.9) than Lion's (177.3).
- The iOS Sandbox is tightly coupled with `AppleMobileFileIntegrity` (discussed next).
- iOS has no `qtn-*` keys (required for the quarantine feature of OS X), as the system does not support this notion. There are also no user-preference* keys.
- By default, the sandbox restricts all third-party applications (from `/private/var/mobile/Applications`) to their directory. This is the well known "jail" that jailbreakers break out of, by patching the sandbox evaluation logic.
- In the OS X version, applications can be unsandboxed. This is not the case with iOS.

The sandbox kernel extension sometimes requests the services of `/usr/libexec/sandboxd`. This daemon, which is started by `launchd(1)`, claims host special port #14 (still `#defined` at `HOST_SEATBELT_PORT`).

As mentioned in Chapter 3, `Sandbox.kext` implements a tinySCHEME-like dialect for defining authorization and operation permissions. This textual format is compiled in user mode on-the-fly, and then submitted to the kernel for later policy approvals. It is the role of a second kext, `AppleMatch.kext`, to perform the policy and regular expression matching.

The Sandbox policy is a static definition, and can be found easily thanks to the hardcoded strings "sandbox" and "Seatbelt sandbox policy." Apple has graciously left these in plain text (along with all too many other strings!). Locating the reference to the policy name leads you to the policy structure, and locating the policy structure leads you straight to the sandbox initialization function.

> *The book's companion jtool, introduced in Chapter 4, has a powerful search feature in Mach-O objects. This feature is exceptionally useful if you're trying to find strings, which can lead you to the more "interesting" parts of a binary. Using the `-f` switch, jtool can be asked to perform a fast search for a string, and reveal its location not only in the file, but also in the resulting memory segment. Using the `-fr` switch will also reveal where the string is referenced, which is usually in or around the function that uses it.*

AppleMobileFileIntegrity.kext

iOS has a far more stringent security mechanism than its older sister. Unlike OS X, wherein code signing is optional, iOS will blatantly `kill -9` any process that is not properly code signed. XNU is not to be blamed for this; it's just following orders. The role of "bad cop" is played by `AppleMobileFileIntegrity.kext`. Like `Sandbox.kext`, AFMI has a henchman in user mode: `/usr/libexec/amfid`. This daemon is started from `launchd`, which also registers for it host special port #18 (`HOST_AMFID_PORT`). The daemon accepts messages from AMFI, and assists it with tasks tasks are best implemented in user mode.

Reverse engineering `initializeAppleMobileFileIntegrity` (which is called from the kext's `_Start` function, and does all its work) reveals that it calls `mac_policy_register`, as all policy modules must. The policy it is mostly NULL, but contains callbacks for the following:

- **`mpo_vnode_check_exec`**: AMFI's callback returns 1 (allowing execution for the `vnode`) but not before setting the code signing flags (`CS_HARD` and `CS_KILL`). This ensures that all processes will have to go code signature checks, and can always die another later if the need arises.
- **`mpo_vnode_check_signature`**: This is the main logic of AMFI, which uses the `amfid` and its own in-kernel signature cache to validate the code signature of a file. If this function returns true, then Listing 14-10 returns true as well, and the binary is allowed. This is also why this check (specifically, the in-kernel cache check) is a favorite target for patching.
- **`mpo_proc_check_get_task`**: This protects `task_for_pid` calls, which as described earlier in this book enable obtaining the task's port (and complete control over it). The hook checks two entitlements (`get-task-allow` and `task_for_pid-allow`, as well as a call to check if unrestricted debugging is enabled (using the `amfid`), and returns true if any of the above is affirmative.
- **`mpo_proc_check_run_cs_invalid`**: This checks if the `get-task-allow`, `run-invalid-allow`, or `run-unsigned-code` entitlements are set, or if unrestricted debugging is enabled. If this check returns true, `cs_allow_invalid` (from `bsd/sys/kern_proc.c`) clears the `CS_KILL`, `CS_HARD`, and `CS_VALID` bits, and returns true as well, allowing unsigned code.

AMFI recognizes several boot arguments, which it parses (using `PE_parse_boot_argn`), that can disable some checks. These are listed in Table 14-6. Bear in mind, however, that there is no known way to pass boot-args to XNU on A5-devices and later.

TABLE 14-6: AMFI Boot Arguments

AMFI BOOT ARGUMENT	USAGE
`PE_i_can_has_debugger`	Global boot argument used throughout XNU to denote debugger attachment is permitted. Disables most checks.
`cs_debug`	Disables code signing.
`cs_enforcement_disable`	Disables enforcement of code singing; check is still performed, but neutered.

`amfi_allow_any_signature`	Allow any signature on code, not just Apple's.
`amfi_unrestrict_task_for_pid`	Allow `task_for_pid` regardless of whether the process has the `get-task-allow` and `task_for_pid-allow` entitlements.
`amfi_get_out_of_my_way`	Just disable AMFI altogether. Apparently Apple's own developers get tired of AMFI's meddling every now and then.

Other policy modules may be dynamic, but `AppleMobileFileIntegrity` is certainly not. Although the kext has a stop function, any attempt to unload it will result in a kernel panic ("Cannot unload AMFI — policy is not dynamic"). Likewise, if for some reason it cannot initialize, it panics the kernel, complaining that "AMFI failed to initialize. This would compromise system security."

You can locate AMFI in a manner similar to the one described for the Sandbox: Searching for references to "Apple Mobile File Integrity" will lead you right to `initializeAppleMobileFileIntegrity`, as shown in Output 14-1:

OUTPUT 14-1: Locating AMFI in the iOS 5 kernelcache using jtool

```
morpheus@Ergo (/)$ jtool -fr "Apple Mobile File Integrity" ~/iOS/iOS.5.0.0.kernelcache
Searching for string "Apple Mobile File Integrity" and all references to it:
 - Found at file offset: 0x5ae5ba, Memory: 0x805f15ba (Segment: __PRELINK_TEXT)
References to 0x805f15ba:
 - Reference found at file offset: 0x5a1144, Memory: 0x805e4144(Segment: __PRELINK_TEXT)
```

SUMMARY

This chapter discussed advanced aspects of XNU's BSD layer. It began by reviewing BSD memory management, both the POSIX exported calls and the internal functions used. It further covered dealing with memory pressure, and touched on kernel address space layout randomization (KASLR), a feature soon to appear in Mountain Lion, and very likely iOS 6.

We continued with a review of the kernel perspective of several BSD features, such as `sysctl(2)`, kqueues and auditing. Finally, the spotlight moved to the kernel implementation of the Mandatory Access Control Framework (MAC), and the implementation of two important policy modules: the Sandbox and iOS's AMFI.

Our discussion of the BSD layer is only beginning, as we turn our gaze towards two important subsystems: File Systems (Chapter 15), and Networking (Chapter 17).

REFERENCES

1. Hovav Shacham, et al, "Return-Oriented Programming: Exploits Without Code Injection," `http://cseweb.ucsd.edu/~hovav/talks/blackhat08.html`

2. Apple Developer. “Concurrency Programming Guide,” http://developer.apple.com/library/mac/#documentation/General/Conceptual/ConcurrencyProgrammingGuide
3. Sakamoto, Kazuki and Tomohiko Furumoto, *Pro Multithreading and Memory Management for iOS and OSX*. Apress; 2012
4. Kqueues, http://people.freebsd.org/~jlemon/papers/kqueuepdf
5. Blazakis, Dionysus “The Apple Sandbox,” http://www.semantiscope.com/research/BHDC2011/

15

Fee, FI-FO, File: File Systems and the VFS

One of the kernel's major responsibilities is handling data, both the user's and of the system's. To this end, data is organized into files and directories, which reside on file systems of various types.

XNU's BSD layer is responsible for implementing file systems and does so using a framework known as the Virtual File System Switch, or VFS. This framework, which has its origins with (the now deceased) Sun's Solaris operating system, has become a standard interface used in UNIX between the kernel and various file system implementations, both local and remote.

PRELUDE: DISK DEVICES AND PARTITIONS

OS X and iOS follow the BSD convention of treating the hard disks as device nodes. Each disk can be accessed as a block device (/dev/disk#) or a character (raw) device (/dev/rdisk#). Likewise, partitions — or "slices" in UNIX-speak — can be accessed in a similar manner, both block and character, as /dev/[r]disk#s#.

Normally, disks and partitions are block devices. It is over the block device representation that the system can then mount(2) a file system. The raw mode is used primarily by low-level programs such as fsck(8) and pdisk(8), which need to seek and write directly to blocks.

Disk drivers also offer a standard ioctl(2) interface, defined in <sys/disk.h>, to allow for various query operations. The header is pretty well documented and defines the codes shown in Listing 15-1.

LISTING 15-1: The standard disk ioctl codes from <sys/disk.h>

```
/* Definitions
/*
/* ioctl                                     description
/* ------------------------------------- -------------------------------------
/* DKIOCEJECT                                eject media
/* DKIOCSYNCHRONIZECACHE                     flush media
/*
/* DKIOCFORMAT                               format media
/* DKIOCGETFORMATCAPACITIES                  get media's formattable capacities
/*
/* DKIOCGETBLOCKSIZE                         get media's block size
/* DKIOCGETBLOCKCOUNT                        get media's block count
/* DKIOCGETFIRMWAREPATH                      get media's firmware path
/*
/* DKIOCISFORMATTED                          is media formatted?
/* DKIOCISWRITABLE                           is media writable?
/*
/* DKIOCREQUESTIDLE                          idle media
/* DKIOCDISCARD                              delete unused data
/*
/* DKIOCGETMAXBLOCKCOUNTREAD                 get maximum block count for reads
/* DKIOCGETMAXBLOCKCOUNTWRITE                get maximum block count for writes
/* DKIOCGETMAXSEGMENTCOUNTREAD               get maximum segment count for reads
/* DKIOCGETMAXSEGMENTCOUNTWRITE              get maximum segment count for writes
/* DKIOCGETMAXSEGMENTBYTECOUNTREAD           // get max segment byte count, reads
/* DKIOCGETMAXSEGMENTBYTECOUNTWRITE          // get max segment byte count, writes
/*
/* DKIOCGETMINSEGMENTALIGNMENTBYTECOUNT  get minimum segment alignment in bytes
/* DKIOCGETMAXSEGMENTADDRESSABLEBITCOUNT get maximum segment width in bits
/*
/* DKIOCGETPHYSICALBLOCKSIZE                 get device's block size
/* DKIOCGETCOMMANDPOOLSIZE                   get device's queue depth
/*/
```

Using these is straightforward, as demonstrated by Listing 15-2:

LISTING 15-2: Using <sys/disk.h> ioctls to query information on a disk

```
#include <sys/disk.h> // disk ioctls are here..
#include <errno.h>    // errno!
#include <stdio.h>    // printf, etc..
#include <string.h>   // strncpy..
#include <fcntl.h>    // O_RDONLY
#include <stdlib.h>   // exit(), etc..

#define BUFSIZE 1024

// Simple program to demonstrate use of DKIO* ioctls:
// Usage: ... /dev/disk1 or ... disk1

void main (int argc, char **argv)
{
```

```
    uint64_t bs, bc,rc;
    char fp[BUFSIZE];
    char p[BUFSIZE];

    strncpy (p, argv[1], BUFSIZE);
    if(p[0] != '/') {
        snprintf(p, BUFSIZE -10,  "/dev/%s", p);
    }

    int fd = open(p, O_RDONLY);
    if(fd == -1) {
        fprintf(stderr, "%s: unable to open %s\n", argv[0], p);
        perror ("open");
        exit (1);
    }

    rc = ioctl(fd, DKIOCGETBLOCKSIZE, &bs);
    if (rc < 0)
        {
           fprintf (stderr, "DKIOCGETBLOCKSIZE failed\n"); exit(2);
        }
        else {
                fprintf (stderr, "Block size:\t%d\n",bs);
        }

    rc = ioctl(fd, DKIOCGETBLOCKCOUNT, &bc);
    fprintf (stderr, "Block count:\t%ld\n", bc);

    rc = ioctl(fd, DKIOCGETFIRMWAREPATH, &fp);
   fprintf (stderr, "Fw Path:\t%s\nTotal size:\t%ldM\n", fp, (bs * bc) / (1024 * 1024));


}
```

Note that obtaining the disk device for `ioctl()` requires read permission, which is normally not granted to non-root (or non-group operator) users.

Partitioning Schemes

File systems do not exist on their own. They reside in *partitions* on the disk. Every disk has at least one partition, and partitions can be individually formatted to contain file systems. In some cases, it is possible to have a file system span multiple partitions. A *partitioning scheme* defines the disk layout, logically segmenting the disk into one or more areas (hence, partitions) of contiguous sectors. Usually, this involves reserving the first several sectors of a disk for the partition table, which lists the areas (starting sector and sector count) and the file system type of each partition.

OS X traditionally supported three partitioning schemes:

- **Master Boot Record (MBR) partitioning:** MBR is a legacy of the old days of the PC XT and AT and is still widely used today. This partitioning scheme relies on a BIOS, is very limited (up to four partitions), and is 32-bit (for a maximum of 4 billion sectors), but it is supported across the board by all operating systems.

- **Apple Partition Map:** A custom, Apple-only scheme. Originally widespread in PPC-based Macs, it is also a 32-bit scheme and is Apple proprietary. It is now largely deprecated in favor of the next scheme, GPT, but still used for formatting Classic and Nano iPod devices.
- **GUID Partition Table (GPT):** A 64-bit scheme, which allows it to be used for disk sizes well into the exabyte range and beyond. It also effectively relieves any maximum partition restrictions. This is especially important: Both MBR and APT, being 32-bit schemes, allow for a maximum addressable 2^{32} sectors. Given the standard sector size is 512 bytes, this allows for disk sizes of up to 2 TB. Apple's default partitioning scheme has thus moved to a 64-bit architecture. GPT is also part of the EFI standard, which works well because Apple's Intel hardware is EFI-based.

  Some 32-bit systems, however (most notably Windows XP), still cannot support GPT. OS X on Intel, being EFI, supports it natively. As of 10.4, and as detailed in Apple Tech Note TN2166[4] ("Secrets of the GPT"), GPT has been favored by Apple as the default partitioning scheme.
- **Lightweight Volume Manager (LwVM):** An Apple-proprietary partition scheme, used in iOS 5 and later (as well as some older Apple TVs). Although it is proprietary and undocumented, it is fairly simple and has been reverse-engineered.

Kernel extensions can implement additional or custom partition schemes, by inheriting from IOKit's `IOPartitionScheme` class (itself a subclass of `IOStorage`, which contains it).

The MBR Partitioning Scheme

The Master Boot Record scheme, the last relic of the 16-bit days, is fast losing ground yet remains the default partitioning scheme in all other operating systems save OS X and 64-bit Windows. It is, without a doubt, the simplest partitioning scheme available. It reserves the first sector of the disk — the boot sector — for up to 440 bytes of bootstrap code that the BIOS uses to start up the machine. The 440 bytes typically read through the partition table, located at offset 446, and jump to the beginning of the partition, the *Partition Boot Record*, wherein operating system–specific code resides. The partition table is a fixed size — 64 bytes. This leaves only two more usable bytes — which are fixed to `0x55AA` — the MBR signature.

The MBR table is kept very simple. Because it is always 64 bytes, it allows for no more than four "primary" partition entries. Each entry is exactly 16 bytes long and describes the partition type, size, and address. The entries in the table provide the partition start and end address in one of two formats: Cylinder/Head/Sector (C/H/S) coordinates, or — more commonly — in Large Block Address (LBA) offsets. The latter is more often used, as the C/H/S scheme is limited to what, by today's standards, are fairly small drives.

If you have a portable hard drive, chances are it is MBR-formatted, and you can try the following in a terminal on the raw disk device (note that you will need to be root for read access). If not, you can always use OS X `hdiutil` to create an MBR-based image, as shown in Output 15-1. (Disk images, or `.dmg` files, are discussed later in this chapter.)

OUTPUT 15-1: Creating an MBR disk image with hdiutil

```
root@Ergo (/)# hdiutil create -layout MBRSPUD -megabytes 64 /tmp/testMBR.dmg
..........................................................................
created: /tmp/testMBR.dmg

root@Ergo (/)# ls -l /tmp/testMBR.dmg
-rw-r--r--@ 1 root  wheel  67108864 Jun 19 10:53 /tmp/testMBR.dmg
```

Using the `od` command, we can dump the file system; we care only about the first block, (up to offset `0x200`):

```
root@Ergo (/)# od -A x -t x1    /tmp/testMBR.dmg | more
0000000    00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
*
00001b0    00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  fe
00001c0    ff  ff  af  fe  ff  ff  01  00  00  00  ff  ff  01  00  00  00
00001d0    00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
*
00001f0    00  00  00  00  00  00  00  00  00  00  00  00  00  00  55  aa
```

Seeing as the image we created isn't bootable, the first 440 (`0x1b8`) bytes are all zero. Following them is an optional 32-bit disk signature (none in our case) and another reserved 2 bytes. At the unusual offset of `0x1be` is the partition table — *unusual*, because it is aligned on a 16, not a 32-bit boundary. Each entry is 16 bytes, and in the preceding example we have only one. Examining the previous output, and the record format below in Figure 15-1, you should quickly reach the conclusion that the partition is an HFS+ partition (`0xAF`), which is not bootable (`0x00`), starts at LBA block 1, and spans 131,071 blocks (64 MB).

offset	Purpose
`0x00`	Bootable flag (0×80)
`0x01`	Cylinder (10 bits) Head (6 bits) of first sector Sector (8 bits)
`0x04`	Partition Type
`0x05`	Cylinder
`0x06`	Head of last sector
`0x07`	Sector
`0x08`	LBA address of first sector
`0x0C`	Number of sectors

Type	Filesystem
..	...
`0x07`	NTFS/ex Fat
`0x0B`	Fat 32
`0x83`	Linux Ext
`0xAF`	HFS+
..	...

FIGURE 15-1: MBR partition format.

From the simple example provided, it should be obvious why MBR is a dying breed. It is not 32-bit optimized, it is limited to four primary partitions, extracting the C/H/S is not straightforward (requires multiple bit shifts), and the addressing and it is limited to 1023 cylinders, 63 heads, and 254 sectors. The only thing that permits MBR's survival so far is using LBA (Large Block Access) addresses of blocks, rather than C/H/S, as LBA can address up to 2 TB — but that, too, is fast becoming an obstacle as disk space grows ever more abundant by the day. Apple ran into these and other limitations fairly early on, which is why it adopted its own partitioning scheme — the Apple Partition Scheme.

The Apple Partitioning Scheme

The Apple Partitioning Scheme (APM) was designed by Apple as an alternative to MBR, meant to address the limitation of the four primary partitions and allow for LBA. Nowadays, you're generally less likely to run into any disks formatted with the Apple Partitioning Scheme, unless you have a PPC-based Mac or an iPod Classic or Nano. However, it is possible here, too, to use OS X's `hdiutil` tool to create a DMG file that is APM-formatted. You can then follow along on your device using the commands shown here in Output 15-2:

OUTPUT 15-2: Creating and attaching an Apple Partition Map formatted disk image

```
root@Minion (/)# hdiutil create -layout SPUD -megabytes 256 /tmp/testAPM.dmg
..........................................................................
created: /tmp/xx.dmg

root@Minion (/)# ls -l /tmp/testAPM.dmg
-rw-r--r--@ 1 root  wheel  268435456 Jun 19 07:13 /tmp/testAPM.dmg


root@Minion (/)# hdid -nomount /tmp/testAPM.dmg
/dev/disk4              Apple_partition_scheme
/dev/disk4s1            Apple_partition_map
/dev/disk4s2            Apple_HFS


root@Minion (/)# diskutil partitionDisk disk4 APM HFS+ "Test HFS+" 25% hfsx \
                      "Test HFSX" 25% jhfs+ "Journaled+" 25% free "ignored" 25%
Started partitioning on disk4
Unmounting disk
[ \   \   \   \   \   \   \   \   \   \   \   \   \   \  ]
[ \   \   \   \   \   \   \   \   \   \   \   \   \   \  ]
Creating partition map
Waiting for disks to reappear
Formatting disk4s2 as Mac OS Extended with name Test HFS+
Formatting disk4s3 as Mac OS Extended (Case-sensitive) with name Test HFSX
Formatting disk4s4 as Mac OS Extended (Journaled) with name Journaled+
[ / 0%..10%..20%..30%..40%..50%..60%..70%..80%.......... ]
Finished partitioning on disk4
/dev/disk4
   #:                       TYPE NAME                    SIZE       IDENTIFIER
   0:     Apple_partition_scheme                        *268.4 MB   disk4
   1:        Apple_partition_map                         32.3 KB    disk4s1
   2:                  Apple_HFS Test HFS+               67.1 MB    disk4s2
   3:                 Apple_HFSX Test HFSX               67.1 MB    disk4s3
   4:                  Apple_HFS Journaled+              67.1 MB    disk4s4
```

```
root@Minion (/)# hdid -nomount /tmp/testAPM.dmg/dev/disk4
                 Apple_partition_scheme
/dev/disk4s1                Apple_partition_map
/dev/disk4s2                Apple_HFS                         /Volumes/Test HFS+
/dev/disk4s3                Apple_HFSX                        /Volumes/Test HFSX
/dev/disk4s4                Apple_HFS                         /Volumes/Journaled+
```

> *You might also want to take a look at* `IOApplePartitionScheme.h` *in the* `IOStorageFamily` *driver* (`http://www.opensource.apple.com/source/IOStorageFamily/IOStorageFamily-24/IOApplePartitionScheme.h`).

In the example, we created a 256 MB disk image, initially with one partition, and then repartitioned it to three — each containing a separate file system type. Because the partition map itself uses up a partition (in the preceding example, `/dev/disk4s1`), we end up with four partitions, the usable ones being `/dev/disk4s2` through `/dev/disk4s4`. Technically, there is one more partition — to hold the free space, as there is a requirement in APM that all blocks on the disk be covered by a partition. The free space, however, is not accessible as a device node (that is, there is no `/dev/disk4s5` in the preceding example).

At the disk level, APM reserves the first block of the disk, block 0, for a special Driver Descriptor Map. This block 0, as defined in `<IOStorage/IOApplePartitionScheme.h>`, is identifiable by a fixed signature of ER (`0x4552`). The block is left largely unused, with the structure occupying only 82 out of the 512 of the block bytes. Typically, most of the structure fields are left as zero as well, with the only two important ones being the signature, blocksize, and block count, as you can see in Figure 15-2.

```
root@Ergo (/)# od -A x -t x1 /dev/disk4 | head -3
0000000    45  52  02  00  00  08  00  00  00  00  00  00  00  00  00  00
0000010    00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
*                       (rest is all zeroed out)
```

```
typedef struct Block0 {
   UInt16 sbSig;       /* (unique value for block zero, 'ER')*/
   UInt16 sbBlkSize;   /* (block size for this device)       */
   UInt32 sbBlkCount   /* (block count for this device)      */
   UInt16 sbDevType;   /* (device type)                      */
   UInt16 sbDevId;     /* (device id)                        */
   UInt32 sbDrvrData;  /* (driver data)                      */
   UInt16 sbDrvrCount; /* (driver descriptor count)          */
   DDMap sbDrvrMap[8]; /* (driver descriptor table)          */
```

FIGURE 15-2: APM's Block 0

As you can see from the previous example, our disk block size is 512 bytes (`0x0200`), and the disk contains 524,288 (`0x80000`) blocks — which is right on the mark, for a total of 256 MB.

The partition map can be found in the first block (offset `0x200` for a 512-byte block size). Each entry in it occupies one block. If you count one entry for the map itself, and another for the free space (`Apple_Free`), there will always be two more entries than usable partitions for example, five entries for the three in our example. (See Figure 15-3.)

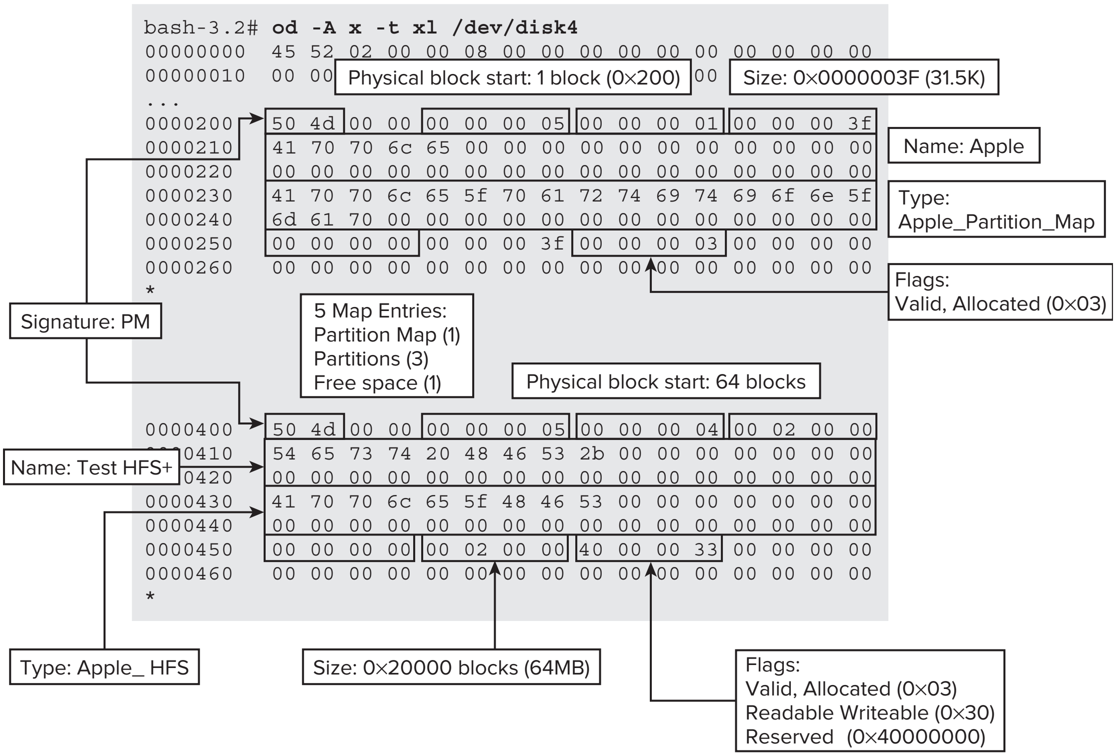


FIGURE 15-3: Apple Partition Map

The GPT Partitioning Scheme

The Globally Unique Identifier Partition Table (GUID PT, or GPT, for short), was developed as part of the Extensible Firmware Interface specification. When Apple moved to an Intel-based architecture, it made sense to adopt GPT rather than modify APM for larger disks. Indeed, Apple's Tech Note TN2166 effectively deprecated APM, stating that Apple could imagine disks with 2 TB becoming standard. While still ahead of its time, GPT is now used in OS X and in iOS alike.

GPT is fully specified as part of the Extensible Firmware Interface standard. EFI has already been discussed in detail in Chapter 6. The full specification of EFI also provides comprehensive detail of GPT. The system administration command `gpt(8)` can be used to manipulate GPT tables (although only to add/remove/label partitions, not resize them). (See Output 15-3.)

OUTPUT 15-3: The output of gpt(8). –v prints the first line, with device details

```
root@ergo (/)# gpt  -v show -l /dev/disk0s1
gpt show: /dev/disk0s1: mediasize=209715200; sectorsize=512; blocks=409600
   start    size  index  contents
       0       1         MBR
409599
```

To provide some backward compatibility with MBR, the first sector (LBA 0) of any GPT-formatted disk contains a "protective MBR." This defines for legacy operating systems the entire disk as an unknown partition (type 0xEE), thus preventing misclassification as an unformatted disk.

The actual GPT resides in the second sector (LBA 1). This sector contains the GPT header, which begins with the GPT magic string `EFI PART` (0x45 0x46 0x49 0x20 0x50 0x41 0x52 0x54) and contains the partition map details. Following the header is the partition map, which is simply an array of entries. These structures are defined in the IOKit framework's `storage/IOGUIDPartitionScheme.h`, as illustrated in Listing 15-3.

LISTING 15-3: The GPT header, from the IOKit framework's storage/IOGUIDPartitionScheme.h

```
struct gpt_hdr
{
    uint8_t  hdr_sig[8];
    uint32_t hdr_revision;
    uint32_t hdr_size;
    uint32_t hdr_crc_self;
    uint32_t __reserved;
    uint64_t hdr_lba_self;
    uint64_t hdr_lba_alt;
    uint64_t hdr_lba_start;
    uint64_t hdr_lba_end;
    uuid_t   hdr_uuid;
    uint64_t hdr_lba_table;
    uint32_t hdr_entries;
    uint32_t hdr_entsz;
    uint32_t hdr_crc_table;
    uint32_t padding;
};


struct gpt_ent
{
    uuid_t   ent_type;
    uuid_t   ent_uuid;
    uint64_t ent_lba_start;
    uint64_t ent_lba_end;
    uint64_t ent_attr;
    uint16_t ent_name[36];
};
```

GPT partitions can be named (or "labeled"), which allows for more flexibility when defining boot partitions. This avoids unbootable system scenarios that may result from rearranging the partitions or adding/removing disks.

Lightweight Volume Manager

The Lightweight Volume Manager (LwVM) is an Apple-proprietary partitioning scheme, which has inherited GPT as the default in iOS 5. It is conceptually somewhat similar to GPT but allows for partition encryption as well.

The proprietary format has been reverse-engineered by the developers of OpeniBoot and is known to be somewhat similar to Listing 15-4:

LISTING 15-4: The LwVM header

```
#define MAX_PARTITIONS     12

struct LwVM_MBR
{
  guid_t magic;              // One of two LwVM Magic "types"
  guid_t guid;               // 128-bit GUID for this device
  uint64_t mediaSize;        // Media size
  uint32_t numPartitions;    // Number of partitions defined (<= MAX_PARTITIONS)
  uint32_t crc32;            // CRC-32, if specified by a CRC-32 type.
  uint8_t padding[464];      // Padding to 512-byte block
} ;

// First block is followed by up to MAX_PARTITIONS records (of which
// numPartitions are actually defined)

struct LwVMPartitionRecord {
   guid_t   magic;             // Magic of partition, as per GPT
   guid_t   guid;              // GUID of partition, generated per device
   uint64_t startSector;
   uint64_t endSector;
   uint64_t attributes;
   char partitionName[64];
} ;

// The two types defined in iOS 5.0 iPod4,1: (0x80887910, 0x80887920)

#define LWVM_MAGIC { 0x6A, 0x90, 0x88, 0xCF, 0x8A, 0xFD, 0x63, 0x0A, 0xE3, 0x51,
0xE2, 0x48, 0x87, 0xE0, 0xB9, 0x8B }

#define LWVM_NO_CRC_MAGIC { 0xB1, 0x89, 0xA5, 0x19, 0x4F, 0x59, 0x4B, 0x1D, 0xAD,
0x44, 0x1E, 0x12, 0x7A, 0xAF, 0x45, 0x39 }
```

The only known attribute is `encrypted`, which specifies that the partition is encrypted and needs to be decrypted by the kernel.

For example, consider the output of `od(1)` in Output 15-4 on an iOS 5 system from a 64 GB device (the author's iPod Touch 64GB), with two partitions.

OUTPUT 15-4: The output of od(1) from an iOS 5 64 GB iPod, with LwVM fields highlighted and explained

```
root@Podicum (/)# od -A x -t x1  /dev/rdisk0  | more

0000000 6a 90 88 cf 8a fd 63 0a e3 51 e2 48 87 e0 b9 8b      LWVM Magic 128-bit
0000010 a8 e9 b0 f0 ba 20 bf cc d5 bd f8 46 d5 b1 76 58      Device GUID
0000020 00 80 34 09 0f 00 00 00 02 00 00 00 ad ab 86 28      CRC-32
0000030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
                                                              # of partitions
*          Media Size (61,587MB, for a 64G iPod)

0000200 48 46 53 00 00 00 11 aa aa 11 00 30 65 43 ec ac      HFSX Magic GUID
0000210 8f 52 e0 a1 a1 1f 4a 88 e1 1a fc e7 8c b0 60 6a      Partition GUID
0000220 00 80 00 00 00 00 00 00 00 e0 04 67 00 00 00 00
0000230 00 00 00 00 00 00 00 00 53 00 79 00 73 00 74 00      "System"
0000240 65 00 6d 00 00 00 00 00 00 00 00 00 00 00 00 00
0000250 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
*
0000280 48 46 53 00 00 00 11 aa aa 11 00 30 65 43 ec ac      HFSX Magic GUID
0000290 f0 ab dd 89 55 24 33 6f 24 d8 51 7b 11 af db f4      Partition GUID
0000300 00 e0 04 67 00 00 00 00 00 80 00 e8 0e 00 00 00
0000310 00 00 00 00 00 00 01 00 44 00 61 00 74 00 61 00      "Data"
0000320 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

          Attributes (first partition -  none, second partition encrypted)
```

LwVM is handled in iOS by a dedicated kernel extension, `LightweightVolumeManager.kext` (`com.apple.driver.LightweightVolumeManager`), which, like all kexts in iOS, is prelinked into the kernel.

CoreStorage

CoreStorage is a new partition type, introduced in Lion, which brings to OS X the much-needed support for logical volume management. CoreStorage partitions are logical volumes that can be dynamically extended or shrunk, allowing them to span several partitions. CoreStorage also enables full disk encryption (commonly referred to as FDE), and is required if FileVault 2's features are to be used. CoreStorage volumes may be created on GPT drives only, and HFS+ partitions must be journaled.

At present, the CoreStorage volume format is undocumented, though supported as of Lion. Partitions may be created with `diskutil(8)`, which has a new "corestorage" sub-command, wherein the commands shown in Output 15-5 may be used:

OUTPUT 15-5: CoreStorage verbs supported in Mountain Lion

```
root@simulacrum (/)# diskutil corestorage
Usage:  diskutil [quiet] coreStorage|CS <verb> <options>,
        where <verb> is as follows:

     list                     (Show status of CoreStorage volumes)
     info[rmation]            (Get CoreStorage information by UUID or disk)
```

continues

OUTPUT 15-5 *(continued)*

```
      convert                  (Convert a volume into a CoreStorage volume)
      revert                   (Revert a CoreStorage volume to its native type)
      create                   (Create a new CoreStorage logical volume group)
      delete                   (Delete a CoreStorage logical volume group)
      createVolume             (Create a new CoreStorage logical volume)
      deleteVolume             (Delete a volume from a logical volume group)
      encryptVolume            (Encrypt a CoreStorage logical volume)
      decryptVolume            (Decrypt a CoreStorage logical volume)
      unlockVolume             (Attach/mount a locked CoreStorage logical volume)
      changeVolumePassphrase   (Change a CoreStorage logical volume's passphrase)
 diskutil coreStorage <verb> with no options will provide help on that verb
```

The `encryptVolume` and `decryptVolume` verbs are new in Mountain Lion. The `deleteVolume` command was present in Lion, though undocumented. Additionally, `addDisk`, `resizeDisk`, `resizeVolume`, `resizeStack`, and `removeDisk` — undoubtedly all very useful, remain undocumented in both. If you try them, however, help on their usage will be displayed.

Conversion of a volume to CoreStorage is reversible (and may be undone using the `revert` verb), so long as encryption isn't involved.

In addition to `diskutil`, the `fsck_cs(8)` command is also provided as of Lion to check and repair CoreStorage partitions. The actual partition handling logic is provided by a kernel extension `CoreStorage.kext`, (also known as `com.apple.driver.CoreStorage`), with an addition `CoreStorageFsck` plug-in kext.

Using the `gpt(1)` command on a CoreStorage disk can display the partition structure. Output 15-6 shows the result of this command (on Snow Leopard, which does not support CoreStorage) on a CoreStorage formatted disk:

OUTPUT 15-6: Running gpt on a CoreStorage formatted disk

```
root@Ergo (/)# gpt show /dev/disk3
     start      size  index  contents
         0         1         PMBR
         1         1         Pri GPT header
         2        32         Pri GPT table
        34         6
        40    409600   1  GPT part - C12A7328-F81F-11D2-BA4B-00A0C93EC93B  # EFI System
    409640   3847656   2  GPT part - 53746F72-6167-11AA-AA11-00306543ECAC  # CoreStorage
   4257296    262144   3  GPT part - 426F6F74-0000-11AA-AA11-00306543ECAC  # Apple Boot
   4519440  27183567                                                       # Free Space
  31703007        32         Sec GPT table
  31703039         1         Sec GPT header
```

Inspecting partitions directly through their raw device reveals the structures associated with CoreStorage:

- The GPT GUID associated with CoreStorage is `53746F72-6167-11AA-AA11-00306543ECAC`. Viewed through the lens of `od -x`, this would appear as `6f72 5374 6167 11aa 11aa 3000 4365 acec`.

- The CoreStorage volume GUIDs also appear in the CoreStorage partition header. The GUIDs of the logical volume and the volume group are located at offset 304 and 320, respectively.
- The CoreStorage partition is actually an HFS+ file system implementation (HFS+ is covered in great detail in Chapter 16). It is not directly mountable, however, and mostly contains files intended for use by Spotlight. The ***hfsleuth*** tool on the book's companion website, which is specifically suited for debugging and showing HFS+ file system structures, can also be used to display CoreStorage partitions.

Reverse engineering CoreStorage, for the purposes of extending it outside OS X, is an ongoing project. You are welcome to check the book's companion website for the latest status and information.

GENERIC FILE SYSTEM CONCEPTS

Although different file systems take totally different approaches to managing files on the disk, all generally work with the same primitives. The kernel interface to files, called the Virtual FileSystem Switch (VFS) builds on these concepts.

Files

It should come as no surprise that the most fundamental concept in a file system is that of the file itself. A file, from the file system's point of view, is one or more arrays of blocks on the underlying media (disk, CD-ROM, or other). In the optimal case, a file would be a single, contiguous sequence of blocks. More often than not, however, files span multiple block ranges. These are generally referred to as *extents*. HFS+ also defines *clumps*, which are the default allocation blocks provided to a file when it is allocated or expanded.

Regardless of fragmentation, the file system must present the appearance of a file as a contiguous, freely seekable (random access) area. The requestor need not know anything of the underlying implementation. Indeed, some file systems are entirely virtual (such as Linux's `/proc`) while others can be mapped over the network (such as NFS or AFS). The requestor therefore obtains only a file descriptor (the `int fd` returned from `open(2)` or the `FILE *` returned from `fopen(3)`), but treats this is an opaque handle. The kernel, when serving the file requests, translates the handle into an identifier in the file system.

Extended Attributes

In addition to the normal file attributes, XNU's VFS supports the notion of extended attributes. These are user (or system) defined attributes, which can contain information used by applications, or — in many cases — the system itself. Extended attributes are used in Darwin to support advanced features, such as transparent compression and forks (both discussed in the next chapter), as well as Access Control Lists (discussed next).

Permissions

Not all files are created equal. Some files contain potentially sensitive information, and every self-respecting file system (with the exception of the FAT family) must support permissions. UNIX file systems, which Mac's native HFS+ is one of, support the traditional user/group/other read/write/execute model. This is a fairly primitive model, as it only allows you to set permissions for a single user and a single group — casting everybody else into the "other" category.

As of OS X 10.4, however, VFS, adds support for finer-grained permissions, similar to the well-known NTFS permissions, but complying with the POSIX 1.e security standard. These are commonly referred to as *Access Control Lists*, or ACLs. OS X allows the setting and modification of ACLs using `chmod(1)`. The access control lists can be displayed using `ls(1) -e`. Files with ACLs appear in the output of `ls(1) -l` with a plus (+) sign. VFS relies on extended attributes to support ACLs, and their enforcement is performed by a separate mechanism called KAUTH (`bsd/kern/kern_authorization.c`).

Timestamps

A file system needs to record timestamps for the various files it contains. UNIX calls for three timestamps to be maintained: Creation, Modification, and Access. These are the familiar `-acm` switches from the `touch(1)` command and can be displayed with `ls(1)` when using `–u` (access), `-U` (creation), or neither (modification).

Shortcuts and Links

Most UNIX users are familiar with links, both soft (also called "symbolic") and hard. Soft links are created with `ln(1) -s`, whereas their hard siblings are created without the switch. From the VFS perspective, a soft link is a different file (i.e. another inode), of type `l`, containing the name of the file pointed to. Hard links, on the other hand, are another directory entry, pointing to the same underlying file (or, as you will see from the VFS perspective, the same inode). Another way of looking at it is that hard links exist at the directory level, whereas soft links exist at the file level. (See Figure 15-4.)

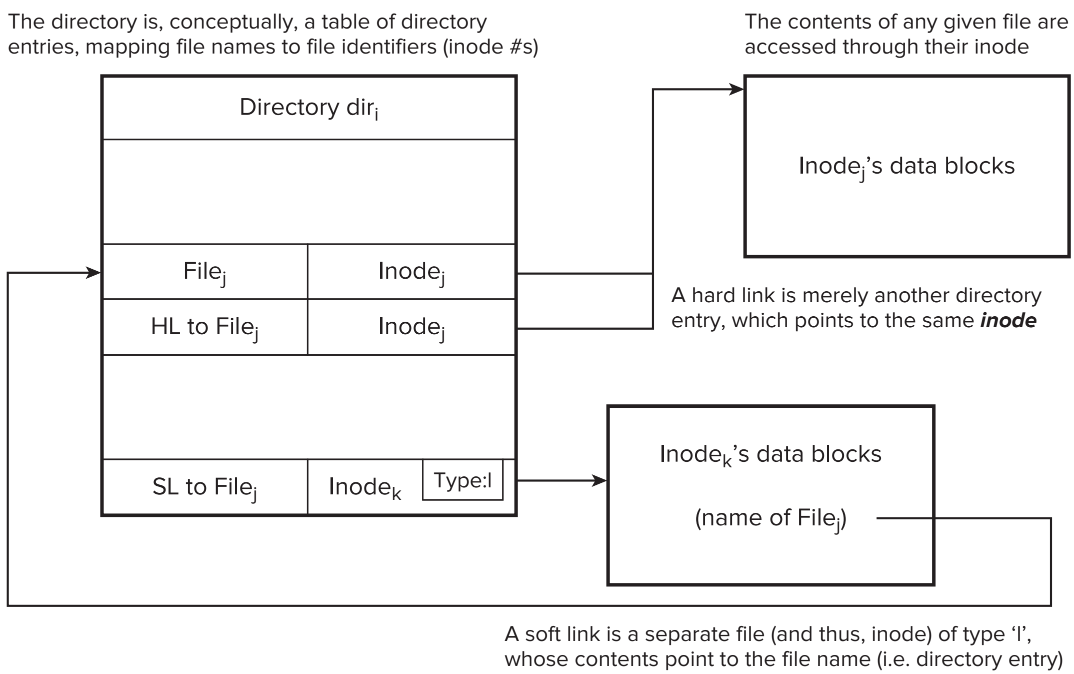


FIGURE 15-4: Visualizing hard and soft (symbolic) links

Hard links provide a mechanism, as soft links do, for setting up shortcuts to files. Unlike soft links, however, hard links prevent the accidental deletion of a file, as a file will only be removed from the file system when the very last link to it has been removed. Table 15-1 illustrates the differences between the link types:

TABLE 15-1: Hard and Soft Links Compared

	SOFT	HARD
Inode	Different directory entry (dentry) to different inode, containing name	Different dentry to same inode
Scope	Across file systems	Same file system
Directories	Linkable	Officially, no (only "." and ".."). In practice, implementations differ
On target rm/mv	Soft link breaks	Hard link persists
On target recreation	Soft link "heals"	Hard link points to "old" file.
Find with	`find –L -samefile <target>`	`find –samefile <target>` `find –inum <targetinodenum>`

A detailed discussion on symbolic and hard links can be found in the manual page for `symlink(7)`.

FILE SYSTEMS IN THE APPLE ECOSYSTEM

OS X and iOS both support myriad file systems. Essentially, any number of file systems can be supported, thanks to the kernel's modularity, as long as they all adhere to the standard kernel of VFS (which is described next). In this section, we detail those file system types.

Unless otherwise stated, file systems can be loaded with a `mount_`*`xxx`* command (with *`xxx`* being the name of the file system in question). The actual file system support is provided by a kernel extension (from `/System/Library/Extensions`, usually named *`xxx`*`fs.kext`). An additional directory, `/System/Library/Filesystems`, holds subdirectories for the specific file systems, in which corresponding "util" binaries are provided for file system maintenance.

Native Apple File Systems

Apple has traditionally used its own file systems as far back as the earliest days of the Mac. Support for these file systems is still present in OS X.

Hierarchical File System (HFS)

The Hierarchical File System (HFS) was the native file system structure developed by Apple to use in the early days of Mac OS, before the present age of OS X. Nowadays, it's an obsolete file system, having been superseded by HFS+, described next.

Hierarchical File System Plus (HFS+)

As disk storage increased exponentially, HFS proved to be a very limited file system. This called on Apple to develop quite a few extensions to overcome the limitations, and provide for better, full 32-bit and potentially 64-bit functionality. The result of these improvements is Hierarchical File System Plus (HFS+).

HFS+ has been, and at the time of writing still is, the native file system on Apple's products. From the lowly iPod Nanos through the iPads and Macs, HFS+ (or its case-sensitive variant, HFSX) is widely used. Because it is so ubiquitous, this book dedicates the entire next chapter to unraveling its inner workings.

Outside Apple's products, the adoption of HFS+ is low, not to say virtually non-existent. There are various implementations of HFS+, most notably for Linux and Windows (including one written by the author, but remaining closed source), but as a whole the file system has very limited adoption.

HFS+ and its variant, HFSX, are both supported in OS X natively, as part of the kernel. The implementation is in XNU's `bsd/hfs` directory.

DOS/Windows File Systems

The non-Apple world has always been dominated by Microsoft — and likewise its file systems were the de facto standard. Apple had little choice but to support these systems in Mac, and still does, to the present day.

File Allocation Table (FAT)

The File Allocation Table (FAT) is one of the simplest and oldest file systems in use. Because of its relatively low overhead in small volumes, it was the file system of choice back in the days of floppy disks, and — as a result of its simple implementation — is still widely used in mobile media, such as SD cards and most USB flash drives.

The most recognizable trait of FAT is its short file names — what became to be known as "8.3" — wherein the file name is limited to eight characters, and an optional extension, up to three characters. Another limitation of the basic FAT is that it is limited to 2 GB, and — even if stretched — cannot go past 4 GB volumes, which are paltry by today's standards.

Over the years, Microsoft, the chief developer of FAT, found itself bogged down in the quagmire of backward compatibility. This led to FAT being modified into various variants. From the original FAT-12 (a 12-bit file system suited for use in the 1980s era of 640 k), through FAT-16, or simply, "FAT," which was the native file system in most incarnations of DOS. Windows 95 brought along VFAT (to accommodate long file names), followed by FAT-32 (to overcome the measly 2–4 GB volume size, and raise the bar to 2 TB).

FAT, in all of its basic variants discussed so far, is supported in OS X by means of the `msdosfs` kernel extension.

Since FAT-32, the most popular FAT type, is still limited to 2 TB volumes — and larger hard drives are presently available — it is being phased out in favor of ExFAT, a new system with a theoretical limit of 64 ZetaBytes. Because 1 ZetaByte is 2^{70} bytes (or one Giga-TeraByte), ExFAT should last for a while. ExFAT has been especially designed for Flash drives, taking into consideration the limitations of the Flash medium.

Mac OS X supports ExFAT as of later releases of Snow Leopard and Lion, with the `exfat` kernel extension and the `mount_exfat(8)` command.

NT File System (NTFS)

Windows NT was Microsoft's first multiuser operating system, and FAT (back then, in its 16-bit incarnation) proved vastly inadequate for its needs. The main features missing from FAT were *permissions* and *quotas*. Permissions were required to allow discretionary access control to files. Quotas are a mechanism to restrict users from abusing a shared file system and cluttering it up with too many files.

To meet both ends, Microsoft introduced the NT File System, which has become the native file system in all its operating systems as of Windows 2000.

Apple provides a driver for NTFS — `ntfs.kext` — but it only supports read-only operations. (Snow Leopard had experimental write, but Lion seems to have disabled it.) Both commercial and freeware drivers for NTFS exist, offering the much needed full read-write capability.

CD/DVD File Systems

CDs and DVDs have used their own proprietary file systems, depending on media type and usage.

The CD-Audio File System (CDDAFS)

Audio CDs can be mounted just like CD-ROMs. The audio tracks themselves appear as files, in AIFF format. A "cat" on the AIFF files provides the raw CD data (which is how iTunes can rip, or "import" CD tracks into its library).

If the iTunes database can be consulted, the files actually have the same names as the audio track they correspond to, and the volume is named like the CD (a wicked cool feature for command line users, in one writer's humble opinion). Otherwise, the generic "Audio CD" is used for the volume name, and "# Audio Track" for the tracks (with # being the track number). The track name resolution is done in user mode (as one would expect), and the names are passed to the `mount_cddafs(8)` utility as arguments.

The mounted CD file system has an additional, hidden file, `.TOC.plist`, which is generated by the kext (`CreateNewXMLFile()` in `AppleCDDAFileSystemUtils.c`). The file is an XML `.plist` containing the CD sessions (usually only one) and track listing. Output 15-7 shows such a CD listing:

OUTPUT 15-7: A CDDA FS

```
morpheus@Ergo (/)$ ls -a /Volumes/Favorite\ Piano\ Concertos/
.       .TOC.plist            2 Saint-Saëns Op. 29.aiff
..      1 LVB Op. 61a.aiff    3 Bruch Op. 88b.aiff
morpheus@Ergo (..ertos/)$ file 1\ LVB\ Op.\ 61a.aiff
LVB Op. 61a.aiff: IFF data, AIFF-C compressed audio
morpheus@Ergo (/)$ head .TOC.plist
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/
DTDs/PropertyList-1.0.dtd">
        <plist version="1.0">
        <dict>
        <key>Format 0x02 TOC Data</key>
        <data>
        AGUBAQEAKAAAA.. // Base 64 encoded data, followed by track "map"
```

CD-ROM File System (CDFS/ISO-9660)

The CD-ROM File System is supported in the `cd9660.kext` kernel extension. It is loaded by the mount_cd9660 program. "9660" refers to the ISO standard of the same number, which defines the format used by CD-ROMs (or, at least when CD-ROMs were still widely used).

Universal Disk Format (UDF)

UDF is a file system format developed for DVDs. UDF exists in several versions. Mac OS X supports all of them — up to and including the latest, 2.60, as of Tiger.

Network-Based File Systems

Network file systems are used to extend storage to reach beyond the local host, and onto remote hosts, which may be on the local area network or on the far side of the Internet.

Up until Snow Leopard, OS X used the private frameworks of URLMount and URLAccess, but have since shifted to a public NetFS framework. (Snow Leopard still contains the private frameworks, but Lion drops them.)

Apple Filing Protocol

Apple's own Apple Filing Protocol (AFP) was the default network file system in Mac OS 8 and 9, where it was known as AppleShare. This is an application protocol, originally carried over Apple's proprietary AppleTalk protocol (before Apple joined the rest of humanity in embracing TCP/IP). It currently uses TCP ports 427 or 528.

AFP has undergone several revisions, with version 3.0 being released along with the first versions of OS X server. Since then, it has been further revised to work in conjunction with HFS+'s extended attributes, and, more recently, Apple's Time Machine for backups.

AFP URLs adopt the form `afp://`. In the `mount(8)` and `df(1)` commands, AFP file systems appear as `afp_xxx` in Output 15-8

OUTPUT 15-8: AFP file system mount

```
morpheus@Ergo (/)$ df
File system                        512-blocks      Used Available Capacity  Mounted on
/dev/disk0s2                       489562928  471302120  17748808    97%    /
..
afp_0W9DWS1qQM2m00kG0H0Pyetl-1.300 1949330784 1556003544 393327240  80%  /Volumes/Nexus
```

Network File System

Network File System (NFS) is a veteran application level protocol that was developed back in the day by Sun Microsystems (now a division of Oracle). NFS, which started life as RFC 1094, underwent several revisions before becoming the *de facto* standard network file system of choice in UNIX with NFSv3 (RFC 1813), and later with NFSv4 (RFC 3010). It has rather recently received improvements for clusters, with NFSv4.1 (RFC 5661).

Mac OS supports NFSv3 natively, as part of XNU in the `bsd/nfs/` directory. Snow Leopard provided partial support for NFSv4, and Lion claims full support.

Server Message Block (SMB/CIFS/SMB2)

Microsoft's network file system implementation is built on top of the Server Message Block protocol, or SMB. This protocol, which originated in the good old days of LAN Manager and NetBIOS (i.e., the 1980s!) is still backward compatible, and relies on NetBIOS (an even more archaic protocol, RFC1001-1002, which predates DNS for naming services).

Microsoft rebranded SMB as the rather ambitious Common Internet File System (CIFS), which is by no means common on the Internet but definitely makes for a more catchy acronym. The differences between the two are minor, with the major difference being the ability to run natively over TCP (port 445) and do without NetBIOS.

Even reincarnated as CIFS, SMB is still woefully inefficient, primarily due to many messages associated with each transaction. With Vista, the protocol has been further modified, and — back to its origin — is now known as SMB2.

SMB and CIFS are both supported with `smbfs.kext`, which handles all the SMB client requests.

For server features, prior to Lion, Apple has relied on SAMBA, an open source package, to allow OS X to emulate Windows in serving shares. This support has been discontinued with Lion, primarily due to licensing issues associated with the GNU Public License (GPLv3). Lion now supports SMB using an Apple proprietary implementation, called SMBX. The binary (`/usr/sbin/smbd`) has been completely rewritten.

File Transfer Protocol

FTP (RFC959), is one of the Internet's oldest protocols. In the 1980s and early 1990, it accounted for the most traffic, but has since been pushed back by HTTP and SMTP. OS X still offers support for it and even abstracts it so that instead of the usual get and put of an FTP client, FTP server files can be made visible as regular files on an FTP file system.

Web Distributed Authoring and Versioning

Web Distributed Authoring and Versioning (WebDAV) is a proposed extension to HTTP, which adds to the latter various methods that can be used to upload files (via `PUT`), create folders (`MKCOL`), and search (`PROPFIND`). Originally defined in RFC2518, WebDAV was criticized for security issues, but has become increasingly more popular with the advent of the Cloud computing infrastructures. Slightly modified in RFC4918, it serves as the basis for many web-borne file systems, most notably Microsoft's Web Folders, Amazon's S3 services, and Apple's (now defunct) MobileMe.

Pseudo File Systems

Pseudo file systems aren't file systems at all. Rather, they can be seen as one of two types:

- **A file-based interface to kernel data structures and devices:** Linux-savvy readers are no doubt familiar with Linux's `/proc` and `/sys`, which provide a plethora of diagnostic data and kernel parameters. Other UNIX-philes likely know `/dev`, by means of which the kernel exposes its various device drivers.

- **File system components:** These are not file systems at all, but they provide mechanisms for handling special file types or special mount options. BSD's (and XNU's) deadfs, specfs, FIFOfs, and unionfs fall into this category.

XNU compiles-in support for several pseudo file systems. These can be found in the `bsd/miscfs` directory and are discussed next.

The devfs File System

The device file system is used to host the various BSD device files — character and block. These files are necessary for user-mode representation of hardware devices, allowing utilities to access hardware — primarily the disk (`/dev/disk##` or `/dev/rdisk##`) and the terminal (`/dev/tty##`). The device file system is also home to the fdesc filesystem, which lets processes access their own file descriptors using `/dev/fd/##` (see `mount_fdesc(8)` command).

Typically, the kernel creates devices automatically (responding to plug-and-play events), but the user may also create device nodes with the `mknod(1)` utility or the `mknod(2)` system call. The block and character devices are represented by bdevsw and cdevsw structures (respectively) defined in bsd/sys/conf.h.

devfs exports four functions, as shown in Table 15-2.

TABLE 15-2: devfs Exported Functions

DEVFS FUNCTION	USED FOR
`devfs_make_node`	Creating a device node (`DEVFS_CHAR` or `DEVFS_BLOCK`). The function returns an opaque handle, which must be kept until the device is removed.
`devfs_make_node_clone`	As `devfs_make_node`, but with a "clone" function used to update the device minor on creation.
`devfs_remove`	Remove a previously created device, specified by the handle returned by the make function.
`devfs_make_link`	Link to an already existing device. This function is `BSD_KERNEL_PRIVATE`, and unused in XNU.

The FIFOfs vnode Type

FIFOs are the UNIX implementation of "named pipes." Anonymous pipes can be created with the `pipe(2)` system call, but cannot be shared across unrelated processes. Instead, `mkfifo(2)` can be used to create a pipe special file. The special file exists only to ensure global uniqueness — that is, that unrelated processes can access the pipe by some name, which is available system-wide, with no naming conflicts.

The FIFOfs implementation is simply a set of vnode operations (in `bsd/miscfs/fifofs/fifo_vnops.c`). These operations (discussed in detail later, under VFS) are the callbacks that are executed by the kernel when a corresponding system call is executed on the file in question. In the case of FIFOfs, these vnode operations override the default vnode operations by nullifying some, voiding others, and providing default implementations for the rest. These are declared in `bsd/miscfs/fifofs/fifo.h`. This is shown in Output 15-9:

OUTPUT 15-9: The FIFOfs implementation

```
/*
 * This structure is associated with the FIFO vnode and stores
 * the state associated with the FIFO.
 */
struct fifoinfo {
    unsigned int    fi_flags;
    struct socket   *fi_readsock;
    struct socket   *fi_writesock;
    long        fi_readers;
    long        fi_writers;
        unsigned int    fi_count;
};
...
/*
 * Prototypes for fifo operations on vnodes.
 */
// Note that each of these operations correspondds to a system call,
// or system call with flags:

// e.g. fifo_create for open (..., O_CREAT), fifo_mmap for mmap(2), etc..
int     fifo_ebadf(void *);

#define fifo_create (int (*) (struct  vnop_create_args *))err_create
#define fifo_mknod (int (*) (struct  vnop_mknod_args *))err_mknod
#define fifo_access (int (*) (struct  vnop_access_args *))fifo_ebadf
#define fifo_getattr (int (*) (struct  vnop_getattr_args *))fifo_ebadf
#define fifo_setattr (int (*) (struct  vnop_setattr_args *))fifo_ebadf
#define fifo_revoke nop_revoke
#define fifo_mmap (int (*) (struct  vnop_mmap_args *))err_mmap
#define fifo_fsync (int (*) (struct  vnop_fsync_args *))nullop
#define fifo_remove (int (*) (struct  vnop_remove_args *))err_remove
#define fifo_link (int (*) (struct  vnop_link_args *))err_link
#define fifo_rename (int (*) (struct  vnop_rename_args *))err_rename
#define fifo_mkdir (int (*) (struct  vnop_mkdir_args *))err_mkdir
#define fifo_rmdir (int (*) (struct  vnop_rmdir_args *))err_rmdir
#define fifo_symlink (int (*) (struct  vnop_symlink_args *))err_symlink
#define fifo_readdir (int (*) (struct  vnop_readdir_args *))err_readdir
#define fifo_readlink (int (*) (struct  vnop_readlink_args *))err_readlink
#define fifo_reclaim (int (*) (struct  vnop_reclaim_args *))nullop
#define fifo_strategy (int (*) (struct  vnop_strategy_args *))err_strategy
#define fifo_valloc (int (*) (struct  vnop_valloc_args *))err_valloc
#define fifo_vfree (int (*) (struct  vnop_vfree_args *))err_vfree
#define fifo_bwrite (int (*) (struct  vnop_bwrite_args *))nullop
#define fifo_blktooff (int (*) (struct vnop_blktooff_args *))err_blktooff
```

continues

OUTPUT 15-9 *(continued)*

```
// the following operations are provided for fifos:
int     fifo_lookup (struct vnop_lookup_args *);
int     fifo_open (struct vnop_open_args *);
int     fifo_close (struct vnop_close_args *);
int     fifo_read (struct vnop_read_args *);
int     fifo_write (struct vnop_write_args *);
int     fifo_ioctl (struct vnop_ioctl_args *);
int     fifo_select (struct vnop_select_args *);
int     fifo_inactive (struct  vnop_inactive_args *);
int     fifo_pathconf (struct vnop_pathconf_args *);
int     fifo_advlock (struct vnop_advlock_args *);
```

The specfs vnode Type

Similar to FIFOs, device special files (VBLK and VCHR) are given their "personality" and vnode operations by the custom specfs. In much the same way, most of the vnode operations defined in `bsd/miscfs/specfs/specdev.h` are nullified or voided, with the rest given default implementations. This is shown in Output 15-10:

OUTPUT 15-10: Implementations of the specfs

```
morpheus@Ergo (...xnu/1699.26.8)$  cat bsd/miscfs/specfs/specdev.h | grep ^int
 // the following are BSD_KERNEL_PRIVATE
int spec_blktooff (struct  vnop_blktooff_args *);
int spec_offtoblk (struct  vnop_offtoblk_args *);
int    spec_fsync_internal (vnode_t, int, vfs_context_t);
int spec_blockmap (struct  vnop_blockmap_args *);
int spec_kqfilter (vnode_t vp, struct knote *kn);
 // and the rest are visible kernel-wide
int    spec_ebadf(void *);
int    spec_lookup (struct vnop_lookup_args *);
int    spec_open (struct vnop_open_args *);
int    spec_close (struct vnop_close_args *);
int    spec_read (struct vnop_read_args *);
int    spec_write (struct vnop_write_args *);
int    spec_ioctl (struct vnop_ioctl_args *);
int    spec_select (struct vnop_select_args *);
int    spec_fsync (struct  vnop_fsync_args *);
int    spec_strategy (struct vnop_strategy_args *);
int    spec_pathconf (struct vnop_pathconf_args *);
```

The deadfs vnode Type

deadfs is used primarily in the implementation of the `revoke(2)` system call. This system call, which is supported only on devices, invalidates all existing open file handles on the given device file. To do so, the kernel maps the vnode operations of the corresponding vnode to the `dead_vnodeop_entries`, defined in `bsd/miscfs/deadfs/dead_vnops.c`. Subsequent read/write operations on the vnode then fail.

The main use of revocation is to instantiate a terminal for login. Because most terminals are pseudo terminals, they are created and released frequently, and the system must ensure that a new terminal instance has no previous owner.

The unionfs Layering Mechanism

unionfs is a special mechanism for layering: It allows the mounting of more than one file system on the very same mount point, overlaying one on top of the other, so that both file systems' files are visible. In the event of conflicting files with the same name, the file from the top-most mounted file system in the union hides the one beneath it. Any file system can be union-mounted by specifying the `-o union` option to mount.

The union file system is not an Apple-specific system and exists in Linux as well as BSD. It has nonetheless played a pivotal role in facilitating the jailbreaking of iOS. Comex (who has since defected, to work for Apple) used the union technique to speed up the jailbreak time of JailBreakMe 3.0 and avoid the need to reboot the device.

MOUNTING FILE SYSTEMS (OS X ONLY)

OS X supports the dynamic mounting and unmounting of file systems, using two mechanisms — the UNIX standard automount, and the OS X–specific diskarbitrationd. OS X also supports the UN*X mechanism of /etc/fstab, but it not present unless manually created, and is deprecated.

Automount

OS X's automount is a direct port of the UNIX automount that can be found in Solaris, BSD, and Linux.

The kernel component of automounting is carried out by the `autofs.kext` kernel extension, which registers the `autofs` file system with VFS. It exposes `/dev/autofs` to user mode.

In user mode, several daemons have to cooperate for the automounting operation to succeed:

- **`autofsd`:** Starts from `launchd`, is responsible for listening on network configuration change notifications and calling automount.
- **`autmount`:** Consults the `/etc/auto_master` file to request particular mounting operations and `automountd` to perform the actual mount.

Disk Arbitration

Even on Macs without network access, automounting is commonplace: The nearly magical automounting functionality triggered by the addition or removal of a USB device is well known. Simply plug in the device, wait for a few seconds, and it appears in the Finder, as well as in `/Volumes`.

The dirty work behind the plug and play magic is performed by the Disk Arbitration Daemon, the aptly named `diskarbitrationd`. This daemon, started by `launchd(8)`, is responsible for listening in on notifications from multiple sources, including the kernel — specifically I/O Kit. The notifications are primarily for matches on IOMedia class devices, which are devices that represent underlying media, such as USB drives, hard disks, and the like.

When a notification is received, the `diskarbitrationd` queries the file system of the device in question, and — if it is recognized — proceeds and attempts to mount it, using the corresponding file system's handler. Third parties can also register with `diskarbitrationd` using the `DiskArbitration.framework` miscellaneous `DARegister*` functions, to receive notification of disk-related events. These events include disk Appeared, Disappeared, Mount, Unmount, Eject, and Peek. The Peek enables its caller to potentially exclusively lock the device (by calling `DADiskClaim`).

A good way to peek into `diskarbitrationd` is to start it with the `-d` command line. This can easily be done by editing `launchd`'s `com.apple.diskarbitrationd.plist`. Messages are logged to `/var/log/diskarbitrationd.log`. A sample log is shown in Output 15-11.

OUTPUT 15-11: Sample log output from diskarbitrationd

```
14:36:34 server has been started.
14:36:34   console user = none
14:36:34
14:36:34 filesystems have been refreshed.
14:36:34   created filesystem, id = /System/Library/Filesystems/afpfs.fs/.
14:36:34   created filesystem, id = /System/Library/Filesystems/cd9660.fs/.
14:36:34   created filesystem, id = /System/Library/Filesystems/cddafs.fs/.
14:36:34   created filesystem, id = /System/Library/Filesystems/exfat.fs/.
14:36:34   created filesystem, id = /System/Library/Filesystems/ftp.fs/.
14:36:34   created filesystem, id = /System/Library/Filesystems/hfs.fs/.
14:36:34   created filesystem, id = /System/Library/Filesystems/msdos.fs/.
14:36:34   created filesystem, id = /System/Library/Filesystems/nfs.fs/.
14:36:34   created filesystem, id = /System/Library/Filesystems/nofs.fs/.
14:36:34   created filesystem, id = /System/Library/Filesystems/ntfs-3g.fs/.
14:36:34   created filesystem, id = /System/Library/Filesystems/ntfs.fs/.
14:36:34   created filesystem, id = /System/Library/Filesystems/smbfs.fs/.
14:36:34   created filesystem, id = /System/Library/Filesystems/udf.fs/.
14:36:34   created filesystem, id = /System/Library/Filesystems/ufs.fs/.
14:36:34   created filesystem, id = /System/Library/Filesystems/webdav.fs/.
14:36:34
14:36:34 iokit [0] -> diskarbitrationd [13]
14:36:34   created disk, id = /dev/disk0s2.
14:36:34   created disk, id = /dev/disk0s1.
14:36:34   created disk, id = /dev/disk0.
14:36:34
14:36:34 diskarbitrationd [13] -> diskarbitrationd [13]
14:36:34   probed disk, id = /dev/disk0s2, with hfs, ongoing.
14:36:34   probed disk, id = /dev/disk0s2, with hfs, success.
14:36:34
14:36:35 kextd [10]:13827 -> diskarbitrationd [13]
14:36:35   created session, id = kextd [10]:13827.
14:36:35   registered callback, id = 000000010000638F:0000000000000000, kind =
disk unmount approval.
14:36:35   set client port, id = kextd [10]:13827.
14:36:35
14:36:35 kextd [10]:14339 -> diskarbitrationd [13]
14:36:35   created session, id = kextd [10]:14339.
14:36:35   registered callback, id = 0000000100005B62:0000000000000000, kind =
disk appeared.
14:36:35   registered callback, id = 00000001000060E1:0000000000000000, kind =
```

```
disk description changed.
14:36:35   registered callback, id = 0000000100005A6C:0000000000000000, kind =
disk disappeared.
14:36:35   set client port, id = kextd [10]:14339.
```

`diskarbitrationd` also allows user clients to participate in mount decisions, potentially blocking any disk mount attempts. Calling `DARegisterDiskMountApprovalCallback` allows a programmer to not only be notified of a disk mount/unmounts operation but also potentially block it. Blocking is a simple matter of creating a dissenter object (using `DADissenterCreate`), and returning it from the approval callback.

The Disk Arbitration framework hides the underlying notification from the kernel driver layer, I/O Kit. Rather than using disk arbitration, it is possible to register for notifications directly from I/O kit. This is discussed in Chapter 19.

DISK IMAGE FILES

OS X makes use of disk images, which typically have a `.dmg` extension. These files are, in essence, complete file systems — usually HFS+ — in a single file. The file format is called UDIF — Universal Disk Image Format — but, surprisingly, remains undocumented and proprietary to Apple. DMG files may be internally compressed (usually with bzip2 compression), and can contain internal license files which Apple's utilities will display on opening. The format has been reverse-engineered sufficiently, however, to allow for third-party tools such as Catacombae.org's `dmgextractor` to offer support for most of the DMG file format idiosyncrasies.

OS X's finder can automatically attach DMGs when double-clicked (by calling CoreServices' `DiskImageMounter.app`), as can the `hdiutil(1)` command, using the `attach` verb. (The `hdiutil` command can also create DMG files, as shown earlier in this chapter.) The attachment is carried out by `DiskImages.framework`, which is a private framework.

The BSD layer offers native support for disk images in its vnode disk driver, which is accessible through the user mode `/usr/libexec/vndevice` command. This command allows attaching a disk image to one of the BSD `/dev/vn*` devices.

Despite the native support, Apple prefers to support DMG files through a custom, proprietary kernel extension. This extension, `IOHDIXController.kext`, which registers itself as `com.apple.driver.DiskImages`, remains closed source. The advantage of using the external kext is that, unlike the vnode disk driver, it can handle compressed and/or encrypted images. While IOHDIXController is intentionally undocumented by Apple, it has been sufficiently reverse engineered to allow — via I/O Kit — attaching DMGs, including on iOS.

Raw DMG Files

The DMG extension is a misleading one. Most DMGs are in proprietary format (sometimes incorrectly identified by `file(1)` as "VAX COFF executable." Others are raw file system images — verbatim copies of the file system blocks, as output of `dd(1)`, and may be further compressed. Double clicking these DMGs (or using the equivalent command, `open(1)`) will fail to attach them. Using `hdiutil(1)`, however, you can force attachment by adding `-imagekey diskimage-class=CRawDiskImage` to the command line. This is especially useful in the case of iOS DMGs, which (when decrypted) can be mounted in this way, as shown in Output 15-12:

OUTPUT 15-12: Attaching the raw ramdisk image of an unencrypted iOS 5.1 restore disk

```
root@Ergo (/)# file ~/iOS/5.1.restore.ramdisk.dmg
/Users/morpheus/iOS/5.1.restore.ramdisk.dmg: Macintosh HFS Extended version 4 data
(mounted) last mounted by: '10.0', created: Wed Feb 15 05:26:23 2012,
last modified: Tue Apr  3 11:16:04 2012, last checked: Wed Feb 15 08:26:23 2012,
block size: 4096, number of blocks: 4218, free blocks: 0

root@Ergo (/)# hdiutil attach ~/iOS/5.1.restore.ramdisk.dmg    hdiutil displays fear of attachment..
hdiutil: attach failed - not recognized

root@Ergo (/)# hdiutil attach ~/iOS/5.1.restore.ramdisk.dmg
-imagekey diskimage-class=CRawDiskImage
/dev/disk3                                   /Volumes/ramdisk
                                                                ..unless coerced with -imagekey
root@Ergo (/)# hdiutil info
image-path      : /Users/morpheus/iOS/5.1.restore.ramdisk.dmg
image-alias     : /Users/morpheus/iOS/5.1.restore.ramdisk.dmg
shadow-path     : <none>
icon-path       : /System/Library/PrivateFrameworks/DiskImages.framework/Resources
   /CDiskImage.icns
image-type      : read/write
system-image    : false
blockcount      : 33748
blocksize       : 512
writeable       : TRUE
autodiskmount   : TRUE
removable       : TRUE
image-encrypted : false
mounting user   : root
mounting mode   : <unknown>
process ID      : 15912
/dev/disk3             /Volumes/ramdisk
```

Booting from a Disk Image (Lion)

With Lion, OS X offers new boot arguments that allow the user to specify the names of DMG files to be used as the root file system. `imageboot_needed()` (in `bsd/kern/imageboot.c`) checks for the presence of the boot arguments, and, if found, calls `imageboot_setup()`. These boot arguments are shown in Table 15-3:

TABLE 15-3: Lion Boot Arguments Used in DMG Processing

BOOT ARGUMENT	CONTAINS
`rp` or `rp0` or `root-dmg`	Name of DMG file to use as root file system. In Lion's install, this is `BaseSystem.dmg`.
`rp1` or `container-dmg`	Name of DMG containing the `root-dmg`. In Lion's installation, this is usually `InstallESD.img`.

The `imageboot_setup()` proceeds to call `imageboot_mount_image()`. The actual loading of the DMG is done by `di_root_image()` (from `iokit/bsddev/DINetBootHook.cpp`), which loads the

IOHDIXController extension by calling di_load_controller. The function returns a BSD device node, the root device, which vfs_mountroot() then mounts as the root file system.

THE VIRTUAL FILE SYSTEM SWITCH

As with most UN*X, OS X uses the virtual file system switch as its layer of abstraction for all file systems. The idea behind VFS is to define a common interface for all file systems, irrespective of their implementations. This interface reduces the file system into fundamental structures: the file system entry, mount entry, and vnode (abstracted inode). Any known file system can then be implemented, while maintaining conformance with this interface. This enables the kernel to present the very same interface to the various POSIX file I/O calls — and, by extension, the user — resulting in a seamless integration of multiple file systems into the same tree.

> *It's interesting to see that, while the VFS is a widely adopted standard across many flavors of UN*X, the implementation can vary greatly. Linux, for example, exposes the* inode*, file, directory entry (*dentry*), and superblock. XNU's VFS is naturally very closely related to BSD's, but is still with some significant differences.*

VFS does not care about the underlying implementation of the file system. It may be table-based (such as FAT) or B-Tree–based (such as NTFS or HFS+). All it requires is that the file system implementation conform to the set interface and allow the mount operation (linking the file system to the UNIX tree) and the retrieval of a file or directory. The file systems may be local or remote, native or foreign — yet the user can access them in the exact same way, which is provided by the familiar UNIX utilities (ls(1), chmod(1), and friends) as well as the POSIX API (open, readdir, etc.). An implementation can always choose to return bogus or default information for features it does not support, a good example being NTFS and UDF — neither of which support the UNIX model of permissions. The file system drivers therefore allow default permissions, which usually allow anyone to read on any file.

The File System Entry

File systems are maintained in the kernel in an array of vfs_fsentry structures. Listing 15-5 defines this structure.

LISTING 15-5: The vfs_fsentry structure, as defined in bsd/sys/mount.h

```
struct vfs_fsentry {
   struct vfsops *vfe_vfsops;     /* vfs operations */
   int            vfe_vopcnt;
   /* # of vnodeopv_desc being registered (reg, spec, fifo...)*/
   vnodeopv_desc **vfe_opvdescs;  /* null terminated;  */
   int            vfe_fstypenum;  /* historic file system type number */
   char           vfe_fsname[MFSNAMELEN]; /* file system type name */
   uint32_t       vfe_flags;      /* defines the FS capabilities */
   void *         vfe_reserv[2];  /* reserved for future use; set this to zero*/
 };
```

File systems are added or removed to the kernel by a call to `vfs_fsadd` or `vfs_fsremove`, respectively, similar to Linux's (un)`register_file system()`. (See Listing 15-6.)

LISTING 15-6: vfs_fsadd and vfs_fsremove, as defined in bsd/sys/mount.h

```
// Add a File system to VFS — provide vfs_fsentry, get vfs_table_t  handle
int vfs_fsadd(_in_ struct vfs_fsentry *, _out_ vfstable_t *);
// Remove a File system from VFS, given the vfstable_t handle
int vfs_fsremove(_in_ vfstable_t);
```

The Mount Entry

The *mount entry* is a `struct mount` (defined in `bsd/sys/mount_internal.h`, and exposed to user mode only as an opaque type), which represents a mounted file system instance. This corresponds, somewhat roughly, to the file system's *superblock*, which is the descriptor holding global file system attributes. The mount entry also holds the file system operations (the `struct vfsops`, discussed later). The structure is shown in Listing 15-7:

LISTING 15-7: A partial detail of the struct mount, from bsd/sys/mount_internal.h

```
struct mount {
        TAILQ_ENTRY(mount) mnt_list;            /* mount list */
        int32_t         mnt_count;              /* reference on the mount */
        lck_mtx_t       mnt_mlock;              // mutex protecting mount point
        struct vfsops   *mnt_op;                /* operations on fs */
        struct vfstable *mnt_vtable;            /* configuration info */
        struct vnode   *mnt_vnodecovered;       /* vnode we mounted on */
        struct vnodelst mnt_vnodelist;          /* list of vnodes this mount */
        struct vnodelst mnt_workerqueue;        /* list of vnodes this mount */
        struct vnodelst mnt_newvnodes;          /* list of vnodes this mount */
        uint32_t        mnt_flag;               /* flags */
        uint32_t        mnt_kern_flag;          /* kernel only flags */
        uint32_t        mnt_compound_ops;       // Available compound ops
        uint32_t        mnt_lflag;              /* mount life cycle flags */
        uint32_t        mnt_maxsymlinklen;      /* max size of short symlink */
        struct vfsstatfs  mnt_vfsstat;          /* cache of file system stats */
        qaddr_t         mnt_data;               /* private data */

        /* Cached values of the IO constraints for the device */
        // ...
        // ...

#if CONFIG_TRIGGERS

        // TRIGGERS is a compile time option which allows the setting of
        // callbacks on mount operations and specific vnodes
```

```
        int32_t         mnt_numtriggers;   /* num of trigger vnodes for this mount */
        vfs_trigger_callback_t *mnt_triggercallback;
        void            *mnt_triggerdata;
#endif
        /* XXX 3762912 hack to support HFS file system 'owner' */
        uid_t           mnt_fsowner;
        gid_t           mnt_fsgroup;

        struct label    *mnt_mntlabel;          /* MAC mount label */
        struct label    *mnt_fslabel;           /* MAC default fs label */

       // Other various cached elements ..

}
```

Note that a file system may be registered (using `vfs_fsadd()` as previously demonstrated), but not necessarily be mounted. Additionally, the same file system type may be mounted multiple times (for example, if several partitions have the same format type).

Key in both the `mount` and `vfs_fsentry` structures are the `vfsops` (in `mount`, `mnt_op`, and in `vfs_fsentry`, `vfe_vfsops`). These are the standard abstracted operations expected of any file system. They are defined (and rather neatly javadoc'ed) in `bsd/sys/mount.h`, and shown in Table 15-4.

TABLE 15-4: The vfs operation callbacks

VFS OPERATION	USED FOR
`int (*vfs_init)(struct vfsconf *);`	Called once, when VFS initializes support for the file system.
`int (*vfs_mount)(struct mount *mp, vnode_t devvp, user_addr_t data, vfs_context_t context);`	Mounts a file system of this type.
`int (*vfs_start)(struct mount *mp, int flags, vfs_context_t context);`	Makes file system active.
`int (*vfs_unmount)(struct mount *mp, int mntflags, vfs_context_t context);`	Called when the user performs and `umount(8)` on the file system.

(Continues)

TABLE 15-4 *(continued)*

VFS OPERATION	USED FOR
`int (*vfs_root) (struct mount *mp, struct vnode **vpp, vfs_context_t context);`	Retrieves a pointer (in *vpp*) to the root of the file system mounted on *mp*.
`int (*vfs_quotactl) (struct mount *mp, int cmds, uid_t uid, caddr_t arg, vfs_context_t context);`	Called when the user calls `quotactl(2)`.
`int (*vfs_getattr) (struct mount *mp, struct vfs_attr *attr, vfs_context_t context);`	Gets attributes of file system mounted at *mp* into *attr*.
`int (*vfs_setattr) (struct mount *mp, struct vfs_attr *attr, vfs_context_t context);`	Sets attribute *attr* for file system mounted at *mp*.
`int (*vfs_sync) (struct mount *mp, int waitfor, vfs_context_t context);`	Syncs file system at *mp*, when `sync(2)` is called. If *waitfor*, return only after sync complete. Otherwise, start sync but return immediately.
`int (*vfs_vget) (struct mount *mp, ino64_t ino, struct vnode **vpp, vfs_context_t context);`	Retrieves a file's vnode (in *vpp*) by the inode number *ino*.
`int (*vfs_fhtovp) (struct mount *mp, int fhlen, unsigned char *fhp, struct vnode **vpp, vfs_context_t context);`	Retrieves the vnode (in *vpp*) corresponding to the file handle *fhp*, of *fhlen* bytes. Inverse of `vfs_vptofh()`.

VFS OPERATION	USED FOR
`int (*vfs_vptofh) (struct vnode *vp, int *fhlen, unsigned char *fhp, vfs_context_t context);`	Copies into *fhp*, which is a buffer of *fhlen* bytes, the file handle bytes, corresponding to the vnode *vp*. Inverse of `vfs_fhtovp()`.
`int (*vfs_sysctl) (int *, u_int, user_addr_t, size_t *, user_addr_t, size_t, vfs_context_t context);`	Implementation of a VFS space `sysctl(2)` request.

The vnode object

The *vnode object* is built on top of the traditional UNIX inode (from the legacy UFS). This is a "virtual inode," containing the information required for retrieving a file or directory from the disk. The `struct vnode` is defined in `bsd/sys/vnode_internal.h`, which — like `struct` mount — is not exposed to user mode. This is shown in Listing 15-8:

LISTING 15-8: The vnode object, from bsd/sys/vnode_internal.h

```
struct vnode {
        lck_mtx_t v_lock;                       /* vnode mutex */
        TAILQ_ENTRY(vnode) v_freelist;          /* vnode freelist */
        TAILQ_ENTRY(vnode) v_mntvnodes;         /* vnodes for mount point */
        LIST_HEAD(, namecache) v_nclinks;       // names (hard links) of vnode
        LIST_HEAD(, namecache) v_ncchildren;    // cache of named children
         ..
        uint32_t v_listflag;                    // flags,(protected by list_lock)
        uint32_t v_flag;                        // flags (unprotected)
        uint16_t v_lflag;                       // and more flags (local flags)
        uint8_t  v_iterblkflags;                /* buf iterator flags */
        uint8_t  v_references;                  // reference of io_count
        int32_t  v_kusecount;                   /* count of in-kernel refs */
        int32_t  v_usecount;                    /* reference count of users */
        int32_t  v_iocount;                     /* iocounters */
        void *   v_owner;                       /* act that owns the vnode */
        uint16_t v_type;                        /* vnode type */
        uint16_t v_tag;                         /* type of underlying data */
        uint32_t v_id;                          /* identity of vnode contents */
```

continues

LISTING 15-8 *(continued)*

```
        union {
                struct mount    *vu_mountedhere;/* ptr to mounted vfs (VDIR) */
                struct socket   *vu_socket;     /* unix ipc (VSOCK) */
                struct specinfo *vu_specinfo;   /* device (VCHR, VBLK) */
                struct fifoinfo *vu_fifoinfo;   /* fifo (VFIFO) */
                struct ubc_info *vu_ubcinfo;    /* valid for (VREG) */
        } v_un;
        struct  buflists v_cleanblkhd;          /* clean blocklist head */
        struct  buflists v_dirtyblkhd;          /* dirty blocklist head */
        struct klist v_knotes;                  // knotes attached to vnode
        /*
         * the following 4 fields are protected
         * by the name_cache_lock held in
         * excluive mode
         */

        kauth_cred_t    v_cred;               /* last authorized credential */
        kauth_action_t  v_authorized_actions; // current authorized actions */
        int             v_cred_timestamp;     //
        int             v_nc_generation;      //

        /*
         * back to the vnode lock for protection
         */
        int32_t         v_numoutput;       /* num of writes in progress */
        int32_t         v_writecount;      /* reference count of writers */
        const char *v_name;                /* name component of the vnode */
        vnode_t v_parent;                  /* pointer to parent vnode */
        struct lockf    *v_lockf;          /* advisory lock list head */
#ifndef __LP64__
        struct unsafe_fsnode *v_unsafefs;  /* pointer to struct used to lock */
#else
        int32_t         v_reserved1;
        int32_t         v_reserved2;
#endif /* __LP64__ */
        int     (**v_op)(void *);          /* vnode operations vector */
        mount_t v_mount;                   /* ptr to vfs we are in */
        void *  v_data;                    /* private data for fs */
#if CONFIG_MACF
        struct label *v_label;             /* MAC security label */
#endif
#if CONFIG_TRIGGERS
        vnode_resolve_t v_resolve; /* trigger vnode resolve info (VDIR only) */
#endif /* CONFIG_TRIGGERS */
};
```

A key element in the vnode structure is the struct `ubc_info`: It can be used to find information on this vnode's objects in the *unified buffer cache*. The unified buffer cache (implemented in `bsd/kern/ubc_subr.c`) is the BSD mechanism for storing cached vnode data, of files fetched from disks and devices (akin to Linux's buffer and page caches). The `ubc_info` links the vnode to a Mach `memory_object_t`, the likes of which were discussed in the previous chapter.

Each file system can define its own internal node representation but should support the basic representation of the vnode, as well as the set of operations defined on a vnode — creating, reading, writing, deleting. The various vnode operations are maintained in the well-documented `bsd/sys/vnode_if.h`, as shown in Listing 15-9.

LISTING 15-9: VNOP_LOOKUP (lookup a vnode in a directory), from bsd/sys/vnode_if.h

```
__BEGIN_DECLS

struct vnop_lookup_args {
        struct vnodeop_desc *a_desc;
        vnode_t a_dvp;
        vnode_t *a_vpp;
        struct componentname *a_cnp;        vfs_context_t a_context;
};

/*!
 @function VNOP_LOOKUP
 @abstract Call down to a file system to look for a directory entry by name.
 @discussion VNOP_LOOKUP is the key pathway through which VFS asks a
   file system to find a file.  The vnode should be returned with an iocount
   to be dropped by the caller.  A VNOP_LOOKUP() calldown can come without
   preceding VNOP_OPEN().
 @param dvp Directory in which to look up file.
 @param vpp Destination for found vnode.
 @param cnp Structure describing filename to find, reason for lookup,
   and various other data.
 @param ctx Context against which to authenticate lookup request.
 @return 0 for success or a file system-specific error.
 */
#ifdef XNU_KERNEL_PRIVATE
extern errno_t VNOP_LOOKUP(vnode_t, vnode_t *, struct componentname *, vfs_context_t);
#endif /* XNU_KERNEL_PRIVATE */
```

The actual I/O operations on the vnodes themselves are defined in a `struct fileops`, as shown in Listing 15-10:

LISTING 15-10: VNode operations

```
// in bsd/vfs/vfs_vnops.
struct  fileops vnops =
          { vn_read, vn_write, vn_ioctl, vn_select, vn_closefile, vn_kqfilt_add, NULL };
```

FUSE — File Systems in USEr Space

One of the main challenges encountered by file system developers is that, traditionally, file systems live in kernel space. This is understandable, as file services are part of the kernel's responsibilities, but it does impose the tight constraints of kernel space, which are exacerbated given the usually complicated logic and data structures needed by file system implementations.

To alleviate this problem, an open source solution porting file system logic into user space has been developed. Known as FUSE (File systems in USEr space), it has been implemented on various UNIX systems and ported into Mac OS X by Amit Singh (who, among other things, has authored the previous reference on OS X internals[1]). Singh's port became known as MacFUSE[2], but was discontinued in 2009 and became incompatible with Lion. A more recent endeavor to pick up where it left off is known as OSXFUSE[3], and has been modified to work with Lion.

The basic idea in FUSE is that the interaction with the kernel is kept to a bare minimum — by means of registering a stub file system, whose callbacks are all bridged back into a user mode process. It is the user mode process that handles all the file system logic and data structures, impacting performance somewhat, but benefitting greatly from nearly boundless virtual memory and the other fringe benefits in user mode, most notably the decoupling from the OS-idiosyncratic kernel interfaces. The user mode process can implement the file system in memory, manage it on disk, or even call a remote server through FTP, SSH, or other protocols. Because all of this can be done using standard POSIX calls, code for FUSE can be relatively straightforward to port in between UNIX systems. FUSE links with a portable runtime library, called `libfuse`.

Table 15-5 shows some of the supported file systems in user mode.

TABLE 15-5: File systems supported by OS X FUSE

FILE SYSTEM	DESCRIPTION
GrabFS	Also known as the WindowFS, this is a read-only file system automatically populated with folders corresponding to all processes that have active Windows. Each folder contains `.tif` files. Each file, if read, provides an updated screenshot of the window it corresponds to. This is an OS X–specific file system, as it uses Cocoa's `CGWindowListCreateImage()` to create the capture images.
LoopbackFS	Allowing the mounting of any local directory as a separate file system under a different mount point.
Procfs	A file system similar to Linux's `/proc`. This is an OS X–specific file system (Linux's own `/proc` is kernel-based).
SpotlightFS	A file system linked to OS X's spotlight, allowing spotlight searches by simply creating a folder in the file system. The folder is populated on-the-fly with results from Spotlight, much like a Smart Folder. This is an OS X–specific file system because it uses Spotlight.
SSHfs	An SSH-based file system allowing the mounting of remote file systems, with all the NFS operations actually being carried over SFTP requests.

The kernel component of FUSE is fairly simple: It registers a VFS (using `vfs_fsadd`) and exports a set of `/dev/fuseXX` character devices. Operations on this file system instance are intercepted by the kernel extension and serialized in a message, which is then dispatched to the user mode file system.

The user mode file systems, on their part, populate a `struct fuse_operations` with their file operation callbacks, and then call `fuse_main()` to do the rest of the work. This is shown in Listing 15-11:

LISTING 15-11: An example fuse_main()

```
int main (int argc, char **argv)
{
    struct fuse_operations  fuseOps;
    // handle any arguments..
    fuseOps.init =    // pointer to initializer
    fuseOps.destroy = // pointer to destructor
    fuseOps.statfs =  // pointer to statfs(2) handler
    fuseOps.open =    // pointer to file open(2) handler
    fuseOps.release = // pointer to file close(2) handler
    fuseOps.opendir =    // pointer to opendir(3) handler
    fuseOps.releasedir = // pointer to closedir(3) handler
    fuseOps.getattr =    // pointer to getattrlist(2) handler
    fuseOps.read =       // pointer to file read(2) handler
    fuseOps.readdir =    // pointer to readdir(3) handler
    fuseOps.readlink =   // pointer to readlink(2) handler
.. // other handlers // ...
    return fuse_main(argc, new_argv, &fuseOps, NULL);
}
```

The `fuse_operations` (defined in LibFUSE's `fuse.h`) contains handlers for all the well-known POSIX file system calls. These are registered and passed to libFUSE's own dispatcher, which receives the callbacks bridged from the kernel and passes them to the file system–specific implementation. A file system may implement only some of the handlers, choosing to leave handlers NULL, in which case libFUSE will simply return an error. Listing 15-12 demonstrates this, with the `do_write` handler. Other handlers are defined in a similar manner.

LISTING 15-12: libFuse's do_write (from fuse's lib/fuse_lowlevel.c)

```
static void do_write(fuse_req_t req, fuse_ino_t nodeid, const void *inarg)
{
 struct fuse_write_in *arg = (struct fuse_write_in *) inarg;
 struct fuse_file_info fi;

 memset(&fi, 0, sizeof(fi));
 fi.fh = arg->fh;
 fi.fh_old = fi.fh;
 fi.writepage = arg->write_flags & 1;

 // If there is a registered write handler, execute it
 if (req->f->op.write)
    req->f->op.write(req, nodeid, PARAM(arg),
                     arg->size, arg->offset, &fi);
 else // no handler - deny system call
 fuse_reply_err(req, ENOSYS);
}

...
... // This is LibFUSE's handler for "low level" operations:
static struct {
void (*func)(fuse_req_t, fuse_ino_t, const void *);
```

continues

LISTING 15-12 *(continued)*

```
const char *name;
} fuse_ll_ops[] = {
[FUSE_LOOKUP] = { do_lookup, "LOOKUP" },
[FUSE_FORGET] = { do_forget, "FORGET" },
[FUSE_GETATTR] = { do_getattr, "GETATTR" },
[FUSE_SETATTR] = { do_setattr, "SETATTR" },
[FUSE_READLINK] = { do_readlink, "READLINK" },
[FUSE_SYMLINK] = { do_symlink, "SYMLINK" },
[FUSE_MKNOD] = { do_mknod, "MKNOD" },
[FUSE_MKDIR] = { do_mkdir, "MKDIR" },
[FUSE_UNLINK] = { do_unlink, "UNLINK" },
[FUSE_RMDIR] = { do_rmdir, "RMDIR" },
[FUSE_RENAME] = { do_rename, "RENAME" },
[FUSE_LINK] = { do_link, "LINK" },
[FUSE_OPEN] = { do_open, "OPEN" },
[FUSE_READ] = { do_read, "READ" },
[FUSE_WRITE] = { do_write, "WRITE" },
[FUSE_STATFS] = { do_statfs, "STATFS" },
[FUSE_RELEASE] = { do_release, "RELEASE" },
... // many other operations
}
```

Once the user mode file system has handled the request, the reply is serialized again into a message, which returns to the kernel — and is returned to the requester, which remains blissfully unaware of the whole bridging process.

FILE I/O FROM PROCESSES

So far, this book has covered the BSD layer's implementation of processes (in the previous chapter), and vnodes (in this one). But one important aspect has yet to be discussed — how user mode processes access files and perform operations on them.

Recall from Chapter 13 that the BSD `proc_t` structure contains, among its many fields, a `struct filedesc *p_fd`; this is the structure holding all the process's open files in the fields shown in Listing 15-13.

LISTING 15-13: The filedesc structure, from bsd/sys/filedesc.h

```
struct filedesc {
        struct  fileproc **fd_ofiles;   /* file structures for open files */
        char    *fd_ofileflags;         /* per-process open file flags */
        struct  vnode *fd_cdir;         /* current directory */
        struct  vnode *fd_rdir;         /* root directory */
        int     fd_nfiles;              /* number of open files allocated */
        int     fd_lastfile;            /* high-water mark of fd_ofiles */
        int     fd_freefile;            /* approx. next free file */
        u_short   fd_cmask;             /* mask for file creation */
        uint32_t  fd_refcnt;            /* reference count */
```

```
        int     fd_knlistsize;          /* size of knlist */
        struct  klist *fd_knlist;       /* list of attached knotes */
        u_long  fd_knhashmask;          /* size of knhash */
        struct  klist *fd_knhash;       /* hash table for attached knotes */
        int     fd_flags;
};
```

The key fields in this structure are `fd_ofiles` and `fd_ofileflags`. Both are arrays, and the familiar integer file descriptors from user mode (`0` — `stdin`; `1` — `stdout`, `2` — `stderr`) are indices into those arrays. The first array holds the file "object" corresponding to the descriptor, whereas the second one is used for the open flags (i.e. the flags specified by the process in the `open(2)` system call). `fp_lookup` can be used to find the `fileproc` corresponding to a given file descriptor. (See Listing 15-14).

LISTING 15-14: fp_lookup (from bsd/kern/kern_descrip.c)

```
/*
 * fp_lookup
 *
 * Description: Get fileproc pointer for a given fd from the per process
 *              open file table of the specified process and if successful,
 *              increment the f_iocount
 *
 * Parameters:  p                               Process in which fd lives
 *              fd                              fd to get information for
 *              resultfp                        Pointer to result fileproc
 *                                              pointer area, or 0 if none
 *              locked                          !0 if the caller holds the
 *                                              proc_fdlock, 0 otherwise
 *
 * Returns:     0                       Success
 *              EBADF                   Bad file descriptor
 *
 * Implicit returns:
 *              *resultfp (modified)            Fileproc pointer
 *
 * Locks:       If the argument 'locked' is non-zero, then the caller is
 *              expected to have taken and held the proc_fdlock; if it is
 *              zero, than this routine internally takes and drops this lock.
 */
int fp_lookup(proc_t p, int fd, struct fileproc **resultfp, int locked)
{
        struct filedesc *fdp = p->p_fd;
        struct fileproc *fp;

        if (!locked)  // take lock to prevent race conditions
                proc_fdlock_spin(p);

       // A negative file descriptor, one that is larger than the count of open files,
       // one that has no fileproc * entry, or one that is reserved—all return EBADF

        if (fd < 0 || fdp == NULL || fd >= fdp->fd_nfiles ||
                        (fp = fdp->fd_ofiles[fd]) == NULL ||
                        (fdp->fd_ofileflags[fd] & UF_RESERVED)) {
```

continues

LISTING 15-14 *(continued)*

```
                if (!locked) // failure. Drop lock first
                        proc_fdunlock(p);
              // and return error..

                return (EBADF);
        }
        fp->f_iocount++;

        // If we found an entry, fp points to it. This is also what we return to caller.
        if (resultfp)
                *resultfp = fp;

        // can safely let go of the lock
       if (!locked)
                proc_fdunlock(p);

        return (0); // success
}
```

The `fileproc` structures in `fd_ofiles` are surprisingly small structures:

```
struct fileproc {
        unsigned int f_flags;
        int32_t f_iocount;
        struct fileglob * f_fglob;
        void *  f_waddr;
};
```

The reason for this is that all the file data is held *globally* in the kernel and is merely pointed to by the `f_fglob` field. This means that if the same file is opened by two processes, each may refer to it by means of a different file descriptor (and, hence, a different `fileproc`, private to each process), but the underlying file data, which is pointed to by the `f_fglob` pointers, resides at the same address in kernel memory. This is shown in Listing 15-15:

LISTING 15-15: the fileglob pointer, from bsd/sys/file_internal

```
/* file types */  // these are the types allowable for fg_type
typedef enum {
        DTYPE_VNODE     = 1,    /* file */
        DTYPE_SOCKET,           /* communications endpoint */
        DTYPE_PSXSHM,           /* POSIX Shared memory */
        DTYPE_PSXSEM,           /* POSIX Semaphores */
        DTYPE_KQUEUE,           /* kqueue */
        DTYPE_PIPE,             /* pipe */
        DTYPE_FSEVENTS          /* fsevents */
} file_type_t;

struct fileglob {
        LIST_ENTRY(fileglob) f_list;/* list of active files */
        LIST_ENTRY(fileglob) f_msglist;/* list of active files */
        int32_t fg_flag;                /* see fcntl.h */
        file_type_t fg_type;    /* descriptor type */
```

```
        int32_t fg_count;       /* reference count */
        int32_t fg_msgcount;    /* references from message queue */
        kauth_cred_t fg_cred;   /* credentials associated with descriptor */
        struct  fileops {  // generic file operations
                int     (*fo_read)      (struct fileproc *fp, struct uio *uio,
                                         int flags, vfs_context_t ctx);
                int     (*fo_write)     (struct fileproc *fp, struct uio *uio,
                                         int flags, vfs_context_t ctx);
#define FOF_OFFSET      0x00000001      /* offset supplied to vn_write */
#define FOF_PCRED       0x00000002      /* cred from proc, not current thread */
                int     (*fo_ioctl)     (struct fileproc *fp, u_long com,
                                         caddr_t data, vfs_context_t ctx);
                int     (*fo_select)    (struct fileproc *fp, int which,
                                         void *wql, vfs_context_t ctx);
                int     (*fo_close)     (struct fileglob *fg, vfs_context_t ctx);
                int     (*fo_kqfilter)  (struct fileproc *fp, struct knote *kn,
                                         vfs_context_t ctx);
                int     (*fo_drain)     (struct fileproc *fp, vfs_context_t ctx);
        } *fg_ops;
        off_t   fg_offset;
        void    *fg_data;               /* vnode or socket or SHM or semaphore */
        lck_mtx_t fg_lock;
        int32_t fg_lflags;              /* file global flags */
#if CONFIG_MACF
        struct label *fg_label;  /* JMM - use the one in the cred? */
#endif
};
```

The `fg_data` field in the `fileglob` structure is a pointer to an object, whose contents depend on `fg_type`. File handling system calls usually switch on the `fg_data` field. A good example can be seen in the implementation of `fstat1()` in Listing 15-16, which is the common implementation of the `fstat()` family of system calls.

LISTING 15-16: fstat1(), the implementation of fstat, from bsd/kern/kern_descrip.c

```
#define f_type f_fglob->fg_type
#define f_data f_fglob->fg_data
..

static int
fstat1(proc_t p, int fd, user_addr_t ub, user_addr_t xsecurity,
       user_addr_t xsecurity_size, int isstat64)
{
        struct fileproc *fp;
...
       // use fp_lookup to first get the fileproc
if ((error = fp_lookup(p, fd, &fp, 0)) != 0) {
                return(error);
        }
        type = fp->f_type; // remember this is really fp->f_glob->f_type;
        data = fp->f_data; // .. and ditto for fp->f_glob->f_data;
..
switch (type) {
  case DTYPE_VNODE: // data cast to a vnode_t
```

continues

LISTING 15-16 *(continued)*

```
                if ((error = vnode_getwithref((vnode_t)data)) == 0) {
                   /*
                    * If the caller has the file open, and is not
                    * requesting extended security information, we are
                    * going to let them get the basic stat information.
                    */
                   if (xsecurity == USER_ADDR_NULL) {
                      error = vn_stat_noauth((vnode_t)data, sbptr, NULL, isstat64, ctx);
                   } else {
                     error = vn_stat((vnode_t)data, sbptr, &fsec, isstat64, ctx);
                  }

                        AUDIT_ARG(vnpath, (struct vnode *)data, ARG_VNODE1);
                        (void)vnode_put((vnode_t)data);
                }
                break;

#if SOCKETS
  case DTYPE_SOCKET:  // data cast to a struct socket *
                error = soo_stat((struct socket *)data, sbptr, isstat64);
                break;
#endif /* SOCKETS */
  case DTYPE_PIPE: // data will be cast into a struct pipe (inside pipe_stat)
                error = pipe_stat((void *)data, sbptr, isstat64);
                break;

  case DTYPE_PSXSHM: // data will be case into a struct pshmnode (inside pshm_stat)
                error = pshm_stat((void *)data, sbptr, isstat64);
                break;

  case DTYPE_KQUEUE: // data actually ignored for a kqueue
                funnel_state = thread_funnel_set(kernel_flock, TRUE);
                error = kqueue_stat(fp, sbptr, isstat64, p);
                thread_funnel_set(kernel_flock, funnel_state);
                break;
..
```

Reading and writing becomes a simple matter of passing the arguments around to the underlying file reading/writing implementation. For example, consider `fo_read` in Listing 15-17 (other functions implemented similarly):

LISTING 15-17: fo_read from bsd/kern/kern_descript.c

```
int fo_read(struct fileproc *fp, struct uio *uio, int flags, vfs_context_t ctx)
{
      // simple pass through. Remember that by f_ops we mean f_fglob->f_ops
       return ((*fp->f_ops->fo_read)(fp, uio, flags, ctx));
}
```

The `f_ops` field on the `fileglob` structure is set to the default set of file operations. Again, this changes with the file type: `vnops` for vnodes, `pipeops` for pipes, and so on. In this way, the generic operations can be adapted to any file type.

SUMMARY

This chapter explored XNU's handling and implementation of file systems. Not unlike its BSD origins, XNU uses the virtual filesystem switch to allow any file system to plug in to the kernel, given the right interface. FUSE, which has been ported to OS X, further allows the extension of VFS for file systems that are implemented in user mode.

The chapter concluded by linking the VFS implementation to the process notion of a file descriptor. This will come in handy in Chapter 17, which is dedicated to the implementation of the socket APIs. The next chapter, however, turns first to a specific file system implementation — Apple's native HFS+.

REFERENCES AND FURTHER READING

1. Singh, Amit, "Mac OS X Internals, A Systems Approach." (Addison-Wesley; 2006)
2. MacFUSE project page on Google Code: `http://code.google.com/p/macfuse/`
3. OSXFUSE project page on `github:http://osxfuse.github.com/`
4. Apple Technical Note 2166 – "Secrets of the GPT" — `http:// developer.apple.com/technotes/tn2006/tn2166.htm`

16

To B (-Tree) or Not to Be — The HFS+ File Systems

Although today's operating systems can support — with the help of drivers — any type of file system, each operating system has a "native" file system. In DOS, it was FAT. Windows has NTFS. Linux has Ext2/3/4. And OS X, being no exception, has HFS+. This chapter dives deep into the internals of HFS+, and its variant — HFSX — used in iOS. The file system internal structure is described, with actual examples and hands-on exercises you can follow.

> *A companion tool for this book, hfsleuth, is available for free download from the book's website. Since this chapter deals with low-level and on-disk structures, hfsleuth provides a great way to follow along and look at low-level disk structures. It does, however, often require read access to the raw disk device, which you can either supply directly (via* `chmod(1)` *on* `/dev/rdisk##`*), or simply run the tool as root. The tool also has a writeable mode, but it is disabled by default for safety.*

HFS+ FILE SYSTEM CONCEPTS

Following the discussion of generic file system concepts in the previous chapter, this section presents these concepts as they pertain to HFS+, as well as a few novel concepts which exist only in Apple's favorite file system.

Timestamps

HFS+ maintains its dates as a count of seconds from January 1, 1904, GMT, as an unsigned integer. This choice of start time is rather peculiar, as computers as we know them didn't exist back then. Even UNIX dates are relative to the "epoch" (January 1, 1970). As a result, despite using a UInt32, the last possible date is February 6, 2040, 06:28:15 GMT. Conversion between the two is easy enough, however, as one need only subtract (365.25 × 66 × 86,400) from the HFS+ date to get to a UNIX date.

Access Control Lists

As noted in the previous chapter, traditional UNIX offers permissions at the `inode` level. These permissions, however, are very limited, conforming to the simple model of User/Group/Other. ACLs enable the meticulous setting of permissions for any number of users and groups on the system, in a manner similar to Windows permissions.

It's important to note that ACLs are actually a VFS feature (or, to be more pedantic, KAUTH), and not an HFS+ one. However, for ACLs to work, the underlying file system must support Extended Attributes (which HFS+ does), as discussed next.

Extended Attributes

Files have, besides the actual blocks containing their data and their permissions, additional attributes. These are commonly referred to as *extended attributes*, and OS X makes extensive use of them, both in user mode applications (Spotlight and Finder, to name two), and in the kernel.

OS X added extended attributes in 10.4, and the previously mentioned ACLs are actually implemented as extended attributes, as in per-file compression, which was introduced in 10.6, and described below. OS X provides the `xattr(1)` command, which enables the listing of extended attributes, as well as a `-@` switch to its `ls(1)`.

➤ Extended attributes are generally opaque; they can be set by anyone, and OS X follows a reverse DNS convention, to ensure attribute uniqueness. The exact meaning of the attribute is left up to the setter to decide. Toggling folder color labels and running `xattr(1)`, for example, quickly reveals that indicated byte value corresponds to the folder color. Another interesting attribute is `com.apple.quarantine`, which is responsible for the familiar "%s is an application downloaded from the internet." This attribute is also used by the SandBox kext to detect which Applications are potentially dangerous.

Table 16-1 lists some of the common extended attributes and their format:

TABLE 16-1: System defined extended attributes

EXTENDED ATTRIBUTE (COM.APPLE)	FORMAT	USAGE
`decmpfs`	`Decmpfs` header	Compressed file indicator or, for small files, data
`FinderInfo`	Undocumented	Finder information, e.g. folder colors

EXTENDED ATTRIBUTE (COM.APPLE)	FORMAT	USAGE
`metadata`	As per the Spotlight Metadata attribute format[1]	Spotlight Metadata. Used by Safari, for example, to catalog where a download originated (using `kMDItemWhereFroms`)
`quarantine`	0000; 32-bit Timestamp; AppName; GUID|appID	Quarantine for files of dubious origin (i.e., only the Internet)
`cprotect`	`struct cp_xattr (bsd/sys/cprotect.h)`	Used by iOS 4 and later for file content protection: Provides encrypted key of file
`system.Security`	`struct kauth_acl (bsd/sys/kauth.h)`	Used by VFS for extended ACLs

Extended attributes form the basis for many features, such as Access Control Lists (described previously), forks, and transparent compression (both described later). Theoretically, any file system that supports extended attributes could support the features built on top of them, as in XNU support for extended attributes is implemented at the VFS level, as callouts to the specific file system logic.

The `xattr(1)` command is, surprisingly enough, a Python script(!) and not a binary. Why Apple left it as Python is puzzling, considering that its functionality is provided directly by system calls, and even more so when due to Python version hell there are no less than four `xattrs`: The main file, which selects one of the actual scripts by Python version. This is true even in Mountain Lion:

```
morpheus@Simulacrum (~)$ ls -l /usr/bin/xatt*
-rwxr-xr-x  2 root  wheel   925 Mar 23 00:58 /usr/bin/xattr
-rwxr-xr-x  1 root  wheel  7786 Mar 23 00:58 /usr/bin/xattr-2.5
-rwxr-xr-x  1 root  wheel  9442 Mar 23 00:58 /usr/bin/xattr-2.6
-rwxr-xr-x  1 root  wheel  9442 Mar 23 00:58 /usr/bin/xattr-2.7
morpheus@Simulacrum (~)$ file /usr/bin/xattr
/usr/bin/xattr: a /usr/bin/python script text executable
```

To add insult to injury, `xattr(1)` filters out some important extended attributes, those dealing with file compression. This is shown in the following experiment.

Experiment: Viewing Extended Attributes

Implementing an actually usable version of `xattr(1)` is as easy as using the `listxattr(2)` system call directly, as is shown in the Listing 16-1:

LISTING 16-1: Simple, but working code to list extended attributes

```
#include <sys/xattr.h>
#include <stdlib.h>
#include <stdio.h>
#define BUFSIZE         4096

// Minimal version of xattr, but one that actually presents compressed attributes
// Can be extended to support reading and writing the attribute themselves
// (left as an exercise for the reader)

int  main (int argc, char **argv)
{
   char *fileName = argv[1];
   int xattrsLen;
   char *xattrNames;
   char *attr;

   // We could call listxattr with NULL to get the name len, but - quick & dirty
   // I have yet to see a file with more than 4K of extended attribute names..

   xattrNames = malloc (BUFSIZE);
   memset (xattrNames, '\0', BUFSIZE); // or calloc..

   switch (listxattr (fileName,
                      xattrNames,
                      BUFSIZE,
                      XATTR_SHOWCOMPRESSION | XATTR_NOFOLLOW))
    {
       case 0:
          fprintf(stderr, "File %s has no extended attributes\n", fileName); return (0);
       case -1:
          perror("listxattr"); return (1);
       default: // it worked. fall through
        ;
   }
   // rely on attributes being NULL terminated..
   for (attr = xattrNames; attr[0]; attr += strlen(attr) + 1)
       {
              printf ("Attribute: %s\n", attr);
       }

   free(xattrNames); // Be nice. Clean up
   return (0);
}
```

The listing should compile nearly. After compiling it (or downloading the tool from the book's companion website), you can use it on any file in the system, and view, for example, compression-related extended attributes (as shown in another experiment, in a few pages).

If you complete the exercise, so as to list the extended attribute values, you can try an extra step of this experiment: Start Finder in the some directory, and assign a color label to a file. Use `xattr` from the listing to look at the `com.apple.FinderInfo` attribute. You should see something like Output 16-1:

OUTPUT 16-1: The com.apple.FinderInfo attribute changing along with color labels

```
morpheus@Ergo (/)$ jxattr  -p ~/Desktop/test
Attribute: com.apple.FinderInfo (32 bytes)
\x0\x0\x0\x0\x0\x0\x0\x0\x0\xc\x0\x0\x0...  # Red

Attribute: com.apple.FinderInfo (32 bytes)
\x0\x0\x0\x0\x0\x0\x0\x0\x0\xe\x0\x0\x0...  # Orange

Attribute: com.apple.FinderInfo (32 bytes)
\x0\x0\x0\x0\x0\x0\x0\x0\x0\x2\x0\x0\x0…    # Gray
```

You can view *almost* all the extended attributes a file has using the system calls. If you use the code from the listing to look for some of the system properties, like content protect or ACLs, you will come up empty handed. This, however, is not a shortcoming of the code, so much as the filtering imposed at the system call level. These attributes are, in fact, there, but you need to read them directly from the file system and this is exactly what low-level tool like hfsleuth can do, as shown later.

Forks

Forks are a concept first devised by Apple (in the original HFS), and later adopted by Microsoft in NTFS (wherein it is referred to as *alternate data streams*). A fork is much like an extended attribute, in that it can be used for additional metadata, but is more suited for data that can be put in a separate, albeit related file. Whereas extended attributes have size limitations, forks do not.

While OS X can support virtually any number of forks, most files have exactly one fork — the *data fork* — which is the where the file's actual data is stored. Some files may also maintain a *resource fork*, though that, too is rare. To see a resource fork, simply append `/..namedfork/rsrc` to any file name. One such file is `/Developer/Icon^M` (the `^M` being Ctrl+M, which you can type by pressing Ctrl+V Ctrl+M — otherwise Ctrl+M doubles as the Enter key), or by hitting Tab to auto-complete. This is demonstrated in Output 16-2:

OUTPUT 16-2: Demonstrating resource forks

```
morpheus@Ergo (~)$ ls -l@ /Developer/Icon^M
-rw-r--r--@ 1 root  admin  0 Nov 14  2011 /Developer/Icon?
        com.apple.FinderInfo 32
        com.apple.ResourceFork 338          ls -l shows the finder extended
                                            attribute, and a 338 byte resource fork
morpheus@Ergo (~)$ xattr -l /Developer/Icon^M
com.apple.FinderInfo:
00000000  00 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  |........@.......|
00000010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00000020                           xattr(1) (or jxattr) can be used to dump the
com.apple.ResourceFork:            extended attributes, including the resource fork
00000000  00 00 01 00 00 00 01 20 00 00 00 20 00 00 00 32  |....... ... ...2|
...
00000110  00 00 00 00 00 00 00 00 64 65 76 66 6D 61 63 73  |........devfmacs|
00000120  00 00 01 00 00 00 01 20 00 00 00 20 00 00 00 32  |....... ... ...2|
00000130  00 00 00 00 09 00 00 00 00 1C 00 32 00 00 62 61  |..........2..ba|
```

continues

OUTPUT 16-2 *(continued)*

```
00000140  64 67 00 00 00 0A BF B9 FF FF 00 00 00 00 01 00  |dg..............|
00000150  00 00                                            |..|
00000152

morpheus@Ergo (~)$ ls  -l /Developer/Icon^M/..namedfork/rsrc
-rw-r--r--  1 root  admin  338 Nov 14  2011 /Developer/Icon?/..namedfork/rsrc

morpheus@Ergo (~)$ od -A x -t x1 /Developer/Icon^M/..namedfork/rsrc
0000000    00  00  01  00  00  00  01  20  00  00  00  20  00  00  00  32
0000010    00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
*
0000100    00  00  00  1c  00  00  00  00  00  00  00  00  00  00  00  00
0000110    00  00  00  00  00  00  00  00  64  65  76  66  6d  61  63  73
0000120    00  00  01  00  00  00  01  20  00  00  00  20  00  00  00  32
0000130    00  00  00  00  09  00  00  00  00  1c  00  32  00  00  62  61
0000140    64  67  00  00  00  0a  bf  b9  ff  ff  00  00  00  00  01  00
0000150    00  00      ..and the fork may be accessed as a normal file,
0000152                by appending ..namedfork/rsrc
```

One place where resource forks are used extensively is in OS X aliases. Aliases make good use of their resource forks. When created, and even if it renamed, an Alias has an extended Finder attribute (`com.apple.FinderInfo`) specifying `alisMACS`, and a resource fork specifying the coordinates of the original file, as well as the icons. Surprisingly enough, in many cases the aliases take up more disk space than the files they are aliases of.

Compression

➤ File compression is one of HFS+'s strongest features, and also the one most easily overlooked. This is because, as of 10.6, it is provided transparently. Compression is implemented by leaving the data fork empty, and placing the compressed data in the resource fork. An additional extended attribute, `com.apple.decmpfs`, marks the file as compressed. OS X utilities, however, silently perform decompression on the fly of system files, and even the extended attribute utility, `xattr(1)`, ignores the extended attribute of `com.apple.decmpfs`, which is used for compression. The kernel supports on-the-fly compression using the specialized `AppleFSCompressionTypeZlib.kext`.

If you are using Lion or later, `ls(1)` has been adapted to detect and display compressed files if the `-O` switch is used on a compressed file. Doing so will not display compression details. However, one of the few ways to see compression in action is using `du`. This is shown in Output 16-3:

OUTPUT 16-3: Demonstrating the actual size of a file using du

```
                                            Note: No extended attributes for ls
morpheus@Minion (~)$ ls -lO@ /bin/ls
-r-xr-xr-x  1 root  wheel  compressed 80752 Feb  6 10:49 /bin/ls
morpheus@Minion (~)$ du -h !$
du -h /bin/ls                                  Yet size used is significantly
32K   /bin/ls                                  smaller than
```

The ditto(1) utility supports compression with a --hfsCompression switch. The compression is implemented by a private framework, Bom, which — in turn — compresses using the private framework AppleFSCompression, libz (gzip style Lempel-Ziv 77 compression), and libbz2 (Bunzip2, or Burroughs-Wheeler). (You can see this for yourself by using otool -l on these files).

The hfsleuth companion tool can be used to display compression details when used on a normal file, as shown in Output 16-4.

OUTPUT 16-4: Using hfsleuth on a compressed file

```
morpheus@Minion (~)$ ls -lO@ /bin/ls
-r-xr-xr-x  1 root  wheel  compressed 80752 Feb  6 10:49 /bin/ls
morpheus@Minion (~)$ hfsleuth -v  /bin/ls
/bin/ls: File size is 80752 bytes, compressed (actual size is 31047 bytes)
No extended attributes (aside from compression)
```

A little known fact is that when Apple integrated compression into HFS+, they did so in a highly modular way, with most of the logic actually decoupled from HFS+. This means that compression support could very well be implemented by other file systems, so long as they support extended attributes.

Detecting File Compression

The kernel can detect if a given file (more accurately, a vnode) is compressed by calling decmpfs_file_is_compressed (bsd/kern/decmpfs.c). This function checks the value of the com.apple.decmpfs extended attribute. Client file systems (in our case, HFS+), can wrap this with their own logic, as HFS+ does with hfs_file_is_compressed (bsd/hfs/hfs_vnops.c). This function first checks a cached value stored in a decmpfs_cnode or compression node, which decmpfs maintains for compressed data. If this is a first time the file is opened, no cached value exists, and so a call is made to the generic function, which also sets up the cnode.

File Decompression

As noted earlier, HFS+ compression in the kernel is implemented in a highly modular fashion. Rather than commit to a particular type of algorithm, the HFS+ code in the kernel's bsd/hfs directory calls out to decompression logic in bsd/kern/decmpfs.c. To further enable modularity, the decompression is performed by one of potentially several (up to CMP_MAX) decompressors, which can be registered externally (i.e., from kexts), using the register_decmps_decompressor function. This is shown in Listing 16-2:

LISTING 16-2: Decompression logic exported in bsd/sys/decmpfs.h

```
#define DECMPFS_REGISTRATION_VERSION 1
typedef struct {
    int  decmpfs_registration; // "1"
    decmpfs_validate_compressed_file_func validate;
    decmpfs_adjust_fetch_region_func      adjust_fetch;
    decmpfs_fetch_uncompressed_data_func  fetch;
```

continues

LISTING 16-2 *(continued)*

```
    decmpfs_free_compressed_data_func     free_data;
} decmpfs_registration;

/* hooks for kexts to call */
errno_t register_decmpfs_decompressor
 (uint32_t compression_type,
  decmpfs_registration *registration);
errno_t unregister_decmpfs_decompressor
  (uint32_t compression_type,
   decmpfs_registration *registration);
```

The `decmpfs` mechanism registers the Type1 compressor, which is used in cases where the data is already too small to be effectively compressed and can fit in the extended attribute itself, in plaintext. Other registrations can be performed by external kexts. The `AppleFSCompressionTypeZlib.kext` registers Type3 and Type4 compressors, and the `AppleFSCompressionTypeDataless.kext` (in OS X, as of Lion) registers Type5.

If a kernel extension has not yet registered the appropriate decompressor, the process works in reverse: `decmpfs` uses I/O Kit to query the driver catalogue for the driver which purports to support the required type. Calls to the actual decompressor functions use `_decmp_get_func`, shown in Listing 16-3.

LISTING 16-3: _decmp_get_func, used to obtain decompressor functions

```
_decmp_get_func(uint32_t type, int offset)
{
  /*
   this function should be called while holding a shared lock to decompressorsLock,
   and will return with the lock held
   */

  if (type >= CMP_MAX) // only up to CMP_MAX decompressors
           return NULL;

  if (decompressors[type] != NULL) {
     // already have a registered decompressor at this offset, return its function
     return _func_from_offset(type, offset);
   }

  // does IOKit know about a kext that is supposed to provide this type?
  char providesName[80];
  snprintf(providesName, sizeof(providesName),
        "com.apple.AppleFSCompression.providesType%u", type);

  // I/O Kit and its "Catalogue" are both discussed in detail in Chapter 19
  if (IOCatalogueMatchingDriversPresent(providesName)) {
        // there is a kext that says it will register for this type, so let's wait for
it
        char resourceName[80];
        uint64_t delay = 10000000ULL; // 10 milliseconds.
```

```
            snprintf(resourceName, sizeof(resourceName),
                     "com.apple.AppleFSCompression.Type%u", type);
            printf("waiting for %s\n", resourceName);
            while(decompressors[type] == NULL) {
                lck_rw_done(decompressorsLock);

                if (IOServiceWaitForMatchingResource(resourceName, delay)) {
                    break;
                }
                if (!IOCatalogueMatchingDriversPresent(providesName)) {

                    printf("the kext with %s is no longer present\n", providesName);
                    break;
                }
                printf("still waiting for %s\n", resourceName);
                delay *= 2;
                lck_rw_lock_shared(decompressorsLock);
            }
            // IOKit says the kext is loaded, so it should be registered too!
            if (decompressors[type] == NULL) {
                ErrorLog("we found %s, but the type still isn't registered\n",
    providesName);
                return NULL;
            }
            // it's now registered, so let's return the function
            return _func_from_offset(type, offset);
        }

            // the compressor hasn't registered, so it never will unless someone
            // manually kextloads it
            ErrorLog("tried to access a compressed file of unregistered type %d\n", type);
            return NULL;
    }
```

I/O Kit is described in more detail in Chapter 19, but the code should still be clear: `decmp_get_func` first checks if it has a registered decompressor (in which case it can just return its function). If it does not, it calls on I/O Kit to look up a driver and load it and waits (with exponentially increasing delays) until that driver is registered. The driver is expected to have registered itself by then at the appropriate offset, and its function can be returned.

Note, that with all this talk about decompression, we have not mentioned compression. This is because the kernel cannot perform the compression, and has no support for external compressors, either: Only the decompression is supported at the kernel level. Apple provides pre-compressed files during the installation process. For compression any time thereafter, you need to use the `ditto(1)` command, with its `--hfsCompression` switch. As stated, the command (part of the `BomCmds` package) is closed source, but the HFS+ compression process can generally be described as follows:

- The file is treated as an array of 64 K blocks.
- Small files are compressed with Type1, with their data stored in the extended attribute, uncompressed.
- Larger files that can still fit inside the `com.apple.decmpfs` extended attribute in one block are stored in the extended attributes.

- All other larger files are compressed using the file's resource fork. Note that, in this case, the file may not have its own resource fork.
- The extended attribute and the resource fork are added to the file.
- The actual file size is recoded as 0, and `chflags(2)` marks the file as compressed.

The following experiment demonstrates how file system compression is implemented.

Experiment: Viewing File Compression

Using the program created in Listing 16-1, you can easily see compression-related extended attributes, even though the normal `xattr` will not. To try this out, create a small file, and then copy it to your directory using `ditto(1)`, applying compression in the process. This will look something like Output 16-5:

OUTPUT 16-5: Compressing a file with ditto(1)

```
morpheus@minion (~)$ echo "This is a test of compression" > file
morpheus@minion (~)$ ditto -hfsCompression file fileComp
morpheus@minion (~)$ ls -lO file*
-rw-r--r--  1 morpheus  staff  -          30 Apr 29 16:39 file
-rw-r--r--  1 morpheus  staff  compressed 30 Apr 29 16:39 fileComp
```

Now use the `xattr` from Listing 16-1 on the file. You should be able to see your file has the `com.apple.decmpfs` attribute, but not the resource fork, since its compressed data is small enough. Trying this again on a larger file (usually over 20 K) will create the resource fork. This is shown in Output 16-6:

OUTPUT 16-6: Who's the real xattr?

```
morpheus@Minion (~)$ /usr/bin/xattr -p com.apple.decmpfs fileComp
xattr: fileComp: No such xattr: com.apple.decmpfs  # Liar!

morpheus@Minion (~)$ xattr /bin/ls                 # no attrs on /bin/ls, either

morpheus@Minion (~)$ ls -l /bin/ls                 # It's a conspiracy!
-r-xr-xr-x  1 root  wheel  80752 Feb  6 10:49 /bin/ls

# by comparison, running our version, from Listing 16-xat
#
morpheus@Minion (~)$ ./xattr fileComp
Attribute: com.apple.decmpfs                       # our version tells the truth
morpheus@Minion (~)$ ./xattr  /bin/ls              # And /bin/ls has a resource fork
Attribute: com.apple.ResourceFork
Attribute: com.apple.decmpfs
```

Completing the exercise and also printing the extended attribute values, will reveal that, interestingly enough, even though the file is technically compressed (with its data in the extended attribute), it is not actually. This is because, for very small files, the overhead of compression headers might actually be larger than the file data that is being compressed. The same does not hold for `/bin/ls`, which has been compressed from 80,752 bytes to a mere 31,047 — a significant savings of about 62%!

```
# Printing out the extended attribute (left as an exercise)
# Note our file is not really compressed, but its content is in the attribute
#
morpheus@Minion (~)$ ./xattr -v fileComp
Attribute: com.apple.decmpfs (47 bytes)
fpmc\x3\x0\x0\x0\x1e\x0\x0\x0\x0\x0\x0\x0\xffThis is a test of compression\xa

# In /bin/ls, the resource fork holds the data, and the extended attribute
# only holds the fpmc ('cmpf', in reverse) header.
morpheus@Minion (~)$ ./xattr -v /bin/ls
Attribute: com.apple.decmpfs (16 bytes)
fpmc\x4\x0\x0\x0p;\x1\x0\x0\x0\x0\x0
Attribute: com.apple.ResourceFork (31047 bytes)
\x0\x0\x1\x0\x0\x0y\x15\x0\x0x\x15\x0...
   //output truncated for brevity, but note file is significantly smaller
```

Now perform any subtle modification you wish on the file. For example, add a character. You will see the file has lost its compression. (See Output 16-7.)

OUTPUT 16-7: Compression is lost on file modification

```
morpheus@Minion (~)$ echo "." >> fileComp
morpheus@Minion (~)$ ls -lO file*
-rw-r--r--  1 morpheus  staff  - 30 Apr 29 16:39 file
-rw-r--r--  1 morpheus  staff  - 32 Apr 29 16:44 fileComp
morpheus@Minion (~)$ ./xattr fileComp
File fileComp has no extended attributes
```

Unicode Support

Gone are the days of 8-bit ASCII. Nowadays, as users download more content from the Internet, there is a need for Internationalization — `I18n` — at the file system level. This means that file names in different languages and character sets must be supported by the file system.

HFS+ solves internationalization problems by simply using Unicode. Of the many Unicode variants, the one used in UTF-16 — double byte Unicode, and filenames can be up to 255 characters (i.e., 510 bytes) in length. The data structure used internally by HFS+ is an `HFSUniStr255`, defined here:

```
struct HFSUniStr255 {
    UInt16  length;
    UniChar unicode[255];
};
typedef struct HFSUniStr255 HFSUniStr255;
```

The Unicode is in big-endian order, meaning that on Intel architecture every byte has to be swapped (using `be16_to_cpu` or some other macro).

Finder integration

HFS+ is tightly integrated with the OS X Finder (discussed in Chapter 7). Both the volume header, as well as the individual catalog entries have a special Finder Information field, which contains flags for use by Finder. The exact information depends on whether it is for a file or a folder. This is shown in Listing 16-4.

LISTING 16-4: Finder Information, from bsd/hfs/hfs_format.h

```
/* Finder information */
struct FndrFileInfo {
        u_int32_t       fdType;         /* file type */
        u_int32_t       fdCreator;      /* file creator */
        u_int16_t       fdFlags;        /* Finder flags */
        struct {
            int16_t     v;              /* file's location */
            int16_t     h;
        } fdLocation;
        int16_t         opaque;
} __attribute__((aligned(2), packed));
typedef struct FndrFileInfo FndrFileInfo;

struct FndrDirInfo {
        struct {                        /* folder's window rectangle */
            int16_t     top;
            int16_t     left;
            int16_t     bottom;
            int16_t     right;
        } frRect;
        unsigned short  frFlags;        /* Finder flags */
        struct {
            u_int16_t   v;              /* folder's location */
            u_int16_t   h;
        } frLocation;
        int16_t         opaque;
} __attribute__((aligned(2), packed));
typedef struct FndrDirInfo FndrDirInfo;
```

The “flags” are listed in `bsd/hfs/hfs_macos_defs.h`, and shown in Listing 16-5.

LISTING 16-5: Finder Flags, from bsd/hfs/hfs_macos_defs.h

```
enum {
        /* Finder Flags */
        kHasBeenInited          = 0x0100,
        kHasCustomIcon          = 0x0400,
        kIsStationery           = 0x0800,
        kNameLocked             = 0x1000,
        kHasBundle              = 0x2000,
        kIsInvisible            = 0x4000,
        kIsAlias                = 0x8000
};
```

The flags and finder information are defined as Apple internal. If you compare the previous listings to TN1150, you will see that flags have been removed and the structure fields and names changed. Also, as noted previously, Finder makes use of the `com.apple.FinderInfo` extended attribute to store such information as file color labels (which were once also supported by finder flag, `kColor`).

Case Sensitivity (HFSX)

File systems are defined as case-insensitive or case-sensitive, depending on whether they consider letter uppercase/lowercase when comparing filenames. Additionally, while a file system may be case-insensitive, it may still opt to be case-preserving — i.e., create files in the exact case passed to it, and maintain that case in all further operations on that file.

HFS+ is case-insensitive, but case-preserving. OS X supports a newer variant, HFSX, which can be made case-sensitive, as well. Originally, HFSX was devised as a forward-looking file system that, one day, would replace HFS+. The idea was to enable many more features, updating the version number as more features are added, but so far (since version 10.3 to the present day), the only feature is case-sensitivity, and it, too, is optional.

OS X uses HFS+ by default. iOS uses HFSX, with case-sensitivity enabled. The decision between case-preserving (HFS+) and case-sensitive (HFSX) can only be made once, during partitioning (with Disk Utility or `diskutil(8)` from the command line), since it affects the ordering of keys in the catalog tree.

Journaling

File transactions can be quite complicated, and write operations in particular may span multiple blocks. In the case of a power outage or other crash, this could lead to data corruption, if a transaction is only partially written to the underlying media. Long time UNIX users are all too familiar with the `lost+found` directory, set up automatically on each file system after running `fsck(1)`. This directory contains lost, or orphaned `inodes`, which have been unlinked from their directory by `rm(1)` or `unlink(2)`, yet whose storage blocks have not been freed. In extreme cases, the entire file system may be corrupted and rendered unmountable by a crash. This results in the system booting in single user mode for recovery, and a tedious manual `fsck` by the administrator.

Journaling is a technique that aims to resolve this. The journal is a special area of the disk, allocated but invisible to the user, in which the file system can record its transactions, prior to actually committing them to the disk. If the changes can be committed successfully, they are removed from the journal. But if a crash should occur, the file system can quickly be restored to a consistent state — by either replaying the journal (i.e., committing all its recorded transactions), or rolling it back (in the case it contains incomplete transactions).

A journal is no panacea against data loss. Some data may still be lost, either as a result of a rollback, or due to never making it to the journal in the first place (for example, if it stays in the system buffer cache, and isn't flushed before a crash). It does, however, significantly reduce the chance of a crash making the file system unusable.

Modern file systems, like Linux's Ext3, and Microsoft's NTFS are journal-based. HFS+ can be mounted either with or without a journal. Journaling is default, though SSD-based Macs may benefit from disabling it (due to the number of erase operations in a journal, which could shorten the underlying flash).

Journaling can be toggled on and off as desired, using `hfs.util –J` or `hfs.util –U`, respectively, as shown in Output 16-6. Note the use of the full path name, since `hfs.util(8)` is not in the path.

OUTPUT 16-6: Toggling journaling using hfs.util

```
root@Minion (/)# /System/Library/Filesystems/hfs.fs/hfs.util -J /
Allocated 24576K for journal file.

root@Minion (/)# /System/Library/Filesystems/hfs.fs/hfs.util -I /
/ : journal size 24576 k at offset 0x15502000


root@Minion (/)# mount
/dev/disk0s2 on / (hfs, local, journaled)
devfs on /dev (devfs, local, nobrowse)
map -hosts on /net (autofs, nosuid, automounted, nobrowse)
map auto_home on /home (autofs, automounted, nobrowse)

root@Minion (/)# /System/Library/Filesystems/hfs.fs/hfs.util -U /
Journaling disabled on /dev/disk0s2 mounted at /.

root@Minion (/)# /System/Library/Filesystems/hfs.fs/hfs.util -I /
Volume / is not journaled.

root@Minion (/)# mount
/dev/disk0s2 on / (hfs, local)
devfs on /dev (devfs, local, nobrowse)
map -hosts on /net (autofs, nosuid, automounted, nobrowse)
map auto_home on /home (autofs, automounted, nobrowse)
```

Dynamic Resizing

HFS+ volumes can be dynamically resized — shrunk or grown, even when the volumes are mounted. This is considered advanced functionality, which is not matched by some of its peers (XFS, for example, can grow but not shrink). HFS+ resizing is handled by `hfs_extendfs` (`bsd/hfs/hfs_vfsutils.c`), and can be performed from user mode by a `HFS_RESIZE_VOLUME ioctl(2)`, an `HFS_EXTEND_FS sysctl(2)`, using the Disk Utility GUI by simply adjusting the lower-right corner of an HFS+ partition.

Metadata Zone

The metadata zone, which was introduced in OS X 10.3, follows the system's volume header, and contains the file system's internal structures (alongside hot files, described next). The zone is intentionally defined in the beginning of the volume, to optimize seek times, and is enabled by `hfs_metadatazone_init (bsd/hfs/hfs_vfsutils.c)` under the following conditions:

- Volume size is at least 10 GB
- Journaling is enabled on the volume
- The caller did not explicitly ask to disable the zone (via `fsctl`, as discussed later)

The zone is off limits to regular file allocations (unless the system is extremely short on blocks). The zone contains files and structures for the file system's internal use, as discussed later (under "Components"). The `hfs_virutalmetafile (bsd/hfs/hfs_vfsutils.c)`, shown in Listing 16-6, is used to find if a file belongs in the metazone:

LISTING 16-6: The hfs_virtualmetafile() function

```
int hfs_virtualmetafile(struct cnode *cp)
{
   const char * filename;


   if (cp->c_parentcnid != kHFSRootFolderID)
      return (0);

   filename = (const char *)cp->c_desc.cd_nameptr;
   if (filename == NULL)
        return (0);

   if ((strncmp(filename, ".journal", sizeof(".journal")) == 0) ||
    (strncmp(filename, ".journal_info_block", sizeof(".journal_info_block")) == 0) ||
    (strncmp(filename, ".quota.user", sizeof(".quota.user")) == 0) ||
    (strncmp(filename, ".quota.group", sizeof(".quota.group")) == 0) ||
    (strncmp(filename, ".hotfiles.btree", sizeof(".hotfiles.btree")) == 0))
        return (1);

        return (0);
}
```

Hot Files

An interesting and quite unique feature of HFS+ is its dynamic adaptation to handle frequently accessed files. HFS+ keeps a temperature measurement on each file. The temperature is computed as the number of bytes divided by the file size (as a `uint32_t`, so it is always rounded down). This calculation is inversely proportional to the file size, so it favors small files, whose contents are read very frequently. Those “hot” files exceeding a certain `HFC_MINIMUM_TEMPERATURE` are added to a special B-Tree in the metadata zone, which maintains up to `HFC_MAXIMUM_FILE_COUNT` entries, and their blocks are moved into the metadata zone as well.

The Hot-File B-Tree is a regular file, created by `hfc_btree_create` (in `bsd/hfs/hfs_hotfiles.c`), and its `FndrFileInfo` flags are set (`kIsInvisible + kNameLocked`), so its name cannot be changed, and it remains invisible to Finder, but you can use `ls -laO` to see that it is very much there, as shown in Output 16-7:

OUTPUT 16-7: Locating the hot file B-Tree

```
morpheus@Minion (~)$ ls -laO /.hotfiles.btree
-rw-------  1 root  wheel  hidden 131072 May 11 16:42 /.hotfiles.btree
```

The hot file B-Tree is kept small and contains entries corresponding to the hottest (i.e., most frequently read from) files on the system. The system records file activity and periodically evaluates candidates. Simmering hot files are moved into the metadata zone in a process known as *adoption*, (assuming there is room for them) in place of files which have cooled off, (in what is known as an *eviction*). The eviction precedes the adoption, since it reclaims precious blocks in the limited metadata zone.

Apple intentionally does not document the algorithms, and TN1150 warns they are subject to change. The B-Tree structure of the hot file B-Tree in Lion is presented later in this Chapter, under "Components." The `bsd/hfs/hfs_hotfiles.h` lists the various settings defined for this mechanism (as `HFC_*` constants).

Dynamic Defragmentation

File fragmentation is a bane for all file systems: As the system creates, modifies, and deletes files, "holes" start to appear where files were deleted, and fragments are created when a file needs to expand but has no immediate contiguous space. There may be plenty of file system real estate available, but it's not particularly effective if it's all in studio and one bedroom apartments.

HFS+ is capable of defragmenting files on the fly. The `hfs_relocate (bsd/sys/hfs_readwrite.c)` function handles these cases. It is called from `hfs_vnop_open` (in the same file), and attempts to relocate files that are deemed sufficiently fragmented. This is shown in Listing 16-7:

LISTING 16-7: Handling fragmented files, from hfs_vnop_open

```
int hfs_vnop_open(struct vnop_open_args *ap)
   /*
    * On the first (non-busy) open of a fragmented
    * file attempt to de-frag it (if its less than 20MB).
    */
    fp = VTOF(vp);
    if (fp->ff_blocks &&
        fp->ff_extents[7].blockCount != 0 &&
        fp->ff_size <= (20 * 1024 * 1024)) {
                int no_mods = 0;
                struct timeval now;
                /*
                 * Wait until system bootup is done (3 min).
                 * And don't relocate a file that's been modified
                 * within the past minute -- this can lead to
                 * system thrashing.
                 */
                 if (!past_bootup) {
                        microuptime(&tv);
                        if (tv.tv_sec > (60*3)) {
                                past_bootup = 1;
                        }
                }

                microtime(&now);
                if ((now.tv_sec - cp->c_mtime) > 60) {
                        no_mods = 1;
                }

                if (past_bootup && no_mods) {
                        // relocate past volume next allocation hint, which is
                        // very likely to be contiguous space
```

```
                    (void) hfs_relocate(vp, hfsmp->nextAllocation + 4096,
                                vfs_context_ucred(ap->a_context),
                                vfs_context_proc(ap->a_context));
            }
        }

        hfs_unlock(cp);

        return (0);
}
```

Moving hot files in and out of the metadata zone also helps in defragmentation, as the files are moved by calls to `hfs_relocate()`. The function itself is clearly documented with nice ASCII art, as shown in Listing 16-8:

LISTING 16-8: hfs_relocate(), from hfs_readwrite.c

```
/*
 * Relocate a file to a new location on disk
 *  cnode must be locked on entry
 *
 * Relocation occurs by cloning the file's data from its
 * current set of blocks to a new set of blocks. During
 * the relocation all of the blocks (old and new) are
 * owned by the file.
 *
 * -----------------
 * |///////////////|
 * -----------------
 * 0               N (file offset)
 *
 * -----------------     -----------------
 * |///////////////|     |               |     STEP 1 (acquire new blocks)
 * -----------------     -----------------
 * 0               N     N+1             2N
 *
 * -----------------     -----------------
 * |///////////////|     |///////////////|     STEP 2 (clone data)
 * -----------------     -----------------
 * 0               N     N+1             2N
 *
 *                       -----------------
 *                       |///////////////|     STEP 3 (head truncate blocks)
 *                       -----------------
 *                       0               N
 *
 * During steps 2 and 3 page-outs to file offsets less
 * than or equal to N are suspended.
 * During step 3 page-ins to the file get suspended.
 */
```

HFS+ DESIGN CONCEPTS

The "+" in HFS+ implies it is an enhancement of its predecessor — The *Hierarchical File System*, or HFS. Apple introduced the latter back in the late '80s, to replace the incumbent Macintosh File System (MFS), which was severely limited and incapable of nested folders. HFS proved to have a very solid design, but met its match with files over 2 GB, filenames over 31 characters, and a relatively low number of allocation blocks — only 16-bits worth.

The design of HFS, therefore, wasn't drastically altered in HFS+. The two file systems share the same underlying concepts, which are described next. HFS+ primarily increases field and record sizes, to allow for far more files, and of larger sizes. Where new features in HFS+ were added, they will be pointed out. Apple has gradually begun to phase out support for HFS, retaining only HFS+. Snow Leopard no longer offers HFS file system format, and provides read-only support of HFS-formatted DMG (Disk Image) files. Apple provides a wonderfully detailed explanation of HFS+, including the differences from its precursor, in Technical Note TN1150[2]. TN1150 has grown to be the definitive reference on HFS+, and — while the discussion here is in depth — you are encouraged to take a look at it, as well.

B-Trees: The Basics

B-Trees are fundamental building blocks of file systems, such as NTFS (Windows), Ext4 (Linux) — and Apple's HFS and HFS+. While they are covered in detail in many a textbook, they provide three out of the five supporting data structures in HFS+. This section aims to quickly refresh some concepts, as they are implemented in the file system.

Motivation for B-Trees

The most fundamental concept in any file system is the mechanism used to store and retrieve the files. A file system needs a mechanism that answers several run-time needs:

- **Searches:** Since the primary goal of a file system is to locate files, it must be able to retrieve files in the most efficient manner possible. Since the number of files tends to be very large, this calls for sub-linear time — `O(n)` simply isn't scalable for millions of files. Searches are often hierarchical, as files are put into folders, and folders are put into subfolders still.
- **Insertions:** Though relatively less frequent than locating files, from time to time files are added to the file system. This translates into an insertion of a file entry.
- **Updates:** As files are renamed, moved, and deleted, the mechanism must be flexible enough not to become fragmented. This type of fragmentation, referred to as *index fragmentation*, occurs in cases where file indices, commonly sequential, become sparse as a result of files being moved to some other location, or deleted.
- **Random access:** Though most files are read sequentially, from start to finish, a user or process can always ask to jump around in a file, out of order, commonly by using the `lseek(2)` system call. A file system is fully flexible if, once a file is located, its blocks on disk can be freely accessed, and can be sought through efficiently. Every file system favors writing files contiguously, but this is not always a simple matter. When contents are frequently added or removed from a file, it is only a matter of time before *block fragmentation* ensues, as the file allocation on disk simply cannot be kept contiguous, and the file has to extend to other blocks.

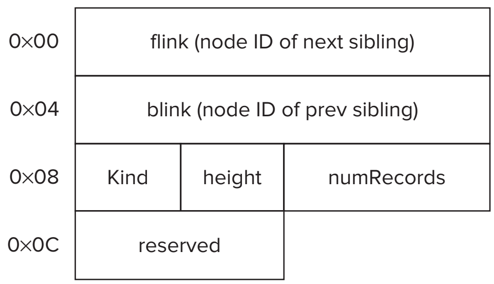
While some file systems remain allocation table based (most notably, FAT, FAT32, and, recently, ExFat — all based on a "File Allocation Table"), most adopt a tree-based solution. Trees, by design, offer all of the above, and provide a hierarchical structure a flat table cannot, "for free." Trees are not without limitations, however. Binary trees only allow for dichotomies at each node. And, as is well known to any computer science major, worst-case operations on trees that involve rebalancing them can be very costly.

Enter *B-Trees*. These can be thought of as an extension to binary trees, in that they maintain a tree structure, but a node can have any number of children — call it *m* — and not just two. This helps to limit their depth, from $\log_2(n)$ (as would be a classic binary tree), to $\log_m(n)$ in the best case, and $\log_{m/2}(n)$ in the worst. Searching, therefore, and most other operations, can be provided at logarithmic time, though in fairness it should be pointed out this is *amortized*. Worst case insertions and deletions are far more costly, although very rare.

The HFS+ logic uses B-Tree operations in `bsd/hfs/hfscommon/BTree`.

B-Tree Nodes

Like all trees, B-Trees are comprised of *nodes*, but unlike other trees, B-tree nodes can be of specific subtypes, or kinds. Different node kinds may hold different data, but all kinds of nodes are derived from a basic type (think, a parent class). They therefore all share the same typical structure: A Node descriptor, followed by 0 or more records. The node descriptor format is exactly the same for all node kinds, and is defined as a `BTNodeDescriptor` in `<hfs/hfs_format.h>`. The structure, along with its in memory representation, is shown in Figure 16-1.




```
/*  BTNodeDescriptor  --  Every B-tree node starts with these fields. */
Struct BTNodeDescriptor {
       u_int32_t          flink;      /* next node at this level*/
       u_int32_t          blink;      /* previous node at this level*/
       int8_t             kind;       /* (leaf, index, header, map)*/
       u_int8_t           height;     /* zero for header, map; child ++ */
       u_int16_t          numRecords; /* number of records in this node*/
       u_int16_t          reserved;   /* reserved - initialized as zero */
} _attribure_((aligned(2), packed));
typedef struct BTNodeDescriptor BTNodeDescriptor;
```

FIGURE 16-1: The B-Tree Node Descriptor

With each row in the illustration representing 32-bits, you can see the common descriptor takes a constant size of 14 bytes. Every node in a B-tree, whether node or internal, also contains 0 or more records. These immediately follow the node descriptor, but may be of variable length. To walk through them, B-Tree nodes place a pointer to the individual records starting at the end of the node, and going back, including a dummy record for any free space which might be contained in the node. This is shown in Figure 16-2.

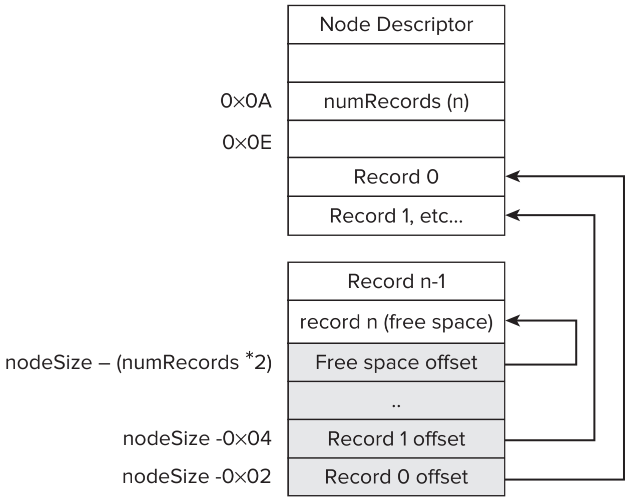


FIGURE 16-2: B-Tree node records

While this approach requires all nodes in the B-tree to have the same size, it allows for the quick traversal of a node's records, as is shown in the following code:

```
void walkNodeRecords (UInt8 *rawNodeData, UInt16 nodeSize)
{

  BTNodeDescriptor *currentNodeDesc = (BTNodeDescriptor *) rawNodeData;

  // Find number of records - note this is stored in Big Endian format.
  UInt16 numRecords = be16_to_cpu(currentNodeDesc->numRecords);
  UInt16 currRec, recordOffset, nextRecordOffset;

  // set a record offset pointer, by going to the end of the node, and
  // count back record offset pointers from it. Each offset pointer is a
  // UInt16. We count back (numRecords + 1): This accommodates for the free
  // space record, as well.

  UInt16 *recordOffsetPtr = (UInt16 *)
     (rawNodeData + nodeSize - sizeof(UInt16) * (numRecords + 1));
  for (currRec = 0;
       currRec < numRecords;
       currRec++)
  {
    // we can now treat recordOffsetPtr as an array of UInt16!
    // we can walk it back, by looking at numRecords - recordNumber

   recordOffset     =  be16_to_cpu(recordOffsetPtr[numRecords - currRec]);
   nextRecordOffset =  be16_to_cpu(recordOffsetPtr[numRecords - currRec -1]);
```

```
    // Our record data is therefore at &rawNodeData[recordOffset]

     /* ... Do something with record data ... */
    }

  }
```

The records themselves are dependent on the kind of node containing them. Internal nodes contain index records, which point to child nodes, whereas leaf nodes contain actual data. Both, however, are *keyed records*, and share the same general record format: A key, followed by data.

The keys *must* be stored in increasing order, and must be unique. I.e., a node cannot contain two identical keys. The key format is shown in Figure 16-3

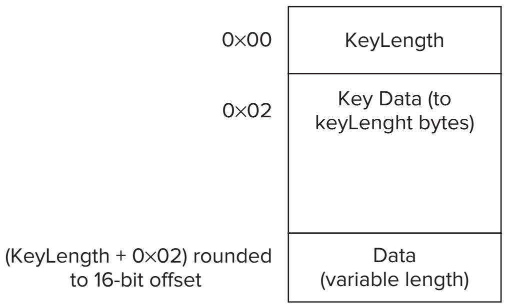


FIGURE 16-3: A B-tree record key

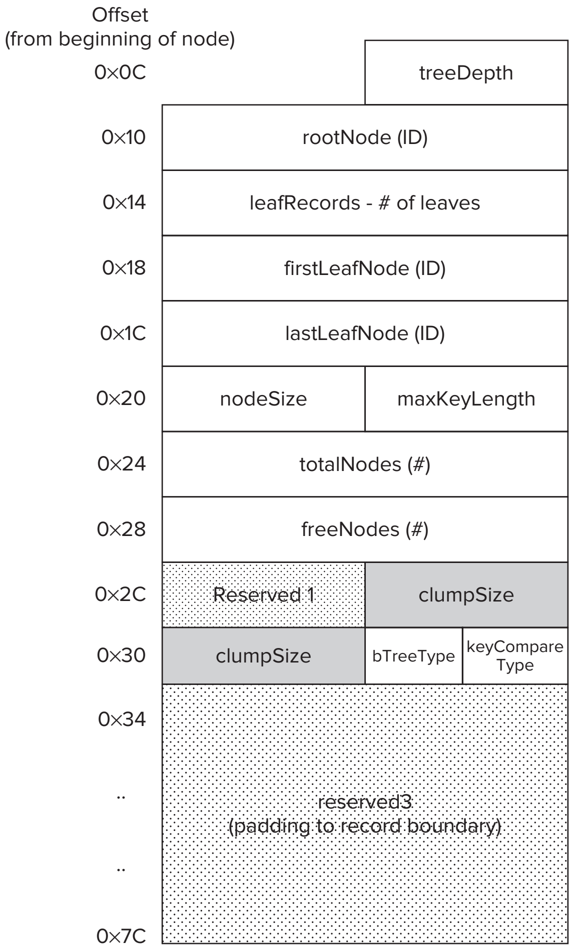
The B-Tree Header Node

The HFS+ B-Tree begins not with a root node, but a special node called the *header* node. This node, of node kind `kBTHeaderNode(1)`, is present even if the tree itself is empty. It contains exactly three records, which are *not* keyed records:

The *header record* contains all the tree metadata. Since it begins immediately after the descriptor, its first field (`treeDepth`, indicating the number of levels in the tree) is a 16-bit quantity, which neatly aligns all other fields (but one, the clump size) on a 32 bit boundary. It is exactly 106 bytes long, which means the next record will start at offset 128 — 32- and 64-bit aligned. The B-Tree header record is shown in Figure 16-4:

The HFS+ B-Tree always has a fixed depth. That is, all of its leaf nodes are on the same level. This depth is defined by the `treeDepth` field. Nodes can be quickly looked up by their ID: As the illustration above shows, the header node contains the ID of the tree root, from which all tree searches begin. Alternatively, the header node allows for quick access to the leaves themselves. This can be used for either sequential or reverse order searches, as the header node provides the index of the first and last leaf, respectively.

Note, that IDs aren't stored anywhere. Each node is always of a fixed size (the `nodeSize` field, in offset `0x1c`), and the nodes are stored in a contiguous node array, enabling the `O(1)` lookup of a node by its ID. This is done by a simple calculation of multiplying the node ID by the header node specified node size.



keyCompareType	Compares
kHFSCaseFolding (0×CF)	Case Insensitive
kHFSBinaryCompare (0×BC)	Case sensitive (HFSX)

FIGURE 16-4: The B-Tree Header record

Following the header record is the *User Data Record* — also exactly 128 bytes long, which is currently reserved. The only B-Tree to actively employ it is the Hot File tree, which is described later.

The last record in the header node is the *Map Record*. It encompasses all the remaining space in the node. This is a bitmap, specifying which nodes in the B-Tree are used, and which are available. If the available space in the node does not suffice, then additional node usage is recorded in one or more special *Map Nodes*, which are single-record nodes that continue the bitmap to cover all nodes in the tree, up to `totalNodes`.

The companion tool for this book, `hfsleuth`, can be used to dump the header node of any of the four B-Trees that are described in this chapter. The example here shows a dump of the main catalog:

```
root@minion (/)#  hfsleuth /dev/rdisk0s2 -b catalog
Processing Catalog tree
Catalog B-Tree dump:
```

```
Tree type:          0
Tree depth:         4
Root node:          32088
First leaf:         14751
Last leaf:          20273
Leaf records        1990354
Total nodes:        77312
Free nodes:         18305
Node size:          8192
Map node:           63104
Compare:            CF
```

Searching the B-Tree

Irrespective of which of the four B-trees is searched, the search logic is always the same. The following pseudo code describes the procedure:

```
void *searchKeyInBTree (void *Key, char *BTreeRawData)
{

  BTHeaderRec *bTreeHeaderRec = (BTHeaderRec *) (BTreeRawData +
                               sizeof(BTNodeDescriptor)); // i.e. + 14

// ASSERT (bTreeHeaderRec->btreeType == kHFSBTreeType); // == 0

  UInt16 nodeSize = be16_to_cpu(treeHeaderRecord->nodeSize);
  UInt16 maxDepth = be16_to_cpu(treeHeaderRecord->treeDepth);

  UInt32 rootNodeID = be32_to_cpu(bTreeHeaderRec->rootNode);


  return (searchKeyInBtreeNode(Key, rootNodeID, BTreeRawData, nodeSize, maxDepth));

} // end searchKeyInBTree

recordData *searchKeyInBTreeNode (key *Key,
                            UInt32 currentNodeID,
                                  char *BTreeRawData,
                                  UInt16 nodeSize,
                                  UInt16 maxDepth)
{

  ASSERT (maxDepth > 0); // sanity check

  char * rawNodeData = (BTreeRawData + nodeSize * currentNodeID);
  BTNodeDescriptor *currentNodeDesc = (BTNodeDescriptor *)(rawNodeData);

  // Loop over records in current node
  // q.v. record walking example: we find number of records in this node
  UInt16 numRecords = be16_to_cpu(currentNodeDesc->numRecords);

  // set a record offset pointer, from end of node
  UInt16 *recordOffsetPtr = (UInt16 *) (rawNodeData + nodeSize
                              - sizeof(UInt16) * (numRecords + 1)];
  for (UInt16 currRec = 0;
```

```
          currRec < numRecords;
          currRec++)
      {
        UInt16 recordOffset     = be16_to_cpu(recordOffsetPtr[numRecords - currRec]);
        UInt16 nextRecordOffset = be16_to_cpu(recordOffsetPtr[numRecords - currRec -1]);
        // Our record data is therefore at &rawNodeData[recordOffset]
        key *recordKey = (key *) (&rawNodeData[recordOffset]);
        recordData *data = (&rawNodeData[recordOffset + (keyLenRoundedToEven(recordKey)]

   // Assume availability of some comparison function, which returns
   // -1 if a < b, +1 if a > b, and 0 on equality
   switch(compareKeys (Key, recordKey))
        {
          case -1: break; // less than - continue

          case 0:         // equal - found, or fall through to recurse
           if (currentNodeDesc->kind == kBTLeafNode)

               return (recordData); // found - return record..

          case 1:         // greater than, or equal and not leaf
           if (currentNodeDesc->kind == kBTLeafNode) return NULL;

                // if NOT a leaf, this HAS to be an index node.
                ASSERT (currentNodeDesc->kind == kBTIndexNode);
                // and if our key is greater, we have to recurse - the data
                // in an index node is the next node ID.
                return (searchKeyInBtreeNode(Key,
                                   (UInt32) recordData,
                                    BTreeRawData,
                                    nodeSize,
                                  --maxDepth));
        } // end switch
      } // end for ..
   } // end searchKeyInBTreeNode
```

COMPONENTS

As mentioned before, HFS+ uses six special files for its own maintenance. Four of them are actually B-Trees:

- **The Catalog B-Tree:** Which contains all the files in the file system.
- **The Attributes B-Tree:** Which was added in HFS+, supports extended file attributes
- **The Extent Overflow B-Tree:** For files with more than eight fragments, or extents.
- **The Hot-File B-Tree:** For small files that are frequently accessed, as discussed previously under "Hot Files."

And two are files:

- **The Allocation File:** Containing a bitmap records of all the blocks in the file system, to track which are in use and which are free.

- **The Startup File:** This is a simple executable file, which can be used for booting the operating system. This is largely ignored by OS X, but can be used by foreign operating systems.

When HFS+ is mounted with journaling, a third file, the Journal, is also used. All these components (including the journal, but excluding the Startup file) are stored in the metadata zone, as well as the quota support files, if quotas are enabled on the volume.

This section describes these components, in detail.

The HFS+ Volume Header

Before the system can start rummaging through miscellaneous B-Trees, it has to be able to find where they are, and identify the HFS+ file system as such. For this purpose, there exists at a fixed location — 1024 bytes from the beginning of the partition (or "Volume"). This is a massive structure — 512 bytes — but it contains all the necessary details required to initiate the file system loading operation. The volume header is shown in Figure 16-5.

The volume header is also, at present, the only cardinal difference between HFS+ and HFSX: The two are identical in nearly every way, with three exceptions:

- HFSX uses the signature `HX` as opposed to HFS+, which uses `H+`.
- HFSX sets the version to 5, rather than HFS+ setting 4.
- In HFSX B-Trees have an option to perform key comparison by binary compare, or by folding the case.

Most of the fields shown in the figure are self-explanatory, but one that needs some elaboration is `FinderInfo`: As noted previously, HFS+ is a rather unusual file system in that it is tightly integrated with the Finder GUI. The `FinderInfo` fields are used by OS X during a boot operation from the volume, and by Finder, upon volume mount. There are eight fields, defined in Table 16-2.

TABLE 16-2: FinderInfo fields in the HFS+ volume header

FIELD	USED FOR
0	Holding the folder Catalog Node Identifier of `/System/Library/CoreServices`, on a bootable volume
1	Holding the folder ID of Finder (or another startup application) on a bootable volume
2	The folder ID of a folder to auto-open on mount
3	Deprecated; previously used to OS 8 or 9 boot folder
4	Reserved
5	Same as [1], for OS X systems
6-7	Unique volume identifier, as 64-bits

The HFS+ volume catalog, as the crucial data which it is, is backed up by an *Alternate Volume Header*, located at the end of the volume — just 1024 bytes before its end. As it occupies exactly 512 bytes, the last 512 bytes of a volume are unused, and reserved.

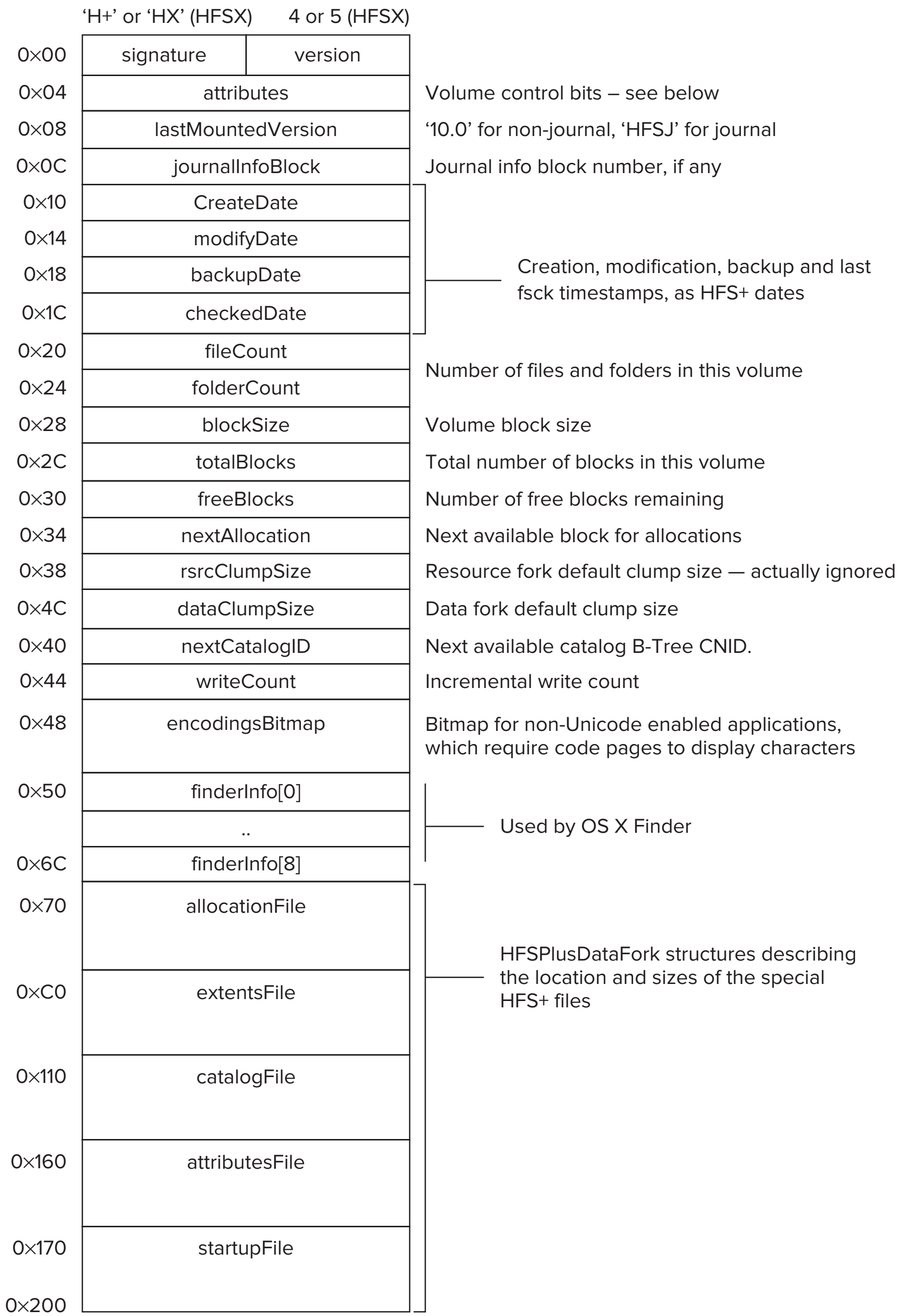


FIGURE 16-5: The HFS+ Volume header

The Catalog File

The main B-Tree of the HFS+ file system is the *catalog*. The catalog contains entries for all the files and the folders in the system, i.e., the `fileCount` files and `folderCount` folders mentioned in the volume header. The system uses this in all file operations: listing, searching, reading, writing and deleting. So it is only fitting that it be the primary focus for this section.

As a B-Tree, the catalog inherits the structure and all the properties previously discussed for generic HFS+ B-Trees. The catalog introduces several new properties:

➤ The Catalog Node ID or CNID is a unique 32-bit identifier of a file or folder. Apple reserves the first 16 CNIDs, but the rest of the namespace is readily allocated by the file system. CNIDs are generally allocated in a monotonically increasing order — by taking the `nextCatalogID` value from the volume header, and incrementing it as each new file or folder is created. At some point, however, they may run out (i.e., after some 4-billion or so files are created). In that case, they wrap around, and the volume header `kHFSCatalogNodeIDsReusedBit` attribute bit is set. At that point, the file system must check the Map record(s) to find the next available CNID.

➤ Catalog file Keys are defined to be a structure, as shown in Listing 16-9:

LISTING 16-9: The HFSPlusCatalogKey

```
struct HFSPlusCatalogKey {
    UInt16              keyLength;
    HFSCatalogNodeID    parentID;
    HFSUniStr255        nodeName;
};
typedef struct HFSPlusCatalogKey HFSPlusCatalogKey;
```

Where *parentID* is the CNID of the parent folder, and the *nodeName* is a Unicode string of the type described in "Unicode Support." To bootstrap the process, the CNIDs reserved by Apple may be used. Specifically, `kHFSRootParentID` (1) — the (fake) parent of the root folder, i.e., the partition itself, is used to obtain the partition name, and `kHFSRootFolderID` (2) is used for the root folder.

➤ Catalogs may contain one of four distinct record types:

  ➤ `kHFSPlusFolderRecord` types (1) store folder data as an `HFSPlusCatalogFolder`. Likewise, `kHFSPlusFileRecord` types (2) store file data as an `HFSPlusCatalogFile`.

  ➤ `kHFSPlusFolderThreadRecord` (3) and `kHFSPlusFileThreadRecord` store "threads." A thread, in both cases, is an `HFSPlusCatalogThread`, defined as shown in Listing 16-10:

LISTING 16-10: The HFSPlusCatalogThread

```
struct HFSPlusCatalogThread {
    SInt16              recordType;
    SInt16              reserved;
    HFSCatalogNodeID    parentID;
    HFSUniStr255        nodeName;
};
```

Thread records are used when looking up a file or folder by its CNID, as is described next.

Catalog Lookups

There are two types of catalog lookups:

- Lookup by file or folder name
- Lookup by CNID

Looking up a path name is performed by breaking the pathname into its constituents, and iteratively looking up each, in turn, beginning with the root folder. As an example, consider the pathname `/private/etc/passwd`:

The first lookup will be for `/private`. To find it, we treat `private` as a name under the root folder. The root folder CNID is well known — `kHFSRootFolderID(2)` — so we prepare its catalog key. (See Figure 16-6.)

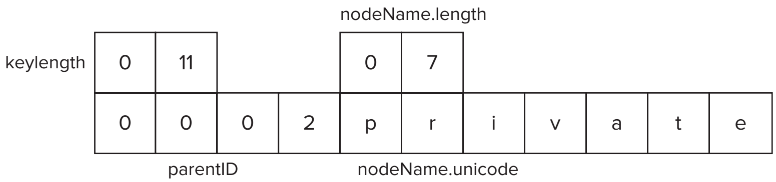


FIGURE 16-6: The catalog key for /private

This will yield a folder, i.e., an `HFSPlusCatalogFolderRecord`. Of its many fields, we care only about one — `FolderID`. This is the CNID of the `/private` folder. In our example, it is 24. The next lookup is shown in Figure 16-7.

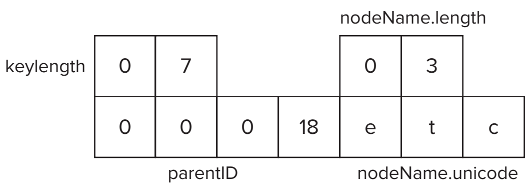


FIGURE 16-7: The catalog key for /etc, as a subfolder of /private (CNID 24=0x18)

As before, this is expected to yield an `HFSCatalogFolderRecord` — yielding the folder ID 1075. This would give us the key shown in Figure 16-8 for our file.

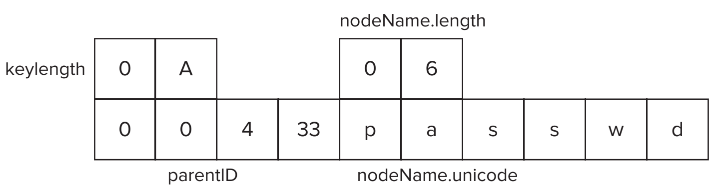


FIGURE 16-8: The Catalog key for passwd, in the folder /private/etc (CNID 1075=0x433)

Giving us the much sought after `HFSCatalogFileRecord` we want. The following pseudo-code in Listing 16-11 demonstrates the breakdown process:

LISTING 16-11: Walking the B-Tree in search of a file

```
#define PATH_SEPARATOR L'/'
//
// pseudo code only - this destroys the inputted PathName..
//
key * fileNameToCatalogKey (char *PathName)
{
   key *returned = malloc (..);
   UInt32 parentCNID = kHFSPlusRootFolderID; // start at the root folder
   char *sep = strchr (PathName, PATH_SEPARATOR)

   while (sep)
   {
      *sep = 0;  // Replace '/' with NULL, so pathname is now parent dir
       parentCNID = getFileCNID (parentCNID, PathName);
       PathName= ++sep; // PathName is now whatever follows the parent
       sep = strchr(PathName, PATH_SEPARATOR);
   }

   // if we are here, what's left of the pathname is a file/folder name
   // and parentCNID holds our containing folder
   returned.parentID = parentCNID;
   returned.nodeName.length = cpu_to_be16(strlen(PathName));
   copyAndFlipUnicode(&returned.nodeName.unicode, PathName);
}
```

If the CNID of the object is known, it can be searched using a thread record. For this, we set up a key where in the node name is empty, and set the `parentID` to the CNID we are seeking. i.e, to look up CNID 1075, we would set up a key as shown in Figure 16-9:

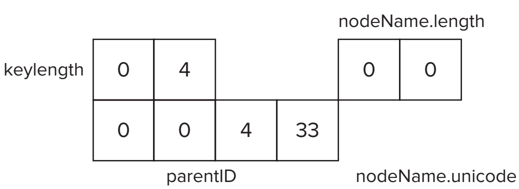


FIGURE 16-9: A thread catalog key for an object with CNID 1075 (=0x433)

This would yield a thread record, containing the data in (ii), i.e., the file name. From there, we can look up its corresponding file or folder record, as before.

The `hfsleuth` tool can perform either lookups, and — using the –v(erbose) feature — can also detail the stages along the way:

```
root@minion (/)# ~/hfsleuth /dev/rdisk0s2 -v -s /System/Library/Extensions
Processing Catalog tree
<Record node="191" num="3" offset="430">
```

```
             <Key len="6"><CNID>38</CNID>
             <Data type="folderThread">
               <parentCNID>37</parentCNID>
               <Name>Library</Name>
              </Data>
             <Path>/System </Path>
       </Record>
<Record node="5" num="26" offset="3024">
       <Key len="6"><CNID>41</CNID><Name/>
       <Data type="folderThread">
         <parentCNID>38</parentCNID><Name>Extensions</Name>
       </Data>
       <Path>/System/Library</Path>
</Record>
..
<Record node="14751" num="1" offset="134">
             <Key len="6"><CNID>2</CNID><Name/>
             <Data type="folderThread">
               <parentCNID>1</parentCNID>
               <Name>Macintosh HD</Name>
              </Data>
             <Path>/</Path>
      </Record>

..
```

Catalog Insertions

When files are created, records need to be inserted into the Catalog tree. This is a straightforward method over the normal B-Tree insert, shown here:

```
insertNameIntoCatalog (char *PathName, char *BtreeRawData)
{
  BTHeaderRec *bTreeHeaderRec = (BTHeader *) (BTreeRawData +
                                 sizeof(BTNodeDescriptor)); // i.e. + 14

  ASSERT (bTreeHeaderRec->btreeType == kHFSBTreeType); // == 0

  UInt16 nodeSize = be16_to_cpu(treeHeaderRecord->nodeSize);
  UInt16 maxDepth = be16_to_cpu(treeHeaderRecord->treeDepth);

  UInt32 rootNodeID = be32_to_cpu(bTreeHeaderRecord->rootNode);

  key *fileKey = *fileNameToKey (PathName);
  return (insertKeyIntoBtree(fileKey, rootNodeID, BTreeRawData, nodeSize,
 maxDepth));

}
```

Catalog Deletions

Likewise, file deletion is a direct override of the B-Tree deletion method:

```
DeleteNameIntoCatalog (char *PathName, char *BtreeRawData)
```

```
{
  BTHeaderRec *bTreeHeaderRec = (BTHeader *) (BTreeRawData +
                               sizeof(BTNodeDescriptor)); // i.e. + 14

  ASSERT (bTreeHeaderRec->btreeType == kHFSBTreeType); // == 0

  UInt16 nodeSize = be16_to_cpu(treeHeaderRecord->nodeSize);
  UInt16 maxDepth = be16_to_cpu(treeHeaderRecord->treeDepth);

  UInt32 rootNodeID = be32_to_cpu(bTreeHeaderRecord->rootNode);

  key *fileKey = *fileNameToKey (PathName);
  return (deleteKeyFromBtree(fileKey, rootNodeID, BTreeRawData, nodeSize,
 maxDepth));

}
```

File and Folder Record Data

HFS+ stores similar data for files and folders. The following illustration compares the `HFSCatalogFolderRecord` and `HFSCatalogFileRecord`. (See Figure 16-10.)

As can be seen, the two structures are designed to be compatible. Most of the fields overlap, and those that have specific meaning for directories (i.e., `valence` and `folderCount`) are reserved in the file record. Likewise, file specific information — i.e., the forks — are implemented after the end of the common information block.

Permissions

Both catalog record formats contain the `bsdInfo` member, which is `struct HFSPlusBSDInfo`:

```
struct HFSPlusBSDInfo {
        u_int32_t       ownerID;        /* user-id of owner or hard link chain previous
link */
        u_int32_t       groupID;        /* group-id of owner or hard link chain next
link */
        u_int8_t        adminFlags;     /* super-user changeable flags */
        u_int8_t        ownerFlags;     /* owner changeable flags */
        u_int16_t       fileMode;       /* file type and permission bits */
        union {
            u_int32_t   iNodeNum;       /* indirect node number (hard links only) */
            u_int32_t   linkCount;      /* links that refer to this indirect node */
            u_int32_t   rawDevice;      /* special file device (FBLK and FCHR only) */
        } special;
} __attribute__((aligned(2), packed));
typedef struct HFSPlusBSDInfo HFSPlusBSDInfo;
```

This structure is the one to implement the back end of the `chown(1)`, `chmod(2)`, `chgrp(2)`, and `chflags(1)` commands. Figure 16-11 shows the mapping of those commands to the structure's fields.

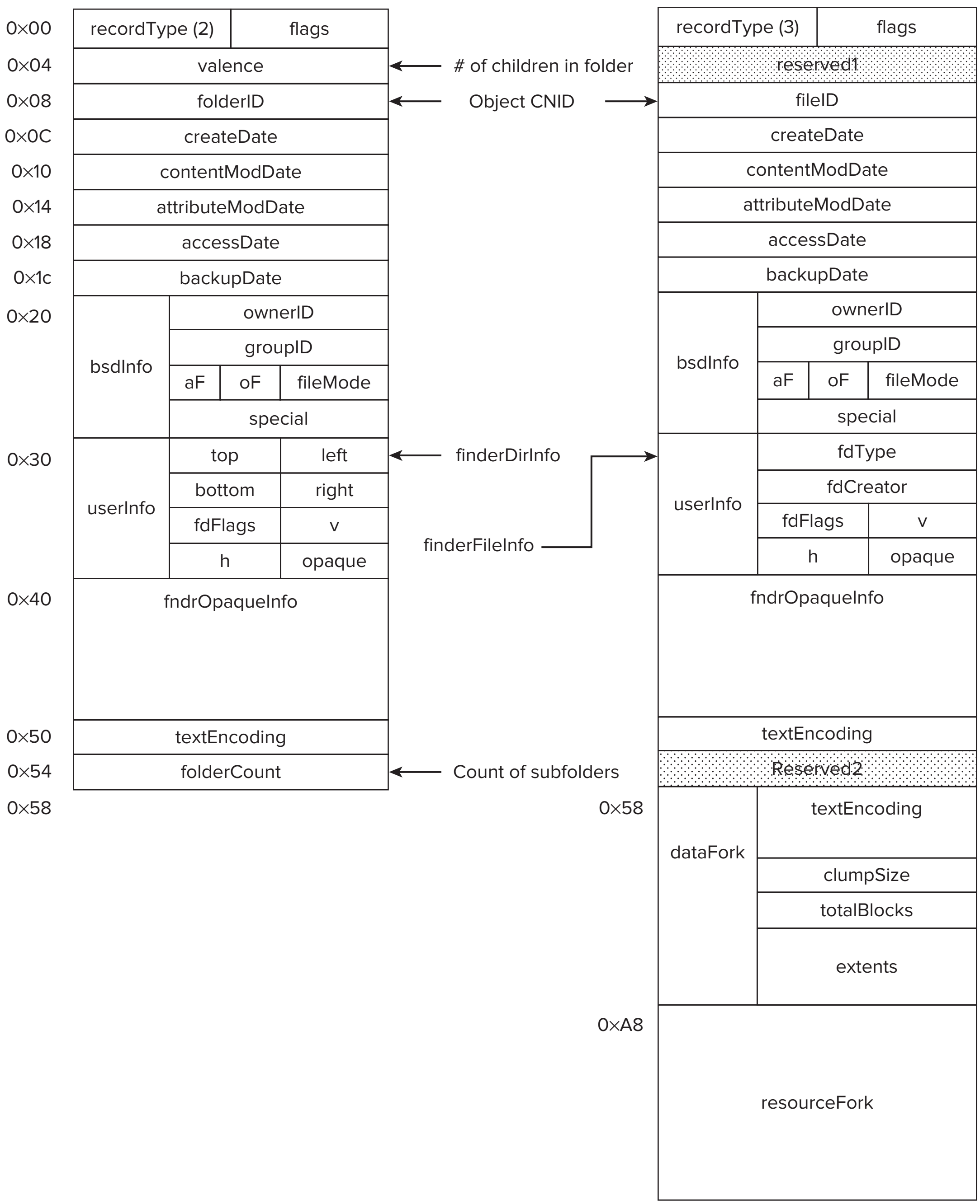


FIGURE 16-10: Comparing HFSCatalogFolderRecord and HFSCatalogFileRecord

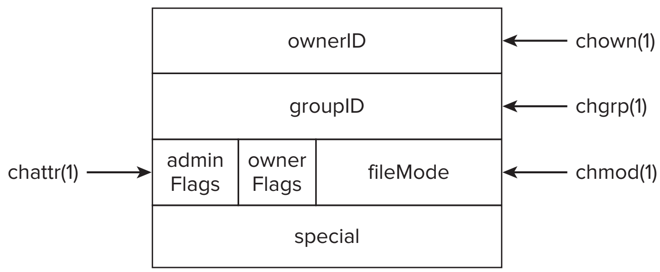


FIGURE 16-11: The UNIX permissions, encoded in HFS+ file and folder records

Hard and Soft Links

HFS+, as any other UNIX file system, supports both hard and soft links. The underlying mechanism, however, is very particular.

Both hard and soft links are distinguished by the `fileType` field of the `userInfo` catalog record. For hard links, this field is a magic value of `0x686c6E6b` (`hlnk`) and — similarly `0x736c6e6b` (`slnk`) for soft links. In both cases, the creator code is `hfs+`.

For soft links, the special handling ends there: Soft links are otherwise regular files, whose contents contain the name of another file on the file system.

Hard links, however, receive special handling by the system. As soon as a hard link is created, the underlying file's forks are relocated — not to say, stashed — in a private and secluded part of the file system — The `\0\0\0\0HFS+ Private Data` directory. HFS+ goes to great lengths to keep this directory hidden and inaccessible. It is invisible to both the UNIX utilities (as it begins with NULL bytes, which terminate C-Strings), and to the Finder (which, additionally, obeys the `kIsInvisible` and `kNameLocked` flags).

The dentries for the hard links exist in their respective locations just as normal files, but their resource forks (and thus, sizes) are set to 0. Instead, the "special" field of BSD Info is set to the `inode` Number of the file, which can be retrieved from `\0\0\0\0HFS+ Private Data`.

Fork Allocation

File records offer two `HFSPlusForkData` structures — one for the resource fork and one for the data fork. As stated before, HFS+ can support any number of named forks (via the Attribute tree, described next), though if forks are at all used, only the data fork is commonly used.

The file's block list is kept in the `dataFork` member. This member is also a `struct`, whose members specify the fork's logical size, as well as clump size. A third member specifies the extents, and is an array of up to eight `HFSPlusExtentDescriptor` structures, each containing an extent `startblock` and `blockCount`. This is shown in Figure 16-12.

Most files don't need more than 8 extent descriptors. In fact, most do quite well with one, if they are allocated once, and take up exactly one extent. But as a file shrinks and grows, it might become fragmented, and require more extents. If the sum of the (`extents[i].blockCount`) is exactly the same as specified in `totalBlocks`, the file can be accessed in its entirety from its record. Otherwise, if it is less (think — it cannot be more!), this indicates some extents spilled over — in which case we need to look them up in the extent B-tree, described later.

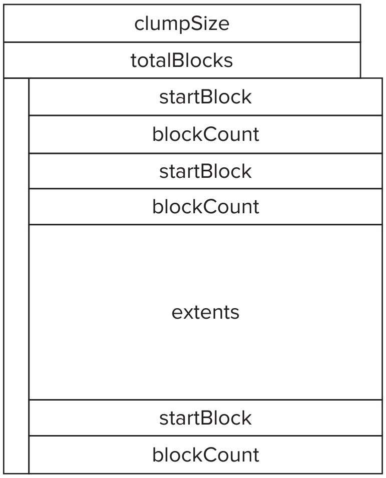


FIGURE 16-12: The fork data structure

The Extent Overflow

As we saw while reviewing the Catalog records, most files fit snugly in eight extents or less. Files with more than eight are considered heavily fragmented, but should obviously still be serviced by the file system. For this, the file system maintains another B-Tree, called the extent overflow B-Tree.

The extent overflow B-Tree is a far simpler B-Tree than the catalog file. Unlike the catalog file, it does not contain multiple index records — only leaves.

The Attribute B-Tree

Another B-Tree used by HFS+ is the Attribute B-Tree. This is used by HFS+ to store various extended attributes. The B-Tree format is defined in `bsd/hfs/hfs_format.h` under the `__APPLE_API_UNSTABLE` warning, but has actually been solid enough to merit inclusion in this book. The relevant definitions are shown in Listing 16-12:

LISTING 16-12: Attribute B-Tree data structures

```
/*
 * Atrributes B-tree Data Record
 *
 * For small attributes, whose entire value is stored
 * within a single B-tree record.
 */
struct HFSPlusAttrData {
        u_int32_t    recordType;   /* == kHFSPlusAttrInlineData */
        u_int32_t    reserved[2];
        u_int32_t    attrSize;     /* size of attribute data in bytes */
```

```
        u_int8_t      attrData[2];  /* variable length */
} __attribute__((aligned(2), packed));
typedef struct HFSPlusAttrData HFSPlusAttrData;

/*      A generic Attribute Record*/
union HFSPlusAttrRecord {
        u_int32_t               recordType;
        HFSPlusAttrInlineData   inlineData;   /* NOT USED */
        HFSPlusAttrData         attrData;
        HFSPlusAttrForkData     forkData;
        HFSPlusAttrExtents      overflowExtents;
};

typedef union HFSPlusAttrRecord HFSPlusAttrRecord;

/* Attribute key */
enum { kHFSMaxAttrNameLen = 127 };
struct HFSPlusAttrKey {
        u_int16_t     keyLength;       /* key length (in bytes) */
        u_int16_t     pad;             /* set to zero */
        u_int32_t     fileID;          /* file associated with attribute */
        u_int32_t     startBlock;      /* first allocation block number for extents */
        u_int16_t     attrNameLen;     /* number of unicode characters */
        u_int16_t     attrName[kHFSMaxAttrNameLen];   /* attribute name (Unicode) */
} __attribute__((aligned(2), packed));
typedef struct HFSPlusAttrKey HFSPlusAttrKey;
```

For most intents and purposes, user mode applications need not care about this B-Tree, because the attributes can be listed, obtained and set with the `listxattr(2)`, `getxattr(2)`, and `setxattr(2)` system calls, respectively. There are, however, extended attributes which will not be visible by means of these system calls. Those include the `com.apple.cprotect` and `com.apple.system.security` shown in Table 16-1. Fortunately, the hfsleuth tool can display the attributes by reading them directly from the Attributes B-Tree.

The Hot File B-Tree

The last B-Tree used by HFS+ is the hot file B-Tree. The tree header is defined (along with all other related definitions) in `bsd/hfs/hfs_hotfiles.h`, as shown in Listing 16-13:

LISTING 16-13: The Hot-File B-Tree header

```
/*
 * B-tree header node user info (on-disk).  // (hasn't changed from TN1150)
 */
struct HotFilesInfo {
        u_int32_t       magic;       // HFC_MAGIC, 0xFF28FF26
        u_int32_t       version;     // HFC_VERSION, 1
        u_int32_t       duration;    /* duration of sample period (secs) */
        u_int32_t       timebase;    /* start of recording period (GMT time in secs) */
        u_int32_t       timeleft;    /* time remaining in recording period (secs) */
        u_int32_t       threshold;
        u_int32_t       maxfileblks;
        u_int32_t       maxfilecnt;
        u_int8_t        tag[32];     // hfc_tag = "CLUSTERED HOT FILES B-TREE     "
};
```

The B-Tree key is keyed by `temperature` and `fileID` (which is the CNID of the hot file in question), as shown in Listing 16-14. Because the temperature is what the system needs to look up most frequently, it can set the key to `HFC_LOOKUPTAG` for lookup purposes:

LISTING 16-14: The Hot-File B-Tree key format

```
struct HotFileKey {
    u_int16_t       keyLength;      /* length of key, excluding this field */
    u_int8_t        forkType;       /* 0 = data fork, FF = resource fork */
    u_int8_t        pad;            /* make the other fields align on 32-bit boundary */
    u_int32_t       temperature;    /* temperature recorded  - set to HFC_LOOKUPTAG */
    u_int32_t       fileID;         /* file ID */
};
```

The actual hot file data structures are implemented in `hfs_hotfiles.c`, no doubt to keep them as private as possible.

The Allocation File

The allocation file is a rather large, yet inaccessible file that keeps track of all the blocks in the volume. It is designed as a simple bitmap, wherein each bit corresponds to a block, and is lit if the block is in use (or, potentially, a bad block). Its size is a direct function of the volume size and block size, and can be calculated directly as `(Volume size / block Size) / 8`, as the volume contains (`volume size / block size`) blocks, and each block occupies a single bit.

Because the allocation file is a file in itself, it may be fragmented. This makes it a very extensible scheme, if the volume is enlarged — the allocation file can simply grow. It is, however, usually contiguous — and contained in a single extent — because it is created as part of the `mkfs` program. This also makes it relatively easy to dynamically change the allocation block size in the file system.

The recent version of HFS (in Lion) has introduced the notion of a red-black tree-based allocator (`#ifdef CONFIG_HFS_ALLOC_RBTREE`). This is somewhat similar to XFS's method of allocating blocks, providing the more efficient R-B tree as an allocation mechanism that can quickly find contiguous blocks as the disk becomes more and more fragmented. A separate kernel thread is created and starts `hfs_initialize_allocator()` to create two R-B trees from the volume bitmap (for the metadata zone and for the rest of the volume). Note, that these trees are created in-memory, and have no on-disk representation, and, therefore, there is no need to change the file system disk structure.

HFS Journaling

Recall the previous discussion of journaling. In HFS+, journaling is a feature that can be freely toggled, though the stated default is enabled. When mounting a file system, HFS+ checks the value of the `lastMountedVersion` field in the volume header. This field can take on one of several values, as shown in Table 16-4.

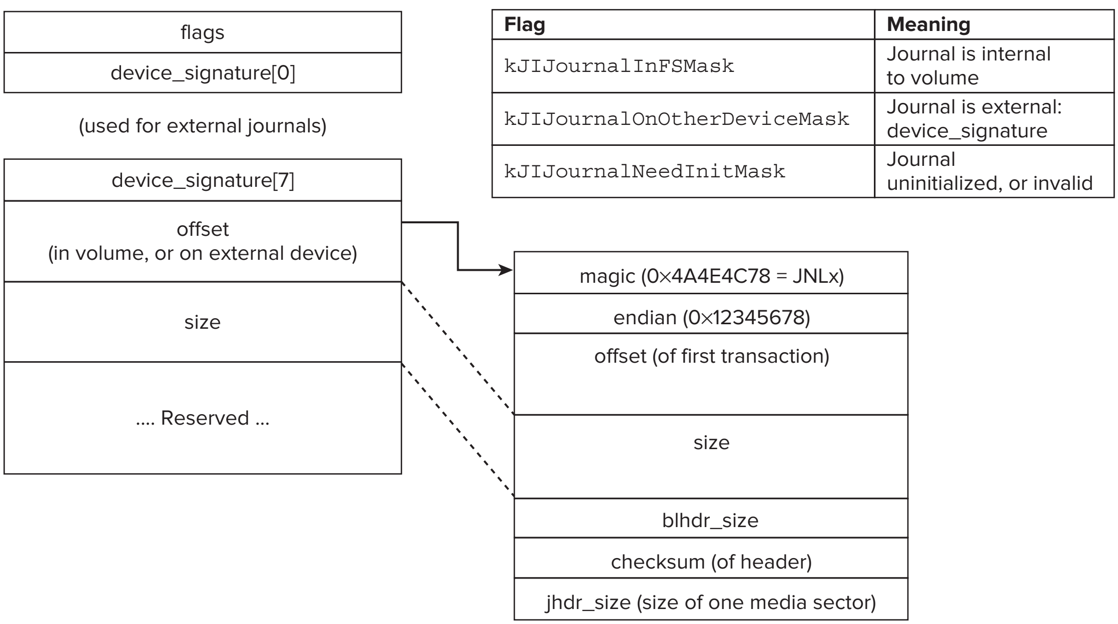
TABLE 16-4: lastMountedVersion

VALUE	HEX	MEANING
`10.0`	`31 30 2e 30`	File system was last mounted by an OS X implementation, yet journaling was not enabled.
`HFSJ`	`48 46 53 4a`	File system was last mounted by an operating system (OS X or other) which did enable the journal
`fsck`	`66 73 63 6b`	File system was last mounted by `fsck(1)` — meaning it is likely some type of file system recovery was performed

This field is especially important during the mount operation, because it tells the system if there is a need to consult the journal, or it can be ignored. If the file system was indeed mounted with journaling, and no `fsck` pass was conducted, it is quite plausible that there would be some transactions in the journal, and it is, therefore, deserving of an inspection. Otherwise, if the last mount was with no journal, consulting the journal would actually be risky, potentially leading to the replay of stale transaction data. Likewise, the HFS+ driver is expected to update `lastMountedVersion` according to the journal option selected for mounting (or toggled during the file system lifetime).

Locating the Journal

To access the journal, the system needs to first read the `journalInfoBlock`, from the volume header (offset `0x0C`). This is an actual LBA offset in the volume, so the next step is to load the block into memory. Its format is as shown in Figure 16-13.



Flag	Meaning
`kJIJournalInFSMask`	Journal is internal to volume
`kJIJournalOnOtherDeviceMask`	Journal is external: device_signature
`kJIJournalNeedInitMask`	Journal uninitialized, or invalid

FIGURE 16-13: The Journal info block

The journal info block is used to find the journal, which is usually somewhere inside the file system (i.e., internal to the volume), but could actually also be on a separate device. The first field, flags, defines *either* `kJIJournalInFsMask` (`0x01`) or `kJIJournalOnOtherDeviceMask` (`0x02`). If the journal is internal, we proceed normally, by checking the offset field. If the journal is on another device, however, the `device_signature` field reserved 32 (`=8*sizeof(UInt32)`) bytes for providing a hint as to where the device is, and offset pertains to somewhere on that device.

The next step is to load the journal header from the specified offset. The journal header is checked and double checked:

First, the system verifies the block read begins with the "magic" field (`JOURNAL_HEADER_MAGIC`, or `JNLx`).

Next, the system verifies `ENDIAN_MAGIC` (`0x12345678`), to make sure the journal is in the right endian-ness (little or big).

Then, the system verifies the journal size in the header matches the size reported in the journal info block.

Finally, the journal header checksum is computed.

The checksum is a simple checksum, not unlike an IP header checksum, or other. TN1150 shows the following code from Listing 16-15, which is straightforward:

LISTING 16-15: Journal checksum calculation

```
static int calc_checksum(unsigned char *ptr, int len)
{
    int i, cksum=0;

    for(i=0; i < len; i++, ptr++) {
        cksum = (cksum << 8) ^ (cksum + *ptr);
    }

    return (~cksum);
}
```

This same checksum logic is applied all over the journal, as journal data blocks must also be checksummed. The rationale behind it is that this way, it is easy to detect an incomplete transaction in the journal itself (i.e., one wherein the checksum on the block is invalid).

Reading through Journal Transactions

If the header is intact, its `start` and `end` pointers point to the transactions in the journal. Two pointers are necessary because the transactions are stored in a circular (ring) buffer on the disk. The buffer is of size (`size - jhdr_size`), and starts immediately at the end of the header (but on a sector boundary, hence `jhdr_size` is always rounded to the size of a sector).

There are several possible scenarios for start and end:

- **`start == end`** — This means the journal is intact, and empty. The journal can never be full.

- **`start < end`** — The journal has transactions, which are stored in a contiguous range between the two pointers. All other blocks are stale, and must be ignored.
- **`start > end`** — The journal has transactions, but wraps. Therefore, start reading normally (at `start`), but when the journal read operation gets to the end of the buffer (which can easily be found by `&header + size`), it must wrap as well, and continue from (`&header + jhdr_size`) until `end`.

Journal Transaction Format

The journal transactions are recorded as an array of `block_list_header` structures. These are structures of size `blhdr_size` (as specified in the journal header). This structure is as shown in Figure 16-14.

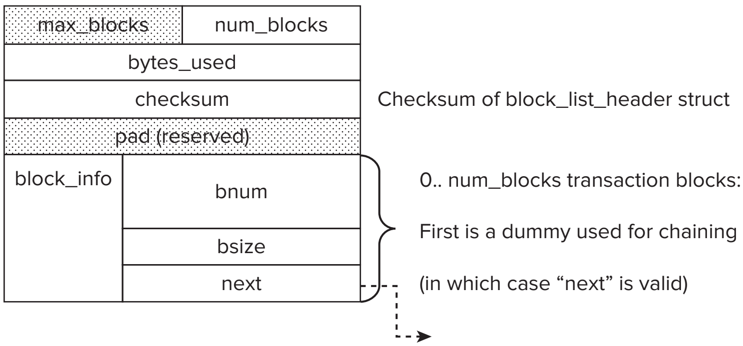


FIGURE 16-14: The Journal block_list_header

A transaction normally spans (`num_blocks -1`) blocks. The first `block_info` field (which is the only one defined in the `block_list_header struct`) is actually a dummy block, which is used if transactions range over more than one block list. In such cases, where the number of blocks in a transaction is more than the number of blocks, transactions can chain block lists together. The file system driver can quickly deduce if that is the case by looking at the "next" field — if it is non-zero, the next block list is at the offset it points to.

The `block_info` is basically a directive indicating that the `bsize` bytes which follow need to be written at block number `bnum` on this volume.

VFS AND KERNEL INTEGRATION

HFS+ has several advanced features, stemming from both its design and its integration with OS X's VFS mechanisms. I describe them here.

fsctl(2) integration

The HFS+ code exposes registers `hfs_ioctl` (`bsd/hfs/hfs_readwrite.c`) as its `fsctl` handler. If VFS's `fsctl_internal (bsd/vfs/vfs_syscalls.c)` receives a control code it does not recognize, it passes it to `hfs_ioctl`, which can recognize and act on the codes listed in Table 16-5:

TABLE 16-5: HFS+ fsctl codes, defined in bsd/sys/hfs/hfs_ioctl.h

CODE	USAGE
HFS_GETPATH	Retrieve path name corresponding to CNID
HFS_PREV_LINK HFS_NEXT_LINK	Retrieve the next or previous link
HFS_RESIZE_VOLUME	Dynamically resize an HFS+ volume. Calls hfs_extendfs() or hfs_truncatefs() internally
HFS_RESIZE_PROGRESS	Report HFS+ resize progress
HFS_CHANGE_NEXT_ALLOCATION	Manually set next allocation
HFS_SETBACKINGSTOREINFO HFS_CLRBACKINGSTOREINFO	Supports sparse devices, for example in disk images, whose space on disk may be significantly lower than the space reported to the file system #if HFS_SPARSEDEV, but enabled by default
HFS_BULKACCESS_FSCTL	Access multiple files in bulk
HFS_SET_XATTREXTENTS_STATE	Extent-based extended attribute support (Default as of Lion). Settable by root only
HFS_FSCTL_SET_LOW_DISK HFS_FSCTL_SET_VERY_LOW_DISK HFS_FSCTL_SET_DESIRED_DISK	Set low disk space notification conditions (see "File System Status Notifications," later)
HFS_VOLUME_STATUS	Get volume status information
HFS_GET_BOOT_INFO HFS_SET_BOOT_INFO	Get or set boot information (the FinderInfo). The SET code is root only
HFS_MARK_BOOT_CORRUPT	Force fsck on next mount (sets kHFSVolumeInconsistentBit in volume header)
HFS_FSCTL_GET_JOURNAL_INFO	Get Journal information
HFS_SET_ALWAYS_ZEROFILL	Fill new files with zeros
HFS_DISABLE_METAZONE	Disable the metadata zone (root only)

In addition to the HFS+ specific codes, `hfs_ioctl` can also handle some generic codes (`F_*` constants), such as `F_FREEZE_FS` and `F_THAW_FS`, `F_[READ|WRITE]_BOOTSTRAP`, and others.

sysctl(2) integration

The HFS+ code exposes the `vfs.hfs` MIB, with an instance for each `mountd` HFS+ file system. Using the `sysctl(8)` command line utility yields little, as it will simply report the number of mounted instances. Programmatically, however, this mechanism can be used to set HFS+ parameters on the mounted file systems. Some of this functionality is also accessibly via `fsctl(2)`, as well. These parameters are shown in Table 16-6.

TABLE 16-6: sysctl(2) MIBs exported by HFS+ (all are leaves)

SYSCTL MIB	PURPOSE
HFS_ENCODINGBIAS HFS_ENCODINGHINT	Set cjk encoding — one of the `kTextEncodingMac`
HFS_EXTEND_FS	Same as `HFS_RESIZE_VOLUME fsctl`, but only allows `hfs_extendfs()`
HFS_ENABLE_JOURNALING HFS_DISABLE_JOURNALING	Toggle journaling on/off
HFS_GET_JOURNAL_INFO	Only supported for 32-bit processes, but otherwise same as `HFS_FSCTL_GET_JOURNAL_INFO`
HFS_SET_PKG_EXTENSIONS	Used by LaunchServices
VFS_CTL_QUERY	Query file system
HFS_ENABLE_RESIZE_DEBUG	Debugging for volume resizing

File System Status Notifications

The HFS+ code in the kernel can generate kernel events when several threshold conditions are met. The thresholds are low disk or dangerously low disk space, defined in `bsd/sys/hfs/hfs.h` to be 98% or 99% utilization (respectively) for a regular volume, and 90% or 95% for a root volume. The thresholds may also be set by means of the `HFS_FSCTL_SET_[VERY_]LOW_DISK` control codes.

The notification are generated by the `hfs_generate_volume_notifications` function, which is the sole denizen of `bsd/vfs/hfs_notification.c`. The function checks for low disk space conditions (such as calls on `vfs_event_signal` (`bsd/vfs/vfs_subr.c`), which generates a knote, which can be read the `EVFILT_FS` filter.

Disabling or enabling the journal will also generate a notification, by directly calling `vfs_event_signal` directly from the `hfs_sysctl` handler.

SUMMARY

This chapter described HFS+ and its variant, HFSX, the native file system format for OS X and iOS. First, following an explanation of HFS+ features (mostly inherited from XNU's VFS layer), we described HFS+ in detail.

The underlying data structure of HFS+ is a B-Tree, and the file system uses several of them — for its main catalog, to store file extents, file attributes and metadata. HFS+ has been built in and around OS X, with features added on the go as OS X evolved. This is also part of its shortcomings: Hard link support is crude, the native data format is still big-endian (forcing byte swaps frequently) and 16/32-bit optimized (limited to 2^{32} blocks). HFS+ also lacks advanced features such as sparse file support and snapshots). Apple has hinted, but so far resisted calls for supporting a newer standard, such as ZFS.

REFERENCES

1. Spotlight MetaData Attribute Reference, `https://developer.apple.com/library/mac/#documentation/Carbon/Reference/MetadataAttributesRef/Reference/CommonAttrs.html`
2. Technical Note TN1150 — HFS Plus Volume Format, `http://developer.apple.com/legacy/mac/library/#technotes/tn/tn1150.html`

17

Adhere to Protocol: The Networking Stack

A fundamental portion of the kernel in contemporary operating systems is devoted to networking, and the same holds true for OS X and iOS. In both, the networking system is a near-exact copy of the BSD networking logic, implementing the classic POSIX model of BSD sockets, which is common to all UN*X. Like BSD, both systems support specific extensions, such as the Berkeley Packet Filter (BPF) and firewalling. Socket support in XNU is actually optional, depending on the `CONFIG_SOCKETS` option, though needless to say it is enabled by default in both OS X and iOS.

This chapter sets as its focus the implementation of the network stack. Following a brief overview of the user mode perspective, which lists the available protocols and various statistics in XNU, we dive into the network stack architecture, layer by layer. (See Figure 17-1.) As in most systems, XNU is responsible for layers II through V. We therefore proceed from the application layer downwards: Starting with sockets, which make up layer V, through the transport protocols of layer IV (TCP/UDP), and the network protocols of layer III (IPv4/IPv6), and finally discussing the network interfaces, which make up layer II. Additional topics, such as packet filtering and QoS are also discussed.

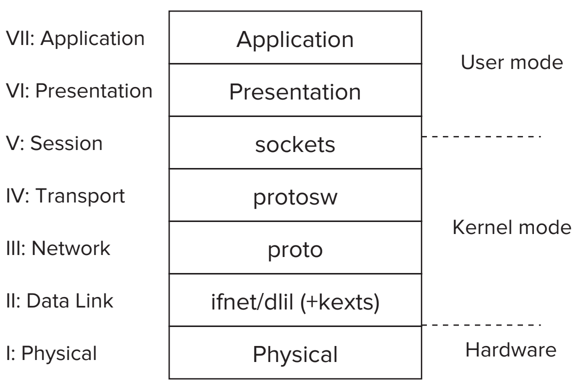


FIGURE 17-1: The OSI (7 layer) model and its relation to the network stack

Throughout the chapter it is assumed that the reader is already familiar with the basic concepts of sockets and the API, whether from the common Windows port (Winsock) or from POSIX. You can find a comprehensive reference for socket programming in Stevens' books, by which UN*X developers swear[1, 2]. Likewise, because the socket code is so close to that of BSD's, this chapter focuses more on the Apple extensions (which are, at times, contained in an `#if __APPLE__` block), and less on the code common to BSD. Several great books whose sole focus is the BSD kernel are available[3], and the avid reader is encouraged to check them out, as well.

Note that the average Cocoa developer doesn't need to know anything about sockets. This is because of the Core Foundation classes, which abstract sockets by CFSocket and CFStream, and the further protocol-aware abstractions of CFFTP, CFHTTP, and the like, offered by CFNetwork. Nonetheless, BSD sockets lie at the root of all networking on XNU (and practically all modern operating systems, including (to an extent) Windows). That, by itself, merits a dedicated chapter.

USER MODE REVISITED

The BSD socket model was designed with multiple protocol support in mind. The most basic operation, creating a socket, calls for three parameters: the address (or protocol) family, the socket type, and the protocol.

The "family," often referred to as an Address Family (AF) or Protocol Family (PF), denotes the socket addressing mode corresponding to the layer 2 or layer 3 addresses. Many such modes exist, and the most widely used one, IP, is but one; for example, `PF_INET` (or `AF_INET`).

There are numerous `PF_/ AF_` constants and they are all defined in `<sys/socket.h>`. Though technically the `PF_` constants should be used, traditionally the `AF_` ones have been. The `PF_` constants are just `#define`d over the `AF_` ones, so they may be used interchangeably. Both OS X and iOS support only a very limited subset of families, namely the ones shown in Table 17-1:

TABLE 17-1: Supported Address Families on OS X and iOS

#	FAMILY	USED FOR
1	`PF_LOCAL`	UNIX domain sockets. Also available as `AF_/PF_UNIX`.
2	`PF_INET`	IPv4 sockets.
14	`PF_LAT`	Local area transport sockets. Only on Snow Leopard.
17	`PF_ROUTE`	Routing sockets.
27	`PF_NDRV`	Network driver. Raw access to network device. Apple extension.
29	`PF_KEY`	IPSec Key Management (RFC2367). `#if IPSEC`.
30	`PF_INET6`	IPv6 sockets. `#If INET6` Can also be used for IPv4 when IPv4 mapped addresses (`::FFFF:a.b.c.d`) are used.
32	`PF_SYSTEM`	System/kernel local communication.

Unless otherwise stated, both OS X (Snow Leopard and Lion) and iOS support these families.

Note, that while these are very close to the address families in BSD, there are some deviations (most notably `PF_NDRV` and `PF_SYSTEM`, which are idiosyncratic to Apple). Address families may also be registered on demand, by kernel extensions. A good example is `PF_PPP`, for Point-to-Point Protocol support. Unlike Linux, protocols such as BlueTooth are not supported over sockets (i.e. there is no `PF_BLUETOOTH`), but over IOKit.

The socket API is designed to be as agnostic as possible to family idiosyncrasies, and therefore deals with the generic struct `sockaddr struct`, which the programmer is expected to cast back and forth from the actual `struct sockaddr_*` specific to the family used (e.g. `sockaddr_un` for `AF_UNIX`, and `sockaddr_in6` for `AF_INET6`). These structures all overlap with the first field of `struct sockaddr`, the `sa_family`, by means of which the kernel may direct the address-related operation to the right provider.

UNIX Domain Sockets

UNIX domain sockets were among the first forms of interprocess communication on UNIX, predating the now ubiquitous IP sockets. They are unique to UNIX-based systems, and they are of local scope only (i.e. inner-host, rather than inter-host) and are therefore less known or popular than their IP brethren. Nonetheless, they are still noteworthy, as they remain an important staple of UN*X systems, OS X and iOS included.

Though restricted to local scope, UNIX domain sockets offer one significant advantage over their IP brethren — namely, the ability to pass file descriptors and credentials over the socket. This makes them very useful for multi-process programming. Note that, in the case of XNU, Mach ports can be passed in messages, and the new fileport system calls can further be used to pass descriptors, but neither of these capabilities conform to POSIX.

UNIX domain sockets bind to local filenames. These, however, are not truly files. The filesystem presence is required to help system-wide uniqueness and visibility. Most sockets can be found in `/var/run`, and will be displayed by default as part of `netstat(8)` output (or specifically, with `netstat -f unix`). A detailed discussion of UNIX domain sockets can be found in Stevens', and many other books.

IPv4 Networking

Sockets are nowadays synonymous with IP, and to a large extent the socket APIs owe their widespread adoption to IP's popularity, and vice versa. As the protocol became more popular, sockets became the preferred API to it. As socket APIs grew more popular, IP became people's first choice.

Mac OS, somewhat like Windows, didn't immediately adopt TCP/IP. Microsoft originally had hopes for IPX/SPX (which reigned shortly, back when Novell still dominated servers), and Apple clung for a while to its proprietary AppleTalk protocol suite, which implemented an entire network stack*. Apple, however, eventually got bored of talking to itself, and so TCP/IP eventually prevailed. AppleTalk support was gradually phased out in OS X, and finally dropped in Snow Leopard, with its main application layer protocol, The Apple Filing Protocol (AFP), converted to function over IP.

*In fairness, Mac OS was an early adopter of TCP/IP with MacTCP, and TCP/IP coexisted with AppleTalk for a while. It was only in after the merger with NeXT, though that TCP/IP officially prevailed.

Apple maintains a fairly up-to-date list of TCP and UDP protocols used by Mac operating systems in TS1629[4]. Most of these protocols are standard (e.g. HTTP, SSH, etc). There are, however, a few Apple proprietary protocols, most of which are poorly documented (if at all) to this very day. These include:

- **mDNS (Bonjour, etc)**: Multicast DNS (or mDNS, for short) is a form of serverless DNS service meant to assist devices in local name resolution. The packet structure is the same as that of DNS[5] but instead of a name server, a multicast request is sent out to 224.0.0.251 (or FF02::FB) on UDP port 5353.

  Microsoft uses a very similar, though not fully compatible protocol called LLMNR (Link Layer Multicast Name Resolution). LLMNR operates on UDP port 5355, and uses the multicast address of 224.0.0.252 (or FF02::1:3).

  Bonjour is the protocol responsible for Macs popping up whenever you find yourself in a public network, such as an airport lounge (and is a great way to discover other people's musical tastes while delayed). It is, in a sense, a legacy of AppleTalk, which provided the same ad-hoc functionality.

- **EPPC (Apple events)**: Event Process-to-Process Communication is the protocol that allows for remote Apple events. It is an intentionally undocumented proprietary protocol that is disabled by default. OS X supports eppc URLs, which — similarly to FTP URLs — allow the specification of a `user:password@host`. The URI component (`"/folder"`) in these URLs is the name of some application. EPPC is carried over TCP port 3031.

- **DAAP (Airplay, iTunes)**: The Digital Audio Access Protocol (DAAP) is an Apple proprietary streaming protocol. It is not part of OS X as much as it is of iTunes, wherein, as the name implies, it is used to access remote iTunes libraries. DAAP is carried over TCP port 3869.

- **AFP (Time Machine, File Sharing)**: The Apple Filing Protocol is another legacy of AppleTalk, which is still actively developed by Apple. It is carried over TCP port 547, and is used when connecting to file servers like the Time Capsule, or when enabling File Sharing from System Preferences ➪ Sharing. The protocol bears similarities to Microsoft's Server Message Block (SMB) and NFS, in that it allows remote mounting of shares, and is optimized for interoperability with HFS+ filesystems. The protocol is somewhat documented by Apple[6], and has been implemented by third parties.

Routing Sockets

The `PF_ROUTE` family is a BSD standard to control routing tables from user mode. It is described in Stevens' book in great detail, and is largely unused outside routing utilities. A comprehensive example of its usage can be found in the open source of the `route(8)` command[7], which is part of the `network-cmds` package. It is not supported outside BSD systems, though Linux achieves (and, to an extent, exceeds) its functionality with NetLink.

Network Driver Sockets

OS X and iOS support `PF_NDRV`, which is a protocol family intended for use by network drivers. This is a little known, but quite useful, socket type, which enables the crafting of raw packets — all the way down to the data link layer — from user mode. This is similar in concept to the standard

SOCK_RAW of IP, but goes one layer lower, and enables full control over the link layer header (usually, Ethernet), as well. In that respect, it is the OS X equivalent of Linux's PF_PACKET. Though powerful, it is generally unused by the masses: libpcap, for example, prefers BPF (discussed later). Apple does use this internally, and implements EAPOL[8] (802.11x) over it.

NDRV sockets bind to local interface names (e.g. en0, en1). This binding, however, does require root privileges. Once the socket is bound, unadulterated access to the interface is at your fingertips. Because NDRV is so scarcely documented (and so darn useful!), the following experiment demonstrates its usage by example.

As (unjustly) unpopular as the NDRV mechanism is, it still provided for a creative use unfathomed by its original developers. An integer overflow vulnerability in an NDRV ioctl(2) *helped liberate iOS 4.3.1. Though this required root permissions, the resulting overflow allowed the "evil" jailbreakers to overwrite arbitrary kernel memory, and then further exploiting the Mach zone allocator to untether a jailbreak. A detailed discussion of this can be found in Esser's Black-Hat 2011 talk[9]. When it comes to security, more (code) implies less (security).*

Experiment: Spoofing Packets with PF_NDRV

Crafting packets with NDRV is child's play. Just as IP's raw sockets allow the manual crafting of the network and transport header, so do NDRV's socket allow this, and further enable any arbitrary link layer framing. This allows the sending and receiving of packets which aren't even IP, such as ARP/RARP, or 802.1x, all of which exist at layer II.

If you've used raw IP sockets before, you will find Listing 17-1 familiar, mayhap nostalgic. A raw NDRV socket is created, and bound to the interface of choice. The bind() call's sockaddr_ndrv is a sockaddr-compatible structure, using the interface name as the binding "address."

LISTING 17-1: A simple program to spoof packets

```
#include <sys/socket.h>
#include <net/if.h>
#include <net/ndrv.h>


void main(int argc, char **argv) {

   int s;
   int rc;
   struct sockaddr_ndrv sndrv;
   u_int8_t packet[1500];

   if (geteuid() != 0)
    { fprintf (stderr, "You are wasting my time, little man. Come back as root\n");
      exit(1);
    }
```

continues

LISTING 17-1 *(continued)*

```
s = socket(PF_NDRV, SOCK_RAW, 0);                // Open socket
if (s < 0) { perror ("socket"); exit (1);}       // Just in case..
//Bind to interface, say "en0", or "en1"
strlcpy((char*)ndrv.snd_name, "en0", sizeof(sndrv.snd_name));
ndrv.snd_family = AF_NDRV;
ndrv.snd_len = sizeof(sndrv);
rc = bind(s, (struct sockaddr*)&sndrv, sizeof(sndrv));

if (rc < 0) { perror("bind"); exit(2);} // Could fail if interface doesn't exist

// Craft packet!
 memset(&packet, 0, sizeof(packet));

// Destination MAC goes in packet[0] through packet[5]
packet[0] = 0xFF; /* ... */;  packet[5] = 0xFF;

// Source MAC address goes in packet[6] through packet[11]
packet[7] = 0xFF; /* ... */;  packet[11] = 0xFF;

// Ethertype is next two
packet[12] = ...; packet[13] = ...;

 // And data (Layer III and up) follows

 strcpy((char*) &packet[14], "You can put whatever you want here.. \0");

 rc = sendto(fd, &packet, 1500, 0, (struct sockaddr*)&sndrv, sizeof(sndrv));

}
```

From that point on, you can verify packets actually get sent by using a packet capture tool (`tcpdump(1)` or Ethereal). The program in the listing naturally doesn't send anything meaningful, but can be adapted (using `structs` for the various protocols) to craft specialized packets. This is highly useful for various network fuzzing tools and (naturally) malicious packet spoofing.

IPSec Key Management Sockets

RFC2367[10] details the use of IPSec Key Management sockets. This socket type is used rarely outside the realm of security software, and the RFC fully explains the usage of these sockets. The intrigued reader is therefore encouraged to consult this RFC, while this book opts to save a few trees (or kilobytes), and focus on less documented aspects.

IPv6 Networking

Like all modern operating systems, OS X and iOS have built-in support for IPv6, the successor to IPv4 that still hangs around the corner. Numerous times it was rumored to finally succeed the aging Internet protocol, yet reports of the demise of the latter seem to have been greatly exaggerated.

The implementation of IPv6 in XNU, like in Linux or BSD, is in an entirely separate protocol handler. Similar to BSD, it is based on a port of the KAME project[11] (which you can see using `sysctl net.inet6.ip6.kame_version`).

The administrator can use the `ip6(8)` command to enable or disable IPv6 on some or all interfaces. The `ip6config(8)` command can likewise be used.

OS X supports the `stf(4)` interface, to enable 6to4 connectivity. The 6to4 standard, specified in RFC3056[12], is one of the more common to connect to the fledgling IPv6 Internet over the aging IPv4 infrastructure, by using IP-in-IP tunneling. It is a fairly simple matter to establish connectivity, assuming your origin IP is a real (read: non-NATed or RFC1918) IPv4 address, and your egress router allows IP-tunneling (protocol number 41). The system's 6to4 settings are kept in `/etc/6to4.conf` (which uses the 6to4 anycast of 192.88.99.1). To start 6to4, a simple `ip6config start-stf` will usually do. Microsoft IPv6 tunneling (or, more accurately, burrowing) standard, Teredo[13] is not supported natively, but the miredo[14] open source package has been ported to OS X.

OS X also supports BSD's generic tunnel interface, `gif(4)`. This is a more generic tunneling than `stf(4)`'s, specified in RFC2893[15]. Unlike the former, it allows any combination of IPv4 and IPv6 tunneling (6 over 4, 6 over 6, 4 over 4, 4 over 6). Output 17-1 shows how to set up and tear down an IP tunnel:

OUTPUT 17-1: Setting up and tearing down an RFC2893 tunnel using ifconfig gif:

```
root@Minion (/)# ifconfig gif0 tunnel <localv4> <remotev4>
root@Minion (/)# ifconfig gif0 inet6 <localv6> <remotev6> prefixlen 128 up
```

System Sockets

The `PF_SYSTEM` address family is a method for kernel/user-space communication used. The address family supports two protocols: The Control Protocol and the Event protocol.

Kernel Control Protocol

`PF_SYSTEM` sockets aren't widely used in OS X, and are only a bit more common in iOS, as shown in Table 17-2. These sockets can be created though `ctl_register`, which is exported for use by kernel extensions.

TABLE 17-2: Known PF_SYSTEM Control IDs

FUNCTION	REGISTERS CTL
`utun_control_register` `(bsd/net/if_utun.c)`	`com.apple.net.utun_control.` Used for user mode tunnels (`utun##`). This type enables a user mode process to register an interface, and accepts all data from sockets binding to that interface. Discussed later under "Layer II Implementation"
`netsrc_init` `(bsd/net/netsrc.c)`	`com.apple.netsrc.` Private Apple API in Lion and iOS.

continues

TABLE 17-2 *(continued)*

FUNCTION	REGISTERS CTL
`nstat_control_register` (`bsd/net/ntstat.c`)	`com.apple.network.statistics.` Private Apple API used in Lion and iOS for active connection statistics (discussed later under "Socket and Protocol Statistics")
`iptap_init` (closed source, iOS, to be made open in Mountain Lion)	`com.apple.net.iptap_control.` Private and undocumented Apple API (in iOS, and starting with Mountain Lion).
`AppleOnBoardSerialBSDClient` (closed source, iOS)	`com.apple.uart.*.` Private and undocumented Apple API for serial port access in iOS.
`IOUserEthernetController` (`en_register`, closed source, iOS)	`com.apple.userspace_ethernet.` Private and undocumented Apple API for user space Ethernet

To register a kernel control socket, the provider needs to set up a `kern_ctl_reg` structure, specifying the control name, some settings and the callback functions which will provide for the user mode API calls. The provider passes this structure to `ctl_register()` along with a pointer to `kern_ctl_ref`, which will be returned with an opaque handle to use with this socket in the various callback functions. This structure is shown in Listing 17-2:

LISTING 17-2: The kern_ctl_reg structure, from sys/kern_control.h

```
struct kern_ctl_reg
{
 /* control information */
  char            ctl_name[MAX_KCTL_NAME];
  u_int32_t       ctl_id;   // ignored, unless CTL_FLAG_REG_ID_UNIT is specified
  u_int32_t       ctl_unit;

 /* control settings */
  u_int32_t   ctl_flags; // CTL_FLAG_PRIVILEGED - uid 0 processes only
                         // CTL_FLAG_REG_SOCK_STREAM - SOCK_STREAM only, not DGRAM
                         // CTL_DATA_NOWAKEUP - Don't wake up process on data received
  u_int32_t   ctl_sendsize; // override default send size, or leave 0
  u_int32_t   ctl_recvsize; // override default recv size, or leave 0

 /* Dispatch functions  */
 // all return errno. The kern_ctl_reg argument is returned by ctl_register()
 ctl_connect_func    ctl_connect;   //(kern_ctl_ref kcr,sockaddr_ctl *sac,void **unit);
 ctl_disconnect_func ctl_disconnect; //(kern_ctl_ref kcr,u_int32_t unit,void *unitinfo);
 ctl_send_func       ctl_send;       // kern_ctl_ref kcr,u_int32_t unit,void *unitinfo,
                                          mbuf_t m, int flags);
 // ctl_setopt and ctl_getopt are used for get/setsockopts and share the same prototype:
 // kern_ctl_ref kcr, u_int32_t unit, void *unitinfo, int opt, void *data, size_t len)
 ctl_setopt_func     ctl_setopt;
 ctl_getopt_func   ctl_getopt;
};
```

Any of the control registration function in Table 17-2 can provide an example of registration. A more detailed example of kernel controls is shown later in this chapter, in the case study of utun.

Kernel Event Protocol

The second protocol supported by PF_SYSTEM sockets is the SYSPROTO_EVENT protocol, used for kernel events. Using this protocol, a kernel component can broadcast events to listeners in both kernel mode and user mode.

Each event contains a vendor code, a class and a subclass, which enables listeners to filter only those events of interest. Apple is the only registered vendor, with a hard-coded vendor code of 1, though third party kexts can also obtain a runtime vendor code, which can be looked up by the client using a SIOCGKEVVENDOR ioctl(2). Apple currently defines six classes of events, shown in Table 17-3:

TABLE 17-3: Apple Event Classes

EVENT CLASS	USED BY
KEV_NETWORK_CLASS (1)	Network stack. Subclasses include DL (DataLink), INET/INET6 (IPv4/IPv6) and LOG (FW Log)
KEV_IOKIT_CLASS (2)	IOKit drivers
KEV_SYSTEM_CLASS (3)	System events. Currently only used for memory status notifications
KEV_APPLESHARE_CLASS (4)	AppleShare (Unused by kernel proper)
KEV_FIREWALL_CLASS (5)	IPv4 and IPv6 Firewalls (IPFW/IP6FW subclasses, respectively)
KEV_IEEE80211_CLASS (6)	Wireless Ethernet (IO80211Family drivers)

A simple event listener doesn't take more than a few lines of code: It merely requires setting up the socket, optionally setting up a filter request, and reading. This is shown in Listing 17-3:

LISTING 17-3: A simple PF_SYSTEM/SYSPROTO_EVENT listener

```
#include <sys/socket.h>     // for socket(2) and friends
#include <sys/kern_event.h> // for kev_* and kern_event_* types

/**
  * A rudimentary PF_SYSTEM event listener, in 50 lines or less. Works on iOS too
  */
void main (int argc, char **argv)
{

  struct kev_request req;
  char buf[1024];
```

continues

LISTING 17-3 *(continued)*

```
    int rc;
    struct kern_event_msg *kev;

    // Setup the system socket
    int ss = socket(PF_SYSTEM, SOCK_RAW, SYSPROTO_EVENT);

    // Set filtering parameters. Only interested in Apple, but not filtering on
    // classes for now
    req.vendor_code  = KEV_VENDOR_APPLE;  // Apple is pretty much the only vendor
    req.kev_class    = KEV_ANY_CLASS;     // No class filtering (show all)
    req.kev_subclass = KEV_ANY_SUBCLASS;  // No subclass filtering (show all)

    // Use ioctl(2) to set the filter on the socket
    if (ioctl(fd, SIOCSKEVFILT, &req)) {
        perror("Unable to set filter\n"); exit(1);
      }

     while (1) {

       // can use if (ioctl(fd, SIOCGKEVID, &id)) to get next ID
       // or simply read and block until an event occurs..

       rc = read (ss, buf, 1024);

       kev = (struct kern_event_msg *)buf;

       // Print event class and class (data is event dependent)
       // A better implementation would convert class, subclass and code to text
       // and is left as an exercise to the reader.
       //
       printf ("Event %d: (%d bytes). Vendor: %d Class: %d/%d\n",
          kev->id, kev->total_size, kev->vendor_code, kev->kev_class, kev->kev_subclass);

       printf ("Code: %d\n",kev->event_code);

     } // end while

  }
```

Perspicacious Linux-philes may notice that this mechanism is also quite similar in functionality to Linux's NetLink sockets, in that both of these can be used to send messages (particularly network configuration messages) from kernel space. NetLink, however, relies on a form of multicast which is somewhat crude by comparison, and does not enable filtering of messages.

SOCKET AND PROTOCOL STATISTICS

XNU keeps statistics for various sockets and the underlying protocols in read-only `sysctl(8)` variables, in the `net.*` namespace. Address families each hold their own sub-namespace (`local`, `inet`, `inet6`, `key`), with sub-protocols in a third level namespace (`stream/dgram` for `local`, ip/tcp/udp/raw/ipsec for inet, and 6 suffixes for the respective inet6 protocols. key does not have sub-protocols).

Output 17-2 shows the variables in the net.inet.udp space, as an example:

OUTPUT 17-2: Variables in the net.inet.udp space, as viewed by sysctl(8)

```
morpheus@ergo (/)$ sysctl net | grep udp
net.inet.ip.fw.dyn_udp_lifetime: 10
net.inet.udp.checksum: 1
net.inet.udp.maxdgram: 9216
net.inet.udp.recvspace: 42080
net.inet.udp.in_sw_cksum: 3830661
net.inet.udp.in_sw_cksum_bytes: 854082494
net.inet.udp.out_sw_cksum: 4248220
net.inet.udp.out_sw_cksum_bytes: 1189771941
net.inet.udp.log_in_vain: 0
net.inet.udp.blackhole: 0
net.inet.udp.pcbcount: 19
net.inet.udp.randomize_ports: 1
```

By trying sysctl -a net you can see some of the counters and settings, though the interesting ones; those seen in netstat -s are hidden. This is because they are opaque structures, and the sysctl(8) command does not know how to deal with them. Using the -A switch, you can see their names, though their values remain an obscure hex dump.

Commands like netstat(8), however, can parse these values. In particular, netstat -s parses the stats keys of the respective protocols, and — in its common usage — netstat(8) obtains the list of active sockets for each protocols by parsing the pcblist or pcblist64 MIBs. This is an internal list of struct inpcbs, which correspond to active connections (discussed later). The netstat(8) command is open source[16], and you are encouraged to check it for a good example of how these MIBs are parsed. The PF_SYSTEM sockets, discussed previously, can also be used for network statistics: The com.apple.network.statistics identifier (available in iOS and Lion), exposed by nstat_control_register(), offers statistics on network connections, similar to netstat(1), but with the ability to be actively notified on connection establishment and teardown. This constitutes a private API, though bsd/net/ntstat.h offers a fairly good idea of its inner workings.

In brief, this allows a curious user mode process to obtain a list of all active sockets from NSTAT_PROVIDER_UDP, NSTAT_PROVIDER_TCP, and routing information NSTAT_PROVIDER_ROUTE. The statistics include more advanced details than offered by netstat(1), including TCP window information, and owning process name, which in Linux is available by -p. Unlike netstat(1), an application can block on the socket to get notifications of connection establishment and teardown. The nstat mechanism exposes the net.statistics MIB, enabling and disabling the statistics collection through sysctl(8).

The book's companion website offers the ***lsock*** tool, which shows an example of using com.apple.network.statistics from user mode, and will compile on Lion or iOS 4 and later. A sample output from iOS 5 is shown in Output 17-3:

OUTPUT 17-3: lsock on iOS 5, catching apsd red-handed

```
root@Podicum (/)# lsock -p tcp -a
TCP #1, IPv4, If 2, State 4, Pid: 10109 (sshd)   192.168.1.105:22->192.168.1.103:53784
TCP #2, IPv4, If 2, State 4, Pid: 81    (apsd)   192.168.1.105:50785->17.172.232.119:443
TCP #3, IPv4, If 1, State 1, Pid: 2 ()           127.0.0.1:8021 (Listening)
TCP #4, IPv6, If 1, State 1, Pid: 2 ()           ::1:8021       (Listening)
TCP #5, IPv6, If 0, State 1, Pid: 2 ()           :::62078       (Listening)
TCP #6, IPv4, If 0, State 1, Pid: 2 ()           0.0.0.0:62078  (Listening)
TCP #7, IPv4, If 0, State 1, Pid: 2 ()           0.0.0.0:22     (Listening)
TCP #8, IPv4, If 0, State 1, Pid: 2 ()           0.0.0.0:22     (Listening)
```

LAYER V: SOCKETS

Most of the generic socket code in XNU is implemented in several key files, all in `bsd/kern`, shown in Table 17-4:

TABLE 17-4: XNU Socket Implementation Code

FILE	IMPLEMENTS
`uipc_domain.c`	Socket domain (address/protocol family) support
`uipc_mbuf.c`	Support functions for MBUFs
`uipc_mbuf2.c`	More support functions for MBUFs
`uipc_proto.c`	UNIX domain protocol support (`SOCK_STREAM` and `_DGRAM`)
`uipc_socket.c`	Socket support routines
`uipc_socket2.c`	More socket support routines
`uipc_syscalls.c`	Main socket API (socket, send, recv, etc.)
`uipc_usrreq.c`	User request support routines

This section details the implementation of sockets, picking up where user mode leaves off (that is, from the moment a socket-related system call is invoked).

Socket Descriptors

A socket, which to the user appears to be just another file descriptor, is a mammoth structure in kernel mode, containing the socket type, state data, and much more. This structure, the `struct socket`, is defined in `bsd/sys/socketvar.h`. It is obtained by a call to `file_socket()`, which (like other file descriptors) uses `fp_lookup()` (shown in Listing 15-17) to obtain the `fileproc` structure corresponding to the file descriptor. The `fileproc` structures belonging to sockets have their `f_type` set to `DTYPE_SOCKET`, and the `f_data` member is the `struct socket` pointer which the system call operated on.

The `struct socket` contains many fields, and has a messy declaration intermixed with inline structures and constants. The most important fields for our discussion are:

- **`so_proto`**: A pointer to the socket's protocol. Through this, the socket protocol, type, and domain can be determined.
- **`so_pcb`**: A pointer to the protocol control block. This is defined as a void pointer, because the underlying protocol can vary (`struct in6pcb` or `struct inpcb`).

An abbreviated form of the structure is shown in Listing 17-4:

LISTING 17-4: An abbreviated socket structure, from bsd/sys/socketvar.h

```
struct socket {
        int     so_zone;                /* zone we were allocated from */
        short   so_type;                /* generic type, see socket.h */
        short   so_options;             /* from socket call, see socket.h */
        short   so_linger;              /* time to linger while closing */
        short   so_state;               /* internal state flags SS_*, below */
        void    *so_pcb;                /* protocol control block */
        struct  protosw *so_proto;      /* protocol handle */
..
        struct  sockbuf {... } so_rcv,  /* Receive queue (incoming) */
                               so_snd;  /* Send queue    (outgoing) */
        //
        // ...  Many many more fields ..
        struct  label *so_label;        /* MAC label for socket */
        struct  label *so_peerlabel;    /* cached MAC label for socket peer */
        // ….
        // last process to interact with this socket
        u_int64_t       last_upid;
        pid_t           last_pid;

}
```

mbufs

Each socket maintains a `struct sockbuf`, which is used in maintaining its receive and send queues. The actual data sent and received in sockets, however, is maintained in "memory buffers", which are `struct mbuf` structures. These structures (similar to Linux's `sk_buffs`) are defined in `bsd/sys/mbuf.h`, but are normally left as opaque `mbuf_ts`, with the preferred method of dealing with them being the various accessors declared in `bsd/sys/kpi_mbuf.h`.

An `mbuf` is composed of a header and a body. The header is a `struct m_hdr` containing the buffer metadata, as well as a link to the next buffer, and a link to the next packet, if any. In this way, `mbufs` are chained, as shown in Figure 17-2.

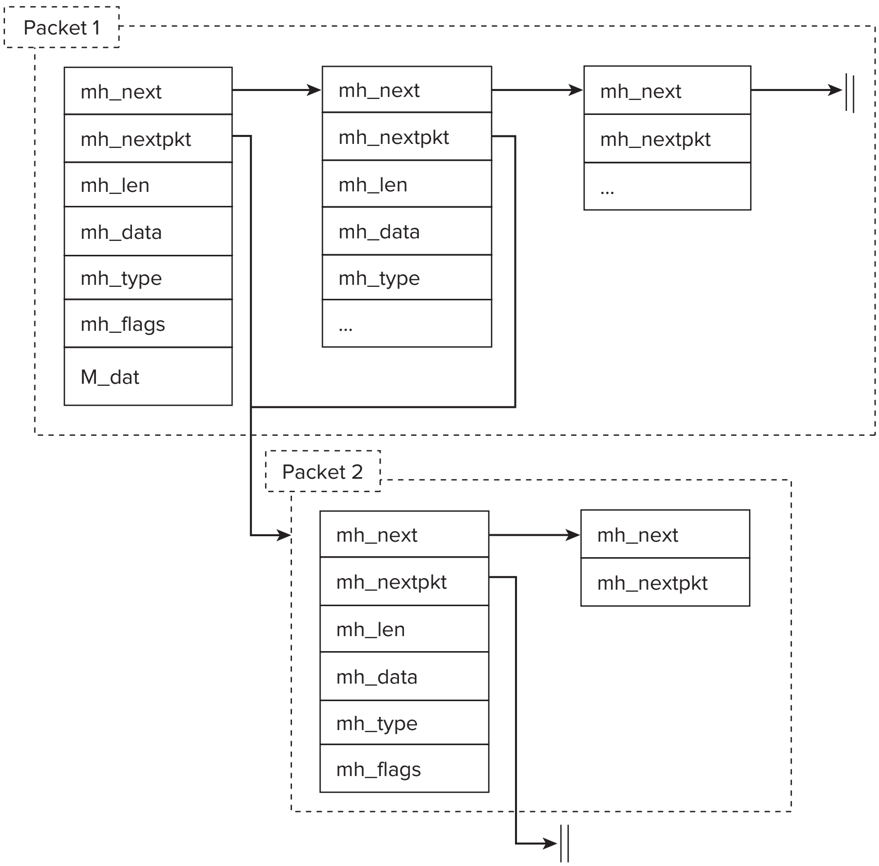


FIGURE 17-2: An `mbuf` chain

The `mbuf` header is defined in `bsd/sys/mbuf.h` as shown in Listing 17-5:

LISTING 17-5: The mbuf header

```
struct m_hdr {
        struct  mbuf *mh_next;          /* next buffer in chain        */
        struct  mbuf *mh_nextpkt;       /* next chain in queue/record  */
        int32_t mh_len;                 /* amount of data in this mbuf */
        caddr_t mh_data;                /* location of data            */
        short   mh_type;                /* type of data in this mbuf   */
        short   mh_flags;               /* flags; see below            */
}

struct mbuf {
        struct  m_hdr m_hdr;
        union {
                struct {
                        struct  pkthdr MH_pkthdr;       /* M_PKTHDR set */
```

```
                        union {
                                struct  m_ext MH_ext;   /* M_EXT set */
                                char    MH_databuf[_MHLEN];
                        } MH_dat;
                } MH;
                char    M_databuf[_MLEN];               /* !M_PKTHDR, !M_EXT */
        } M_dat;
};
```

Following the `m_hdr` is an `m_dat` union that — depending on the settings in `m_hdr.m_flags` — may hold one of three things, as shown in Table 17-5.

TABLE 17-5: Flags in an mbuf Header, and the Corresponding Contents of the mbuf

FLAG	DENOTES THAT WHAT FOLLOWS IS. . .
`M_PKTHDR`	The packet, split into the header in `m_dat.MH.MH_pkthdr`, and the payload — contiguously, in m_dat.`MH.MH_dat.MH_databuf`.
`M_EXT`	A pointer to the packet, stored externally in `m_dat.MH`.MH`_dat.MH_ext`. This is known as a *cluster*.
(No flag)	Packet data in m_dat.`M_databuf`. This is used for packet data spanning multiple `mbuf`s. The first `mbuf` will have `M_PKTHDR` set.

Using the functions in `bsd/sys/kpi_mbuf.h` header for allocating and handling `mbufs`, relieves the programmer from dealing with the header specifics. Functions such as `mbuf_allocpacket/mbuf_alloccluster` (used by drivers), and many accessors (e.g. `mbuf_data()`, `mbuf_setdata()`, etc.) all operate on an `mbuf_t`, which is effectively a void pointer. All of these functions are very well documented elsewhere. One function worthy of mentioning here, however, is `mbuf_tag_allocate`. With it, an `mbuf` can be assigned a 32-bit integer value, which is considered opaque by the kernel. A driver, however, may use the tag to hold external data, from bit flags, to a buffer ID. This is useful for tracking `mbuf` ownership. The `netstat(8)` command can be used to display `mbuf` utilization (using the `–m` switch), which it obtains using `sysctl(8)`.

Once the multiple domains have been registered, and each domain has its associated protocols and socket types, it becomes a simple matter to provide sockets of the supported types. Each socket has a pointer to its corresponding protocol, which is assigned during creation. The `socket(2)` system call is used to create sockets from user mode, as shown in Listing 17-6:

LISTING 17-6: The implementation of socket(2)

```
int socket(struct proc *p, struct socket_args *uap, int32_t *retval)
{
        struct socket *so;
        struct fileproc *fp;
        int fd, error;

        // call AUDIT_ARG to record call in audit subsytem
```

continues

LISTING 17-6 *(continued)*

```
        AUDIT_ARG(socket, uap->domain, uap->type, uap->protocol);

#if CONFIG_MACF_SOCKET_SUBSET
        // call on MAC subsystem to check if sockets are allowed (q.v. Chapter 13)
        if ((error = mac_socket_check_create(kauth_cred_get(), uap->domain,
            uap->type, uap->protocol)) != 0)
                return (error);
#endif /* MAC_SOCKET_SUBSET */
        // allocate file descriptor
        error = falloc(p, &fp, &fd, vfs_context_current());
        ...
        // Mark as a socket, read writable, with standard socket operations
        fp->f_flag = FREAD|FWRITE;
        fp->f_type = DTYPE_SOCKET;
        fp->f_ops = &socketops;

        // Create domain (family) and type/protocol specific socket
        error = socreate(uap->domain, &so, uap->type, uap->protocol);
        if (error) {
                fp_free(p, fd, fp);
        } else {
         ...
           /* if this is a backgrounded thread then throttle all new sockets */
              ...
                 // connect socket data
                fp->f_data = (caddr_t)so;

                proc_fdlock(p);
                procfdtbl_releasefd(p, fd, NULL);

                fp_drop(p, fd, fp, 1);
                proc_fdunlock(p);

                *retval = fd;
        }
        return (error);
}
```

The main work in the preceding code is performed by `socreate`, in `bsd/kern/uipc_socket.c`, shown as follows:

```
socreate(int dom, struct socket **aso, int type, int proto)
{
        struct proc *p = current_proc();
        register struct protosw *prp;
        register struct socket *so;
        register int error = 0;

        // ...

       // First find the protocol for this socket domain (family) and type.
       // If one is specified, look it up. Otherwise, get default
```

```
        if (proto)
                prp = pffindproto(dom, proto, type);
        else
                prp = pffindtype(dom, type);

        // Handle protocol lookup error, or protocol with no attach function
        if (prp == 0 || prp->pr_usrreqs->pru_attach == 0) {
                if (pffinddomain(dom) == NULL) {
                        return (EAFNOSUPPORT);
                }
                if (proto != 0) {
                        if (pffindprotonotype(dom, proto) != NULL) {
                                return (EPROTOTYPE);
                        }
                }
                return (EPROTONOSUPPORT);
        }

        if (prp->pr_type != type)
                return (EPROTOTYPE);

  // If we're still here, all is well. Go ahead and allocate socket
  // TCPv4 sockets are allocated from the Mach socache zone.
  // All other sockets are allocated from BSD's M_SOCKET zone.

  so = soalloc(1, dom, type);

        if (so == 0)
                return (ENOBUFS);

        TAILQ_INIT(&so->so_incomp);
        TAILQ_INIT(&so->so_comp);

         // Allocate various socket fields
        so->so_type = type;

        // Set ownership to uid/gid of current, and mark root owned as SS_PRIV
        so->so_uid = kauth_cred_getuid(kauth_cred_get());
        so->so_gid = kauth_cred_getgid(kauth_cred_get());
        if (!suser(kauth_cred_get(), NULL))
                so->so_state = SS_PRIV;

        // This line is responsible for making everything work:
        so->so_proto = prp;            // Link the protocol

#ifdef __APPLE__
        so->so_rcv.sb_flags |= SB_RECV; /* XXX */
        so->so_rcv.sb_so = so->so_snd.sb_so = so;
#endif
        so->next_lock_lr = 0;
        so->next_unlock_lr = 0;

#if CONFIG_MACF_SOCKET
        // If BSD's MAC layer is configured for sockets, associate this
       // socket with a label
```

continues

LISTING 17-6 *(continued)*

```
        mac_socket_label_associate(kauth_cred_get(), so);
#endif /* MAC_SOCKET */

//### Attachement will create the per pcb lock if necessary and increase refcount
        /*
         * for creation, make sure it's done before
         * socket is inserted in lists
         */
        so->so_usecount++;


        error = (*prp->pr_usrreqs->pru_attach)(so, proto, p);
        if (error) {
              // abort: decrease so_usecount and free socket,
        }
#ifdef __APPLE__

       // Increase reference to this domain (address family)

        prp->pr_domain->dom_refs++;
        TAILQ_INIT(&so->so_evlist);

        /* Attach socket filters for this protocol */
        sflt_initsock(so);
#if TCPDEBUG
        if (tcpconsdebug == 2)
                so->so_options |= SO_DEBUG;
#endif
#endif
        so_set_default_traffic_class(so);
        /*
         * If this is a background thread/task, mark the socket as such.
         */
#if !CONFIG_EMBEDDED
        if (proc_get_self_isbackground() != 0)
#else /* !CONFIG_EMBEDDED */
        thread = current_thread();
        ut = get_bsdthread_info(thread);
        if (uthread_get_background_state(ut))
#endif /* !CONFIG_EMBEDDED */
{
                socket_set_traffic_mgt_flags(so, TRAFFIC_MGT_SO_BACKGROUND);
                so->so_background_thread = current_thread();
        }

       // special handling of AF_LOCAL sockets and workaround for IPv6
       // socket cases follows here..
       // ...

       // return newly created socket as our out parameter, and report success
```

```
        // The so returned will be latched on to the file descriptor
         *aso = so;
         return (0);
}
```

The socket structure is attached to the corresponding file descriptor's `fp_data` field. The protocol operations are themselves a pointer from the socket structure's `so_proto`. Thus, socket-related system calls basically retrieve the socket from the file pointer and perform some housekeeping, with the bulk of the work done by the corresponding `pr_usrreqs` entry for the top-level call.

Sockets in Kernel Mode

As surprising as it sounds, creating a socket in kernel mode is not as straightforward as it should be. A socket normally needs to be mapped to a file descriptor, and failure to properly maintain the relationship can cause the process to crash, or even the entire kernel to panic.

To work with sockets in kernel mode, XNU offers the `kpi_socket` interface. This is a set of `sock_*` functions whose functionality emulates, or in some cases extends, that of user mode (see Table 17-6). This interface enables the creation and manipulation of sockets in kernel mode, similar to the "Winsock Kernel" concept in Windows (Vista or later). This can prove useful for a kernel extension that needs to communicate with a remote server.

TABLE 17-6: KPI Socket Interface Calls, from bsd/kern/kpi_socket.c

KPI SOCKET FUNCTION	IN USER MODE	USED FOR
`errno_t` **`sock_socket`** `(int` *`domain`*`,` `int` *`type`*`,` `int` *`protocol`*`,` `sock_upcall` *`callback`*`,` `void *`*`cookie`*`,` `socket_t *`*`new_so`*`);`	`int` **`socket`** `(int domain,` `int type,` `int protocol)`	Same as `socket`, but allows setting a *`callback`* function that will be invoked on `socket` events with the *`cookie`* parameter. `Socket` is returned in *`new_so`*.
`sock_accept``(socket_t` *`sock`*`,` `struct sockaddr *`*`from`*`,` `int` *`fromlen`*`,` `int` *`flags`*`,` `sock_upcall` *`callback`*`,` `void*` *`cookie`*`,` `socket_t *`*`new_sock`*`)`	`int` **`accept`** `(int socket,` `struct sockaddr * addr,` `socklen_t *addrlen);`	Accepts a connection on *`sock`*, returning a `new_sock`. Optionally, set *`callback`* and the argument *`cookie`* to be used on new `socket` events.

continues

TABLE 17-6 *(continued)*

KPI SOCKET FUNCTION	IN USER MODE	USED FOR
`errno_t` **`sock_bind`** `(socket_t sock, const struct sockaddr *to);`	`int` **`bind`**`(int socket, struct sockaddr *addr, socklen_t addrlen);`	Binds the `sock` to the address specified in *`to`*. The usual type-casting of specific `sockaddr` subtypes applies.
`errno_ t` **`sock_gettype`** `(socket_t so, int *domain, int *type, int *protocol);`	---	Gets the domain, type, and protocol used in a `socket(2)` or `sock_socket` call. Any of the parameters may be left `NULL`.
`int` **`sock_isconnected`** `(socket_t so);`	---	Returns non-zero if `socket` is connected (`SS_ISCONNECTED`).
`int` **`sock_isnonblocking`** `(socket_t so);`	---	Returns non-zero if socket is nonblocking (`SS_NBIO`).
`errno_t` **`sock_setpriv`** `(socket_t so, int on);`	--	Toggles the `SS_PRIV` flag on the socket in question.
`errno_t` **`sock_setupcall`** `(socket_t sock, sock_upcall callback, void* context);`	--	Sets or unsets an event `callback` (“`upcall`”) function.

Nonblocking sockets in the kernel make use of callbacks, or what KPI calls “upcall” functions. These functions accept three arguments — the socket, a “cookie” (a void pointer opaque argument), and a boolean specifying whether blocking in the function is allowed. When creating a socket (with `sock_socket`) or accepting (`sock_accept`), the caller may set the callback with different cookie arguments for each socket, allowing the same upcall to be used in handling multiple sockets. An upcall may be set or unset at any other time using `sock_setupcall` (specifying `NULL` removes the upcall function).

Layer IV: Transport Protocols

The TCP/IP-related protocols are implemented in a separate directory — `bsd/netinet` for IPv4, and `bsd/netinet6` for IPv6. Each layer III protocol can define its own layer IV ones, as IPv4 does in its `struct inetsw` array, (`bsd/netinet/in_proto.c`) and IPv6 in its `struct inet6sw` (`bsd/netinet6/in6_proto.c`).

The protocols in Table 17-7 are supported (note that ICMP and RAW are not transport protocols in the classic sense of the word, but are still defined with the same structure type).

TABLE 17-7: Supported Transport Protocols

PROTOCOL	STRUCT PR_USRREQS	DECLARED IN
ICMPv4	`icmp_dgram_usrreqs`	`bsd/netinet/ip_icmp.c`
ICMPv6	`icmp6_dgram_usrreqs`	`bsd/netinet6/raw_ip6.c`
TCPv4	`tcp_usrreqs`	`bsd/netinet/tcp_usrreq.c`
TCPv6	`tcp6_usrreqs`	`bsd/netinet/tcp_usrreq.c`
RAW (v4)	`rip_usrreqs`	`bsd/netinet/raw_ip.c`
RAW (v6)	`rip6_usrreqs`	`bsd/netinet6/raw_ip6.c`
UDPv4	`udp_usrreqs`	`bsd/netinet/udp_usrreq.c`
UDPv6	`udp6_usrreqs`	`bsd/netinet6/udp6_usrreq.c`

The `pr_usrreqs` contain the implementation of each protocol's "user requests," which correspond to user mode socket API calls (such as `send`, `recv`), discussed later in this chapter. Additional protocols, such as IPSec ones (AH/ESP), are supported but have no `usrreqs` of their own.

Domains and Protosws

The multiple address families supported by the kernel are referred to as *domains* (totally unrelated to the domains of DNS) and are maintained in a global domains list. This list, appropriately called `domains`, is a linked list of `struct domain`, defined in `bsd/sys/domain.h` as shown in Listing 17-7:

LISTING 17-7: The domain structure, from bsd/sys/domain.h

```
struct  domain {
  int     dom_family;             /* AF_xxx */
  const char *dom_name;
  void   (*dom_init)(void);                  // initialize domain structures
  int    (*dom_externalize)(struct mbuf *); /* externalize access rights */
  void   (*dom_dispose)(struct mbuf *);     /* dispose of internalized rights */
  struct  protosw *dom_protosw;              /* Chain of protosw's for AF   */
  struct  domain *dom_next;
  int    (*dom_rtattach)(void **, int);     /* initialize routing table    */
  int     dom_rtoffset;                    /* an arg to rtattach, in bits */
  int     dom_maxrtkey;                    /* for routing layer */
  int     dom_protohdrlen;                 /* Let the protocol tell us */
  int     dom_refs;                        /* # socreates outstanding */
#ifdef _KERN_LOCKS_H_
lck_mtx_t *dom_mtx;                        /* domain global mutex */
#else
  void    *dom_mtx;                        /* domain global mutex */
#endif
  uint32_t   dom_flags;
  uint32_t   reserved[2];
};
```

Because it's a global structure, access to the domains list is protected by a `domain_proto_mtx` `mutex`. Each domain also points to an array of one or more protocol structures that are associated with the domain. The same `mutex` also protects access to these protocols. (See Listing 17-8.)

LISTING 17-8: The protosw structure, from bsd/sys/protosw.h

```
struct protosw {
        short   pr_type;                /* socket type used for */
        struct  domain *pr_domain;      /* domain protocol a member of */
        short   pr_protocol;            /* protocol number */
        unsigned int pr_flags;          /* see below */
/* protocol-protocol hooks */
        void    (*pr_input)(struct mbuf *, int len);
                                        /* input to protocol (from below) */
        int     (*pr_output)(struct mbuf *m, struct socket *so);
                                        /* output to protocol (from above) */
        void    (*pr_ctlinput)(int, struct sockaddr *, void *);
                                        /* control input (from below) */
        int     (*pr_ctloutput)(struct socket *, struct sockopt *);
                                        /* control output (from above) */
/* user-protocol hook */
        void    *pr_ousrreq;            // deprecated
/* utility hooks */
        void    (*pr_init)(void);       /* initialization hook */
#if __APPLE__
        void    (*pr_unused)(void);     /* placeholder - fasttimo is removed */
#else
        void    (*pr_fasttimo)(void);
                                        /* fast timeout (200ms) */
#endif
        void    (*pr_slowtimo)(void);
                                        /* slow timeout (500ms) */
        void    (*pr_drain)(void);
                                        /* flush any excess space possible */
#if __APPLE__
        int     (*pr_sysctl)(int *, u_int, void *, size_t *, void *, size_t);
                                        /* sysctl for protocol */
#endif
        struct  pr_usrreqs *pr_usrreqs; /* supersedes pr_usrreq() */
#if __APPLE__
     int   (*pr_lock)(struct socket *so, int locktype, void *debug); /* lock function */
     int   (*pr_unlock)(struct socket *so, int locktype, void *debug); /* unlock  */
#ifdef _KERN_LOCKS_H_
        lck_mtx_t *     (*pr_getlock)   (struct socket *so, int locktype);
#else
        void *  (*pr_getlock)   (struct socket *so, int locktype);
#endif
#endif
#if __APPLE__
```

```
/* Implant hooks */
        TAILQ_HEAD(, socket_filter) pr_filter_head;
        struct protosw *pr_next;        /* Chain for domain */
        u_int32_t       reserved[1];            /* Padding for future use */
#endif
};
```

The fields in this structure are basically of two types:

- **Protocol requests:** These requests are internal to the protocol and inaccessible from user space. They are used by the networking stack itself to handle various protocol events (see Table 17-8).

TABLE 17-8: Protocol Requests

FUNCTION	USED FOR
`pr_input (struct mbuf *m, int len);`	Ingress traffic from network device. Passes a chain of buffers, *m*, of len `len`. Performs protocol decapsulation and finds socket
`pr_output(struct mbuf *m, struct socket *so);`	Egress traffic. Mostly `NULL`.
`pr_ctlinput (int, struct sockaddr *, void *)`	Protocol commands, PRC_* constants from `bsd/sys/protosw.h`, corresponding to ICMP and network events
`pr_ctloutput (struct socket *, struct sockopt *);`	Implementing `setsockopt(2)`
`void pr_init(void)`	Protocol initialization function. This is called when the protocol is first added — for static protocols, by `domain_init()`, and for dynamically added ones, by `init_proto()` — from `net_add_proto()`. After initialization, this point is set to `NULL` to avoid re-calling.
`void pr_fasttimo ();` `void pr_slowtimo();`	Deprecated. Unused (`NULL` in all protocols). Fast timeout originally used for 200ms timeout, Slow timeout used for 500ms.
`void pr_drain();`	Drain (discard) excess protocol data when system is low on space

continues

TABLE 17-8 *(continued)*

FUNCTION	USED FOR
`void pr_sysctl((int *, u_int, void *, size_t *, void *, size_t);`	An extension over the BSD model to support `sysctl(8)` over the various protocols.
`void pr_lock(struct socket *so, int locktype, void *debug);` `int pr_unlock(struct socket *so, int locktype, void *debug);`	An extension over the BSD model used to enable a lock of `locktype` over the protocol.

➤ **User requests:** These are the various system call implementations of the socket API for the socket of the specified protocol. Originally, a single function, `pr_usrreq()`, was used in an `ioctl()`-like manner for all user requests, with the request specified in a `PRU_` constant. This function has been deprecated (renamed to `pr_ousrreq()` and left unused) and replaced by the `pr_usrreqs` pointer. This is a pointer to a massive structure on its own, a `struct pr_usrreqs`, containing the protocol-specific implementation of functions, or `NULL` for functions that are not applicable for this protocol. The structure is defined and somewhat amusingly commented in `bsd/sys/protosw.h`, as shown in Listing 17-9:

LISTING 17-9: The struct pr_usrreqs definition in bsd/sys/protosw.h

```
/*
 * If the ordering here looks odd, that's because it's alphabetical.
 * Having this structure separated out from the main protoswitch is allegedly
 * a big (12 cycles per call) lose on high-end CPUs.  We will eventually
 * migrate this stuff back into the main structure.
 */
struct pr_usrreqs {
        int     (*pru_abort)(struct socket *so);
        int     (*pru_accept)(struct socket *so, struct sockaddr **nam);
        int     (*pru_attach)(struct socket *so, int proto, struct proc *p);
```

```
        int     (*pru_bind)(struct socket *so, struct sockaddr *nam,
                                 struct proc *p);
        int     (*pru_connect)(struct socket *so, struct sockaddr *nam,
                                    struct proc *p);
        int     (*pru_connect2)(struct socket *so1, struct socket *so2);
        int     (*pru_control)(struct socket *so, u_long cmd, caddr_t data,
                                    struct ifnet *ifp, struct proc *p);
        int     (*pru_detach)(struct socket *so);
        int     (*pru_disconnect)(struct socket *so);
        int     (*pru_listen)(struct socket *so, struct proc *p);
        int     (*pru_peeraddr)(struct socket *so, struct sockaddr **nam);
        int     (*pru_rcvd)(struct socket *so, int flags);
        int     (*pru_rcvoob)(struct socket *so, struct mbuf *m, int flags);
        int     (*pru_send)(struct socket *so, int flags, struct mbuf *m,
                                 struct sockaddr *addr, struct mbuf *control,
                                 struct proc *p);
#define PRUS_OOB        0x1
#define PRUS_EOF        0x2
#define PRUS_MORETOCOME 0x4
        int     (*pru_sense)(struct socket void  *sb, int isstat64);
        int     (*pru_shutdown)(struct socket *so);
        int     (*pru_sockaddr)(struct socket *so, struct sockaddr **nam);

        /*
         * These three added later, so they are out of order.  They are used
         * for shortcutting (fast path input/output) in some protocols.
         * XXX - that's a lie, they are not implemented yet
         * Rather than calling sosend() etc. directly, calls are made
         * through these entry points.  For protocols which still use
         * the generic code, these just point to those routines.
         */
        int     (*pru_sosend)(struct socket *so, struct sockaddr *addr,
                                   struct uio *uio, struct mbuf *top,
                                   struct mbuf *control, int flags);
        int     (*pru_soreceive)(struct socket *so,
                                      struct sockaddr **paddr,
                                      struct uio *uio, struct mbuf **mp0,
                                      struct mbuf **controlp, int *flagsp);
        int     (*pru_sopoll)(struct socket *so, int events,
                                   struct ucred *cred, void *);
};
```

Initializing Domains

During kernel initialization, `domaininit()`, in `bsd/kern/uipc_domain.c`, is called from `bsd_init` and is responsible for initializing all the domains from Table 17-1. All these domains (with the exception of PPP) are hard-coded into the kernel. `domaininit()` adds them by concatenating (before Lion) or prepending (Lion) them, in turn, to the `domains` list. For each domain, if a `dom_init` function exists, it is called. Likewise, for each domain protocol, `init_proto()`, is called. This function calls the protocol's `pr_init` function, if set, then unsets it (to prevent additional calls by accident). Domains and protocols can also be modified dynamically (for example, as PPP is, from the PPP kernel extension), as shown in Table 17-9. Protocol-related functions are defined in `bsd/sys/protosw.h` and domain-related ones in `domain.h`. All are implemented in `bsd/kern/uipc_domain.c`.

TABLE 17-9: Domain and Protocol Dynamic Manipulation Functions

FUNCTION	USAGE
`net_add_domain (struct domain *dp);`	Prepends domain *dp* to the global `domains` list and calls `init_domain()` to invoke the domain's `dom_init()`, if any.
`struct domain *pffinddomain (int pf);`	Looks up a domain whose `dom_family` matches *pf*.
`net_del_domain(struct domain *dp);`	Unlinks domain *dp* from the `domains` list.
`int net_add_proto(struct protosw *pp, struct domain *dp);`	Adds the protocol specified by *pp* to the domain *dp*, and calls `init_proto()` to invoke the protocol's `pr_init` (unsetting it after use).
`struct protosw *pffindtype (int family, int type);`	Looks up a protocol in the domain matching *family* whose `pr_type` matches *type*.
`Int net_del_proto(int type, int protocol, struct domain *dp);`	Removes protocol whose `pr_type` and `pr_protocol` fields match, in domain *dp*.

Conceptually, the resulting representation of domains is simple, though large (see Figure 17-3). The domain points to an array of `protosw` structures, which in turn point to various functions.

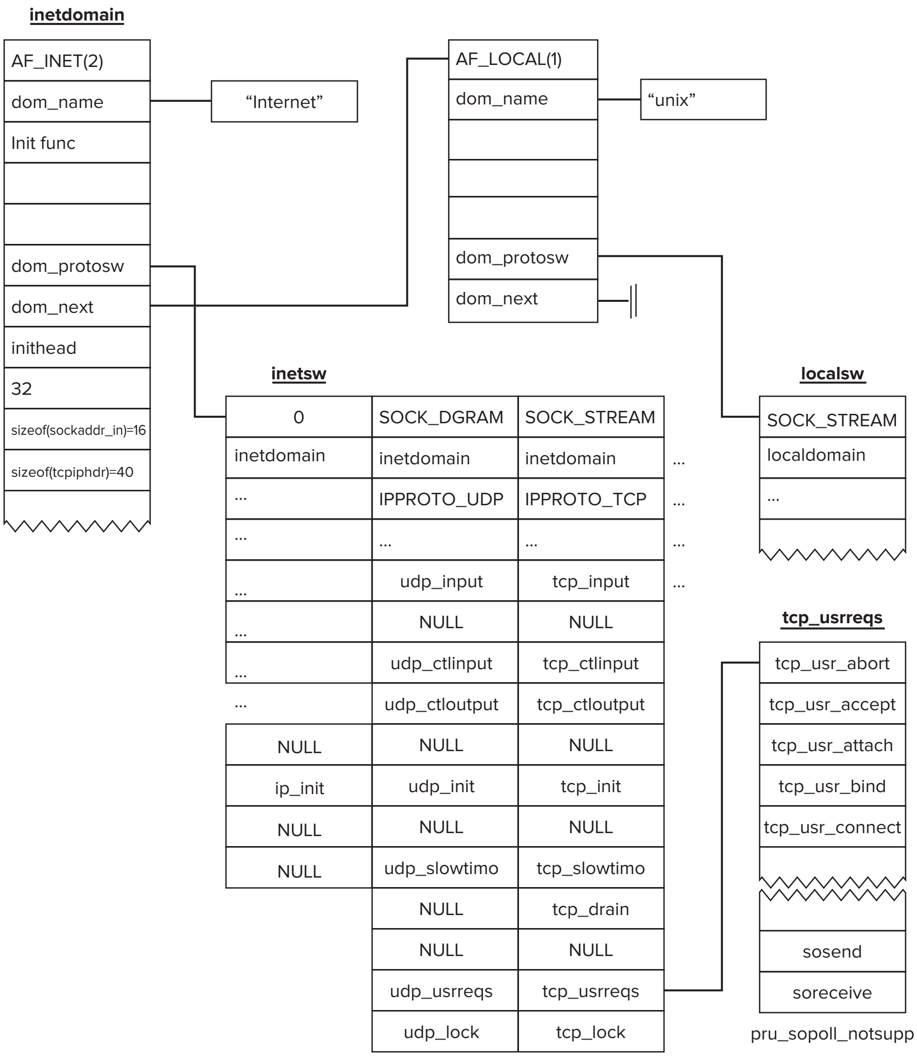


FIGURE 17-3: XNU's domain structures

LAYER III: NETWORK PROTOCOLS

Layer III (network level) protocols are somewhat simpler than their transport level counterparts. These protocols can be registered dynamically, although XNU currently only supports IPv4, IPv6, and AppleTalk. Network protocols may be registered with `proto_register_input()`, which initializes a `struct proto_input_entry` and inserts it into a private `proto_hash` hash table. The hash function used in this case is crude: `proto_hash_value()` simply returns hard coded numbers (0 through 3) for each of the four protocols it recognizes, and a different number (4) for all other protocols.

A layer III protocol is implemented as a `proto_input_entry` defined in `bsd/net/kpi_protocol.c` as shown in Listing 17-10:

LISTING 17-10: struct proto_input_entry in bsd/net/kpi_protocol.c

```
struct proto_input_entry {
        struct proto_input_entry           *next;
        int                                 detach;
        struct domain                      *domain;
        int                                 hash;
        int                                 chain;

        protocol_family_t                   protocol;
        proto_input_handler                 input;
        proto_input_detached_handler        detached;

        mbuf_t                              inject_first;
        mbuf_t                              inject_last;

        struct proto_input_entry           *input_next;
        mbuf_t                              input_first;
        mbuf_t                              input_last;
};
```

You may have noticed that there is no output function in Listing 17-9. This is because the output functions of the layer III protocols are actually called directly by those of layer IV. Although the `ip_output_list()` function (for IPv4) and `ip6_output` (for IPv6) have similar prototypes, they are overall different, and are called by name from TCP, UDP, and RAW's output functions, rather than by pointer. Listing 17-11 shows the prototypes of the IP and IPv6 output functions:

LISTING 17-11: The ip6_output and ip_output_list prototypes in XNU

```
morpheus@ergo (../xnu/1699.26.8/)$ ./findfunc.sh ip6_output ip_output_list
./bsd/netinet6/ip6_output.c:232:ip6_output( struct mbuf *m0, struct ip6_pktopts *opt,
struct  route_in6 *ro, int flags, struct ip6_moptions *im6o, struct ifnet **ifpp,
struct ip6_out_args
 *ip6oa);
./bsd/netinet/ip_output.c:265:ip_output_list( struct mbuf *m0, int packetchain, struct
 mbuf *opt, struct route *ro, int flags, struct ip_moptions *imo, struct
 ip_out_args *ipoa );
```

Note, that while this is a deviation from the neatness of the OSI model (in that the transport has to know its network), this is not a fault of XNU's or BSD's, but of the IP model itself: UDP, for example, includes headers fields from IP (the so called "pseudo-header") in its checksum calculation.

The `bsd/net/kpi_protocol.h` header file defines and documents the KPI interfaces available for manipulating and implementing protocols. Overall, the following functions in Listing 17-12 are defined:

LISTING 17-12: Protocol KPI functions

```
typedef void (*proto_input_handler)(protocol_family_t protocol, mbuf_t packet);
typedef void (*proto_input_detached_handler)(protocol_family_t protocol);

// Input handler registration functions
errno_t proto_register_input(protocol_family_t protocol,
         proto_input_handler input, proto_input_detached_handler detached,
         int chains);
void    proto_unregister_input(protocol_family_t protocol);
errno_t proto_input(protocol_family_t protocol, mbuf_t packet);
errno_t proto_inject(protocol_family_t protocol, mbuf_t packet);

// Plumbing and unplumbing handlers for attaching protocols to interfaces
typedef errno_t (*proto_plumb_handler)(ifnet_t ifp, protocol_family_t protocol);
typedef void (*proto_unplumb_handler)(ifnet_t ifp, protocol_family_t protocol);

// registration functions for above
errno_t proto_register_plumber(protocol_family_t proto_fam, ifnet_family_t if_fam,
        proto_plumb_handler plumb, proto_unplumb_handler unplumb);
extern void proto_unregister_plumber(protocol_family_t proto_fam,ifnet_family_t if_fam);

// functions for plumbing
errno_t proto_plumb(protocol_family_t protocol_family, ifnet_t ifp);
errno_t proto_unplumb(protocol_family_t protocol_family, ifnet_t ifp);
```

Attaching Protocols to Interfaces

To enable a network protocol, it must be attached to one or more network interfaces. These are maintained in the kernel as `struct ifnet` types (discussed in the next section). The operation of attaching a protocol to an interface is called *plumbing*, and the two functions available, `proto_plumb()` and `proto_unplumb()` (declared in `bsd/net/kpi_protocol.h`) are used for this purpose on `PF_INET` and `PF_INET6`. The interface provides a plumber from its end, which is called when the protocol is plumbed, and ties the interfaces's input and output functions to those of the protocol.

As an example, consider the loopback interface (`bsd/net/if_loop.c`). The `lo_reg_if_mods` function (called at the very beginning of `loopattach()`) registers the `lo_attach_proto()` function for both `AF_INET` and `AF_INET6`. As is the case with all plumbers, the function receives the `protocol_family` plumbed as one of its parameters. This is shown in Listing 17-13:

LISTING 17-13: lo_attach_proto() from bsd/net/if_loop.c

```
static errno_t  lo_attach_proto(ifnet_t ifp, protocol_family_t protocol_family)
{
        struct ifnet_attach_proto_param_v2      proto;
        errno_t                                                 result = 0;

        bzero(&proto, sizeof(proto));
        proto.input = lo_input;            // Calls ifnet's proto_input()
        proto.pre_output = lo_pre_output;  // Sets protocol type before output

        result = ifnet_attach_protocol_v2(ifp, protocol_family, &proto);

        if (result && result != EEXIST) {
                printf("lo_attach_proto: ifnet_attach_protocol for %u returned=%d\n",
                           protocol_family, result);
        }

        return result;
}
```

LAYER II: INTERFACES

At the lowest layer, UN*X defines the *interface*. Interfaces are devices, but unlike character or block devices, they have no `/dev` representation, and can only be accessed through sockets. User mode applications can send and receive data through interfaces via sockets, or configure interfaces using `ioctl(2)` calls. An administrator can make use of the `ifconfig(8)` command (which itself uses `ioctl(2)` calls) for various configuration tasks.

Interfaces in OS X and iOS

XNU supports the interfaces shown in Table 17-10 natively:

TABLE 17-10: Interfaces Natively Supported by XNU

NAME	DEFINED IN	TYPE
bond	`bsd/net/if_bond.c`	Bonding two or more interfaces
bridge	`bsd/net/if_bridge.c`	Layer II bridging (new in Lion)
gif	`bsd/net/if_gif.c`	Generic IP-in-IP tunneling (RFC2893)
lo	`bsd/net/if_loop.c`	Loopback interface
pflog	`bsd/net/if_pflog.c`	Packet filtering (new in Lion): receives copies of all packets logged by PF.
stf	`bsd/net/if_stf.c`	6to4 (RFC3056) connectivity. Discussed previously in this chapter, under "IPv6 Networking."

NAME	DEFINED IN	TYPE
utun	`bsd/net/if_utun.c`	User tunnels: used by VPN and other processes to provide a pseudo interface, whose traffic will be rerouted through a user-mode process.
vlan	`bsd/net/if_vlan.c`	Virtual Local Area Networks

Note that not all interfaces are necessarily active and present on any given system. The `lo` is the only interface which is strictly necessary, and is always present (created by a call to `loopattach()` from `bsd_init`, as discussed in Chapter 8). If you have astutely noticed no mention of any "en" interfaces (used for Ethernet and 802.11), it's not that they were forgotten; they are just not natively registered. Even though support for the basic Ethernet logic is built-in to XNU, the kernel still relies on external kexts to create physical interfaces. Table 17-11 shows those kexts known to create such interfaces.

TABLE 17-11: Interfaces Owned by Kernel Extensions

NAME	OWNING KEXT/FAMILY	TYPE
`en`	IONetworkingFamily	Ethernet or 802.11 interfaces
`fw`	IOFireWireIP	IP over FireWire (IEEE-1394). OS X only
`pdp_ip`	AppleBaseBandFamily	Cellular data connection (iPhone, iPad 1/2)
`ppp`	PPP	Point-to-Point protocol (pppd)

Aside from the loopback interface, XNU supports quite a few interfaces natively, but note they are all virtual, or pseudo-interfaces. The `gif(4)` and `stf(4)` interfaces are enabled along with IPv6. The poorly documented `utun` interface can be enabled through a `PF_SYSTEM` socket by tunneling utilities. The bond, bridge, and vlan interfaces are usually created manually by a system administrator using `ifconfig(8)`'s create sub command, as is `pflog(4)`.

Experiment: Manually Creating Interfaces Using ifconfig(8)

For example, consider Output 17-4, which demonstrates the ease with which a bridge interface can be created as of Lion:

OUTPUT 17-4: A short lived bridge, erecting using ifconfig create

```
root@Minion (/)# ifconfig bridge0          # check existence
ifconfig: interface bridge0 does not exist
root@Minion (/)# ifconfig bridge0 create   # Lion and later - create bridge dynamically
root@Minion (/)# ifconfig bridge0
bridge0: flags=8822<BROADCAST,SMART,SIMPLEX,MULTICAST> mtu 1500
 ether ac:de:48:32:5f:a3
 Configuration:
  priority 32768 hellotime 2 fwddelay 15 maxage 20
  ipfilter disabled flags 0x2
 Address cache (max cache: 100, timeout: 1200):
root@Minion (/)# ifconfig bridge0 destroy        # easy come, easy go
```

The same method can be used to create the `vlan0` and `bond0` interfaces, which will display different attributes, and the `pflog0` interface (on Lion and later), which can be used to replicate any logged packets.

The Data Link Interface Layer

XNU contains generic code to handle the various interfaces, irrespective of their actual implementation. This generic code is collectively known as the Data Link Interface Layer (DLIL), and is largely self-contained in `bsd/net/dlil.c` (and exported via `dlil.h`).

The DLIL code maintains interface independence by treating all interface types as one abstract type: the `struct ifnet`. `dlil` provides various maintenance functions for interfaces (read: `ifnet` instances), but does not do any of the actual frame sending and receiving. Specific device drivers are expected to use the `ifnet` and `dlil` functions to maintain and export their interfaces, and set callbacks, which `dlil` can invoke at various stages of the frame's lifetime.

The ifnet Structure

Somewhat similar to Linux's `netdev`, BSD offers the `ifnet` structure to represent and manage network interfaces. OS X uses the same general structure, but with some modifications. The structure is (yet) another one of the massive structures, containing many statistics. Apple's `ifnet` is somewhat different from BSD's. An abbreviated and annotated version of this structure is presented in Listing 17-14:

LISTING 17-14: struct ifnet (abridged) from bsd/net/if_var.h

```
/*
 * Structure defining a network interface.
 *
 * (Would like to call this struct ``if'', but C isn't PL/1.) // and luckily so!
 *
 */
struct ifnet {
    ...
   void             *if_softc;      /* pointer to driver state */
   const char       *if_name;       /* name, e.g. ``en'' or ``lo'' */
   TAILQ_ENTRY(ifnet) if_link;      /* all struct ifnets are chained */
        ...
   struct ifaddrhead if_addrhead;  /* linked list of addresses per if */
   struct ifaddr    *if_lladdr;    /* link address (first/permanent) */
   int               if_pcount;    /* number of promiscuous listeners */
   struct bpf_if    *if_bpf;       /* packet filter structure */
                                   // ties BPF to ifnet
   u_short           if_index;     // sprintf()ed with if_name(%s%d),form instance name
   short             if_unit;      /* sub-unit for lower level driver */
   short             if_timer;     /* time 'til if_watchdog called */
   short             if_flags;     /* up/down, broadcast, etc. */
   u_int32_t         if_eflags;    /* see <net/if.h> */
```

```
    int                 if_capabilities;/* interface features & capabilities */
    int                 if_capenable;   /* enabled features & capabilities */

    // ...MIB and internal if data

    ifnet_family_t          if_family;       /* value assigned by Apple */
    uintptr_t               if_family_cookie;
    // Interface handling functions. Note, unlike BSD, no if_input() handler
    ifnet_output_func       if_output;    // called to send frame through interface
    ifnet_ioctl_func        if_ioctl;     // set ioctl on interface
    ifnet_set_bpf_tap       if_set_bpf_tap;// Required for BPF support (see later)
    ifnet_detached_func     if_free;      //
    ifnet_demux_func        if_demux;     // Demux layer III protocol from incoming frame
    ifnet_event_func        if_event;     // Miscellaneous event handler
    ifnet_framer_func       if_framer;    // Build layer II frame for outgoing frame
    ifnet_add_proto_func    if_add_proto; // Add a layer III protocol binding
    ifnet_del_proto_func    if_del_proto; // Remove a layer III protocol binding
    ifnet_check_multi       if_check_multi;// Approve multicast address for interface
    struct proto_hash_entry *if_proto_hash;// link to bound layer III protocol hash
    void                   *if_kpi_storage;// reserved for NKEs

    // busy state and number of waiters ...
    struct ifnet_filter_head if_flt_head;  // list of interface filters (described later)
    // ... Multicast address tables and parameters

    // Unlike BSD, every interface has its own dedicated input thread (hence no if_input)
     struct dlil_threading_info *if_input_thread;

    // broadcast support

    #if CONFIG_MACF_NET
    struct label            *if_label;      /* interface MAC label */
    #endif
    u_int32_t               if_wake_properties;
    #if PF
    struct thread           *if_pf_curthread;
    struct pfi_kif          *if_pf_kif;
    #endif /* PF */

    // cached source and forward route entries

    // link layer reachability tree and bridge glues

    // flags, route reference count, if_traffic_class (QoS)

    // Extensions for IGMPv3 (IPv4) and MLDv2 (IPv6)
};
```

The `ifnet` structures can be manipulated with several KPI functions, as shown in Table 17-12. Like many other KPIs, they all return `errno_t`.

TABLE 17-12: The KPI Functions Used to Handle Interfaces

FUNCTION	USAGE
ifnet_allocate (const struct *ifnet_init_params* *init, ifnet_t *interface);	Calls dlil_if_acquire() to create an ifnet, and initializes the ifnet fields which are not deemed kernel internal only (and specified in *init*). These are most of those shown in Listing 17-11. The function also ensures uniqueness of the interface instance, and initializes its reference count
ifnet_attach(ifnet_t *interface*, const struct sockaddr_dl *ll_addr); ifnet_detach(ifnet_t interface);	Makes *interface* visible by attaching it to global interface list (and tying its if_link field). Should only be called on a previously allocated interface. Similarly, detach it.
ifnet_reference(ifnet_t *interface*); **ifnet_release**(ifnet_t *interface*);	Increase or decrease the *interface's* reference count, free if count reaches 0. Because the ifnet_allocate() function already sets the reference count to 1, ifnet_release is effectively its inverse.
ifnet_attach_protocol[_v2] (ifnet_t *interface*, protocol_family_t *protocol_family*, const struct **ifnet_attach_proto_param[_v2]** **proto_details*);	Used by the interface when plumbing (attaching) a transport layer protocol. The ifnet_attach_proto_param structure contains callbacks for input and pre_output (required), as well as ioctl and ARP support. The [v2] variant allows for input functions which process packet lists, rather than individual packets.

In addition to the functions in the table, helper functions (like ifnet_find_by_name()), and quite a few accessor functions (all taking the struct ifnet * and returning its respective fields) can and should be used, to manipulate the individual ifnet fields rather than accessing them directly. A good example of the APIs in action can be found in the sources of IONetworkingFamily, the parent class of all networking kexts, wherein these APIs are used (in super methods which are later inherited by specific drivers).

Case Study: utun

OS X supports a special class of interfaces, called utuns. These are not real interfaces, or even kernel-based virtual ones. Rather, they are merely stubs, appearing to the user mode as interfaces, but in actuality redirecting their traffic through a specialized user mode process. Any packets sent through the interface are rerouted to the user mode process, and the same user mode process can instruct the interface to emit a packet.

The user mode processes usually use this mechanism for VPNs and other forms of tunneling, hence the name — User TUNnels. Packets arriving at the process are usually encapsulated and sent through a real network interface. Likewise, replies to those packets can be decapsulated and made to appear as originating from the utun interface. The send path is shown in Figure 17-4.

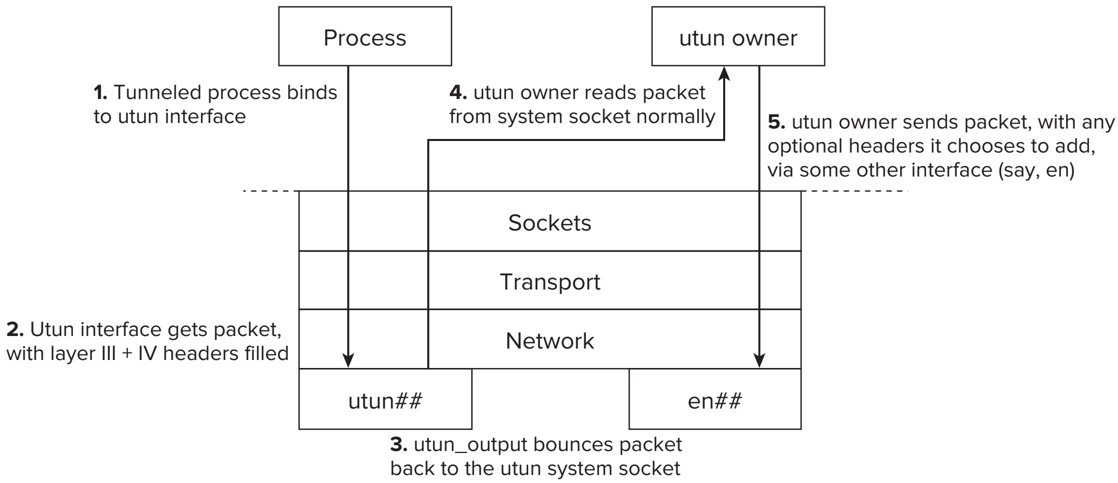


FIGURE 17-4: Sending packets through a user tunnel (utun) interface

Any of the pseudo-interfaces in the kernel make for good examples of how to set up and initialize `ifnet` instances, but `utun` in particular also makes for a good example of system sockets. The utuns are created by the kernel when the user mode tunnel process creates a `PF_SYSTEM` socket, issues a `CTLIOCGINFO ioctl(2)` to bind it to the `utun` namespace, and then calls `connect(2)`. Sample code to do so is shown in Listing 17-15:

LISTING 17-15: Sample code to bind a new utun interface

```
int tun(unsigned int num)
{
        struct sockaddr_ctl sc;
        struct ctl_info ctlInfo;
        int s;                       // returned socket descriptor


        memset(&ctlInfo, 0, sizeof(ctlInfo));
        strncpy(ctlInfo.ctl_name, UTUN_CONTROL_NAME, sizeof(ctlInfo.ctl_name);

        s = socket(PF_SYSTEM, SOCK_DGRAM, SYSPROTO_CONTROL);
        if (s < 0) { perror ("socket"; return -1; }

        if (ioctl(s, CTLIOCGINFO, &ctlInfo) == -1) {
                perror("CTLIOCGINFO");
                close(s);
                return -1;
        }

        sc.sc_family  = PF_SYSTEM;
        sc.ss_sysaddr = AF_SYS_CONTROL;
        sc.sc_id = ctlInfo.ctl_id;
```

continues

LISTING 17-15 *(continued)*

```
        sc.sc_len = sizeof(sc);

        sc.sc_unit = num;
        if (connect(s, (struct sockaddr *)&sc, sizeof(sc)) == -1) {
             perror("connect");
             close(s);
             return -1;
        }
        return s;
}
```

Switching to the kernel perspective, when the user mode process connects, the `utun_ctl_connect` (`bsd/net/if_utun.c`) is called. This function creates and initializes a new `utun` interface, as shown in Listing 17-16:

LISTING 17-16: utun_ctl_connect(), demonstrating interface creation

```
static errno_t
utun_ctl_connect(
        kern_ctl_ref            kctlref,
        struct sockaddr_ctl     *sac,
        void                            **unitinfo)
{
        struct ifnet_init_params        utun_init;
        struct utun_pcb                         *pcb;
        errno_t                                         result;
        struct ifnet_stats_param        stats;

        /* kernel control allocates, interface frees */
        pcb = utun_alloc(sizeof(*pcb));
        if (pcb == NULL)
                return ENOMEM;

        /* Setup the protocol control block */
        bzero(pcb, sizeof(*pcb));
        *unitinfo = pcb;
        pcb->utun_ctlref = kctlref;
        pcb->utun_unit = sac->sc_unit;

        printf("utun_ctl_connect: creating interface utun%d\n", pcb->utun_unit - 1);

        /* Create the interface */     Name + unit will make up visible name (e.g. utun0)
        bzero(&utun_init, sizeof(utun_init));
        utun_init.name = "utun";                  Note setting of utun_init structure,
        utun_init.unit = pcb->utun_unit - 1;      which is an ifnet_init_params,
        utun_init.family = utun_family;           setting all the non-private fields
        utun_init.type = IFT_OTHER;               of the soon to be allocated ifnet
        utun_init.output = utun_output;           structure.
        utun_init.demux = utun_demux;
        utun_init.framer = utun_framer;
```

```
        utun_init.add_proto = utun_add_proto;
        utun_init.del_proto = utun_del_proto;
        utun_init.softc = pcb;
        utun_init.ioctl = utun_ioctl;
        utun_init.detach = utun_detached;

        result = ifnet_allocate(&utun_init, &pcb->utun_ifp);
        if (result != 0) {
                printf("utun_ctl_connect - ifnet_allocate failed: %d\n", result);
                utun_free(pcb);
                return result;
        }

        OSIncrementAtomic(&utun_ifcount);  // OSIncrementAtomic avoids having to lock

        /* Set flags and additional information.*/ // parameters which init cannot set
        ifnet_set_mtu(pcb->utun_ifp, 1500);

        // These flags are visible in ifconfig(8)
        ifnet_set_flags(pcb->utun_ifp,IFF_UP | IFF_MULTICAST | IFF_POINTOPOINT, 0xffff);

        /* The interface must generate its own IPv6 LinkLocal address,
         * if possible following the recommendation of RFC2472 to the 64bit interface ID
         */
        ifnet_set_eflags(pcb->utun_ifp, IFEF_NOAUTOIPV6LL, IFEF_NOAUTOIPV6LL);

        /* Reset the stats in case as the interface may have been recycled */
        bzero(&stats, sizeof(struct ifnet_stats_param));
        ifnet_set_stat(pcb->utun_ifp, &stats);

        /* Attach the interface */   // i.e. make it visible
        result = ifnet_attach(pcb->utun_ifp, NULL);
        if (result != 0) {
                printf("utun_ctl_connect - ifnet_allocate failed: %d\n", result);
                ifnet_release(pcb->utun_ifp);
                utun_free(pcb);
        }

        /* Attach to bpf */   // Must call bpfattach() if we want BPF (described later)
        if (result == 0)
                bpfattach(pcb->utun_ifp, DLT_NULL, 4);

        /* The interfaces resources allocated, mark it as running */
        if (result == 0)
                ifnet_set_flags(pcb->utun_ifp, IFF_RUNNING, IFF_RUNNING);

        return result;
}
```

Very similar logic can be seen in other interface creation routines. XNU's pseudo interface functions (`stfattach()`, `gif_clone_create()`, `pflog_clone_create()` and others), as well as (to an extent) the `IONetworkingFamily's IONetworkInterface::attachToDataLinkLayer()` follow this general flow.

When a packet is sent out through the `utun` interface, control eventually reaches DLIL, which calls the interface's output function, `utun_output`. This function calls `ctl_enqueuembuf(bsd/kern/kern_control.c)`, which finds the system socket the `utun` interface is linked with, and appends the output `mbuf` to its socket buffer, waking up the user mode process which owns this socket as it does so. The user mode process can then read from the socket, and obtain as its data the IP or IPv6 packet sent through the interface. This packet can then be encapsulated in whatever way the tunnel process sees fit.

When the user mode tunnel wants to inject a packet, it writes to the system socket. This results in a call to the system socket's `ctl_send` handler, set by `utun_control_register()` (called when `utun` is set up, during `bsd_init()`) to be `utun_ctl_send()`. This function calls `dlil's ifnet_input()` with the same `mbuf` it was passed, simulating frame arrival, and from there the `mbuf` flows up the normal interface-to-socket receive path. This path, along with its inverse, the send path, are described in the next section.

PUTTING IT ALL TOGETHER: THE STACK

Now that we have covered all the separate layers of the stack: the interface (`struct ifnet`), network protocol (`struct proto_input entry`), the transport protocol (`struct protosw`) and the socket (`struct socket`), we can put the separate pieces of the puzzle to see how the stack operates as a whole for its two most important roles: sending and receiving data.

Receiving Data

Packet reception and processing requires the packet to traverse the stack upwards: from the interface level all the way up to the target socket.

Setup

Before data can be received, each interface must register itself with an input thread, as shown in Figure 17-5.

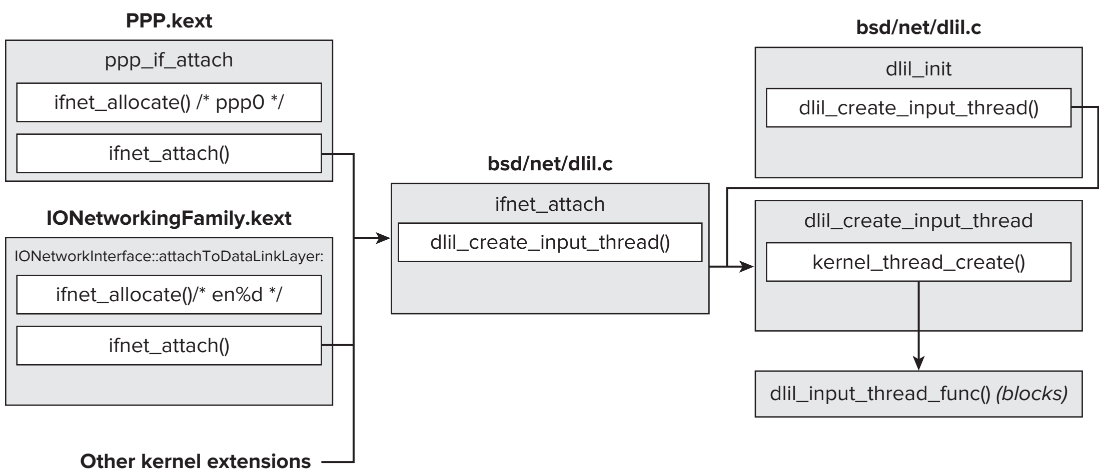


FIGURE 17-5: Setting up interface input threads

The Data Link Layer creates dedicated input threads, using `dlil_create_input_thread()`. The first input thread handles the loopback interface (`lo_ifp`), and is created by `dlil_init()` during system startup (as part of `bsd_init()`). Additional threads are created by calls to `ifnet_attach()`, when new interfaces are created (either XNU's built-in ones, or interfaces created by kexts, such as IONetworkingFamily).

The input threads all run the `dlil_input_thread_func()` continuously. This function accepts a `dlil_threading_info` structure, shown in Listing 17-17.

LISTING 17-17: The dlil_threading_info, from bsd/net/dlil.h:

```
struct dlil_threading_info {
        decl_lck_mtx_data(, input_lck);
        lck_grp_t       *lck_grp;         /* lock group (for lock stats) */
        mbuf_t          mbuf_head;        /* start of mbuf list from if  */
        mbuf_t          mbuf_tail;        // last mbuf from interface
        u_int32_t       mbuf_count;       // total number of mbufs (for walking list)
        boolean_t       net_affinity;     /* affinity set is available   */
        u_int32_t       input_waiting;    /* DLIL condition of thread    */
        struct thread   *input_thread;    /* thread data for this input  */
        struct thread   *workloop_thread; /* current workloop thread     */
        u_int32_t       tag;              /* current affinity tag        */
        char            input_name[DLIL_THREADNAME_LEN];
#if IFNET_INPUT_SANITY_CHK
// ...
#endif
};
```

The `dlil_input_thread_func()` sleeps on its `input_waiting` flag, waiting for input to become available.

Receiving Input

Figure 17-6 illustrates the process of receiving input. When a packet is received on an interface, `ifnet_input()` is called, with a pointer to the interface and a pointer to the head of the packet's `mbuf` chain. The function walks the `mbuf` chain, and finds the dedicated input thread of this interface (or, if none exists, redirects to the loopback thread). It adds the `mbuf` to the thread — either as the first packet (the threading info's `mbuf_head` member) or the last one (`mbuf_tail->m_nextpkt`), raises the `DLIL_INPUT_WAITING` flag on the `input_waiting` member, and increments the interface statistics. This causes `dlil_input_thread_func()` to wake up (as input has become available), and run its course, as shown in Figure 17-7.

The rest of the processing occurs in the interface's input thread: `dlil_input_thread_func()` proceeds to dequeue the first `mbuf` (in `mbuf_head`), and call `dlil_input_packet_list()` on that `mbuf`.

The `dlil_input_packet_list()`, true to its name, walks the `mbuf` chain, beginning with its argument. It finds which interface it is working for (either by its first argument, if it is the loopback interface, or by the `mbuf's m_pkthdr.rcvif` field. It then calls the interface's `ifp_demux` function to find which protocol family this `mbuf` should be handled by. Prior to looking up the actual protocol, it calls `dlil_interface_filters_input()`, which is responsible for running any interface filters on the `mbuf`. The interface filters may claim the `mbuf` (causing `dlil_interface_filters_input()` to return `EJUSTRETURN`, and `dlil_input_packet_list()` to skip to the next `mbuf`).

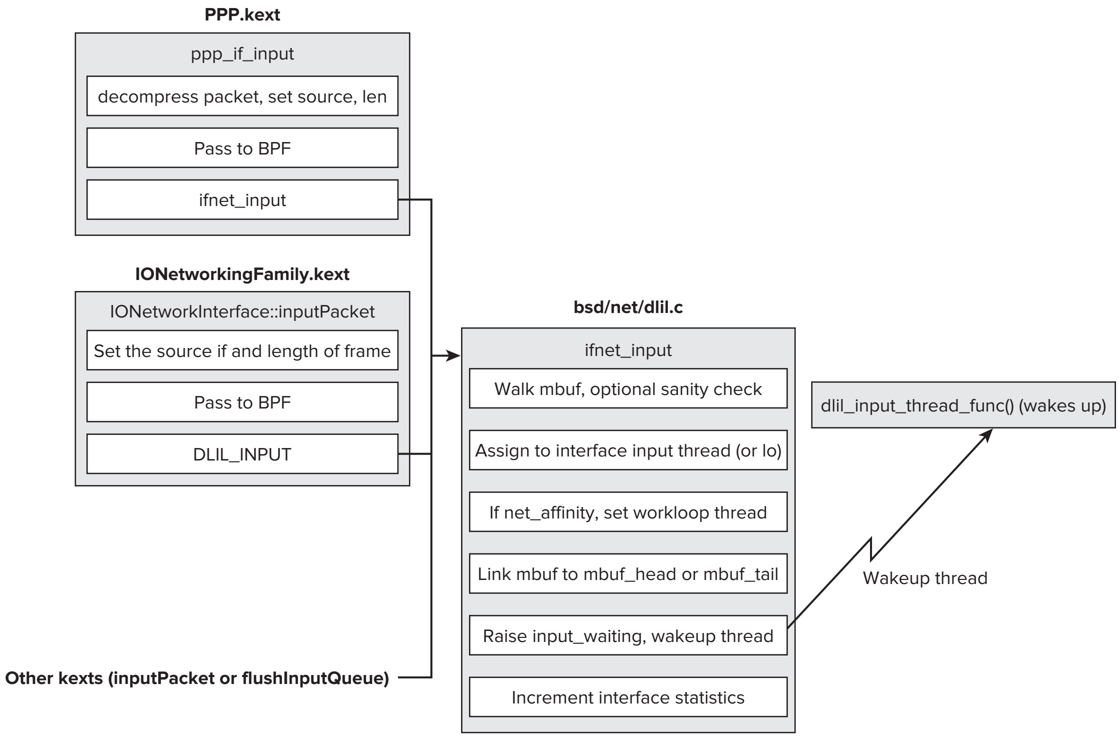


FIGURE 17-6: Frame reception, from driver to DLIL

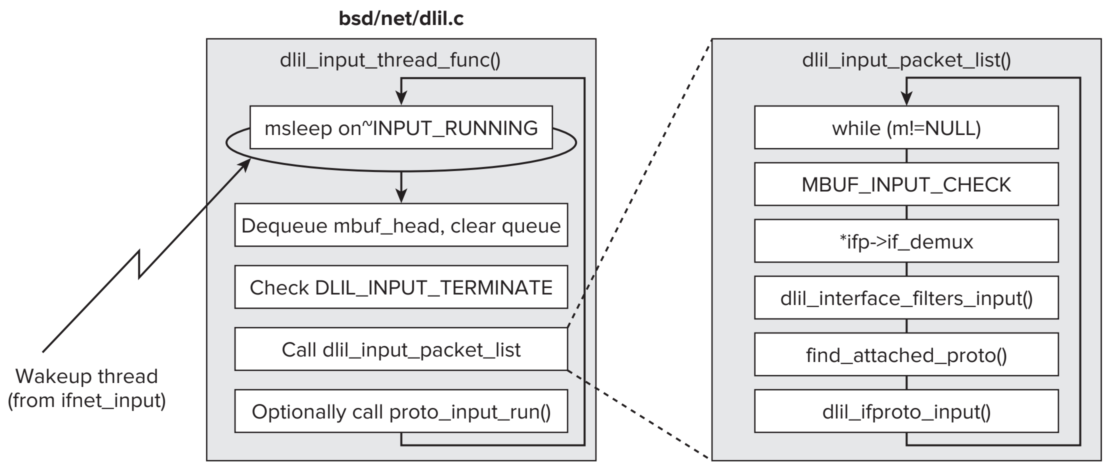


FIGURE 17-7: dlil_input_thread_func(), detailed

If the interface filters did not claim the packet, a call to `find_attached_proto()` (to look up the protocols in the aforementioned `proto_hash` "hash table"), or a cached value of `last_ifproto` obtains a call to the correct protocol handler, and a call to `dlil_ifproto_input()`, with the protocol handler and the first packet of the list, passes control to the protocol handler. Depending on the protocol handler version, it is expected to process one packet at a time (version 1), or the full packet list (version 2), by a call to its registered input function, a `proto_input` function. The IPv4 and IPv6 functions are somewhat similar, but naturally involve different logic. The IPv4 handler is shown in Figure 17-8.

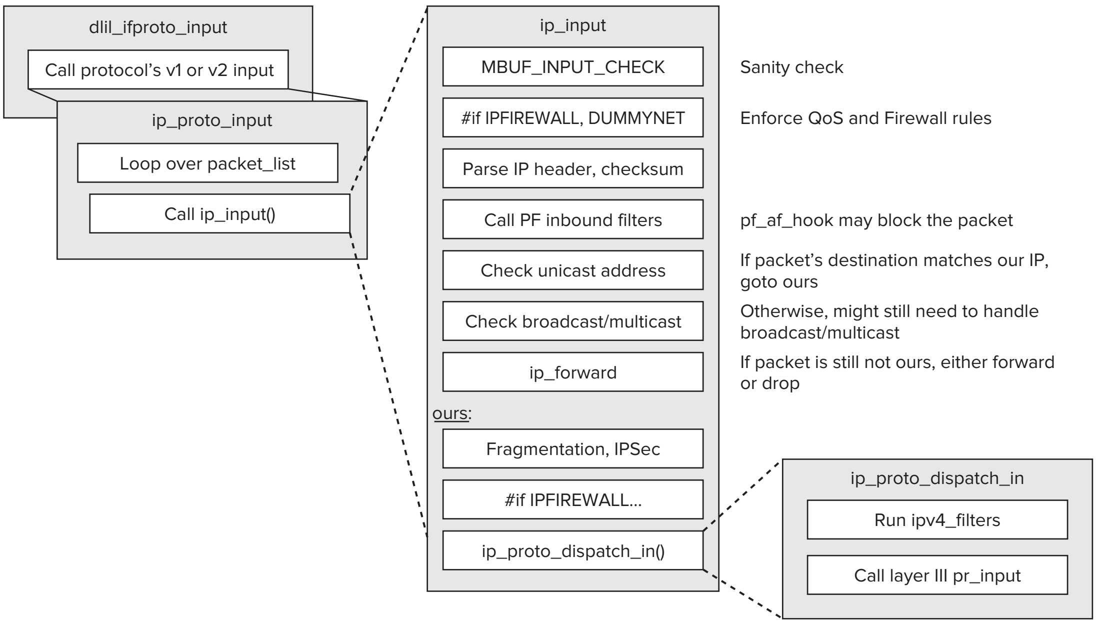


FIGURE 17-8: The ip_proto_input function

The transport protocol handler's `proto_input` function calls its input function. This extra level is necessary to support the legacy design of IPv4's input function (`ip_input`), which can handle only one packet at a time. The `ip_proto_input` function, therefore, walks the packet list. (IPv6 simply falls through to `ip6_input`.) The input functions perform all the necessary header checks, invoke any firewall or PF filter checks, check the destination ("forward" or "ours"), and (if "ours") potentially reassemble the packet, decrypt IPSec, and call the transport protocol's input handler either directly (IPv6) or indirectly (through IPv4's `ip_proto_dispatch_in()`). In either case, before the transport protocol can take over, the network protocol's filters (`ipv4_filters` or `ipv6_filters`, respectively) are called. IP filtering is discussed later in this chapter).

The transport protocol's input function performs the necessary adjustments of that layer, before finding the corresponding socket and delivering the packet. This is done by looking up the packet's corresponding PCB, by looping over the `inp_list` of PCBs. If no PCB can be found, a TCP packet generates a RST, and a UDP one similarly results in an ICMP unreachable. The `mbuf` is appended to the socket's receive buffers (`so_rcv`) by calling one of four functions as shown in Table 17-13. All four return non-zero on success, and are defined in `bsd/kern/uipc_socket2.c`:

TABLE 17-13: Functions Used to Append an mbuf to a Socket's Buffer

FUNCTION	USED FOR
sbappend(struct sockbuf *sb, struct mbuf *m);	Appending an `mbuf` *m* to the `sockbuf` *sb*. Used by `PF_SYSTEM` sockets
sbappendrecord(struct sockbuf *sb, struct mbuf *m0);	As `sbappend()`, but opens a new record. Called by `sbappend` if no record exists for the socket
sbappendstream (struct sockbuf*sb, struct mbuf *m)	As `sbappend()`, but optimized for stream sockets. Used by TCP
sbappendaddr (struct sockbuf *sb, struct sockaddr *asa, struct mbuf *m0, struct mbuf *control, int *error_out);	As `sbappend()`, but also provide the socket address details in *asa*. Used by UDP (for `recvfrom()` in user mode), and by raw IP

When data has been delivered, the socket is awakened by `sowakeup()`. This function wakes up the threads blocking on the socket (i.e. waiting in its wait queue), causing `select(2)/poll(2)` or `recv(2)` to return. If the socket is asynchronous (`so->so_state & SS_ASYNC`), the function sends the process a SIGIO.

Sending Data

When sending data, the data originates from user mode and is passed to a socket using the `send(2)`, `sendto(2)`, `sendmsg(2)`, or `sendfile(2)` (`#if SENDFILE`) system call.

With the exception of the last, all these system calls end up using `sendit` (`bsd/kern/uipc_syscalls.c`). This function looks up the struct socket from the file descriptor (using `file_socket()`and `fp_lookup()`, as described earlier). Process the message headers, if any, and proceeds to send, after consulting the MAC framework (`mac_socket_check_send`) for compliance with the current security policy. The send operation itself is performed by accessing the socket's registered transport protocol (the `protosw`), getting its user request structure (`pr_usrreqs`), and invoking its `pru_sosend` member, as discussed previously in this chapter under "Transport Protocols." The error code the send operation returns is propagated back to the caller, unless it is `EINTR`, `EWOULDBLOCK`, or `ERESTART`. `EPIPE` error codes trigger a `SIGPIPE` to the owning process, unless the socket option of `NOSIGPIPE` was set. This is Shown in Figure 17-9.

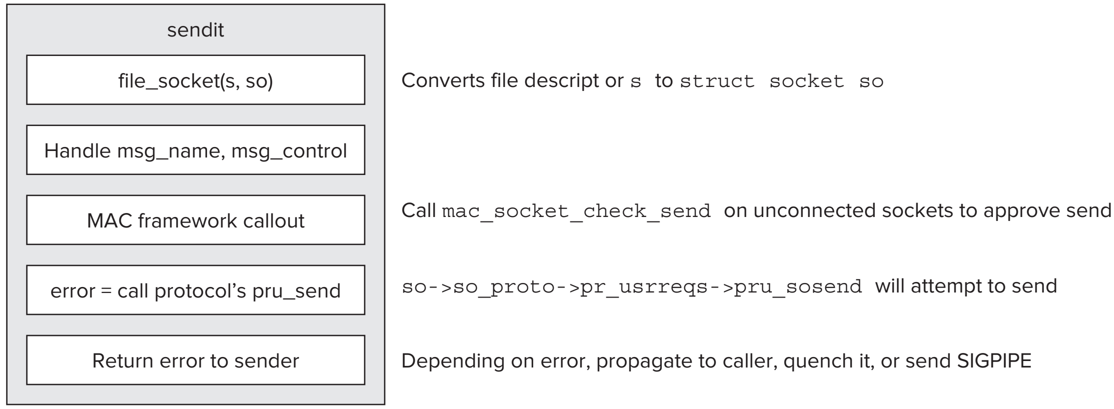


FIGURE 17-9: The flow from socket to transport protocol

The various transport protocols naturally have different `pru_sosend` implementations, depending on the header they need to construct for the data, and the protocol type (stream or datagram). All `pru_sosend` functions, however, share the same prototype: The socket, flags, the `mbuf` containing the data, a `sockaddr` to send to, an `mbuf` containing socket control information, and the current process pointer. The functions generally follow the same flow: convert the socket to a PCB structure using `sotoinpcb()`, construct the header, and pass the `mbuf` to the network protocol (`ip_output_list()` or `ip6_output()`). A simple example is UDP's send, which does this through a call to `udp_output()` shown in Listing 17-18:

LISTING 17-18: udp_send (from bsd/netinet/udp_usrreq.c)

```
static int
udp_send(struct socket *so, __unused int flags, struct mbuf *m, struct sockaddr *addr,
            struct mbuf *control, struct proc *p)
{
        struct inpcb *inp;

        inp = sotoinpcb(so);
        if (inp == 0) {
                m_freem(m);
                return EINVAL;
        }

        return udp_output(inp, m, addr, control, p);
}

// note retro style function definition of udp_output (if it ain't broken, don't fix it)
static int
udp_output(inp, m, addr, control, p)
        register struct inpcb *inp;
        struct mbuf *m;
        struct sockaddr *addr;
```

continues

LISTING 17-18 *(continued)*

```
        struct mbuf *control;
        struct proc *p;
{
   // ...
   int soopts = 0;
   struct mbuf *inpopts;
   struct ip_moptions *mopts;
   struct route ro;
   struct ip_out_args ipoa = { IFSCOPE_NONE, 0 };
   // ...
   inpopts = inp->inp_options;
   soopts |= (inp->inp_socket->so_options & (SO_DONTROUTE | SO_BROADCAST));
   mopts = inp->inp_moptions;
   error = ip_output_list(m, 0, inpopts, &ro, soopts, mopts, &ipoa);
   // ...
}
```

The network protocol's output function finds a route for the packet, from which the outgoing interface can be inferred. Before that can happen, IPv4's ARP or IPv6's ND need to be used to find the next hop's link layer address (unless previously cached). When the address is at hand, a call to `ifnet_output()` (which wraps `dlil_output()`) finally passes the packet to the data link interface layer (See Figure 17-10).

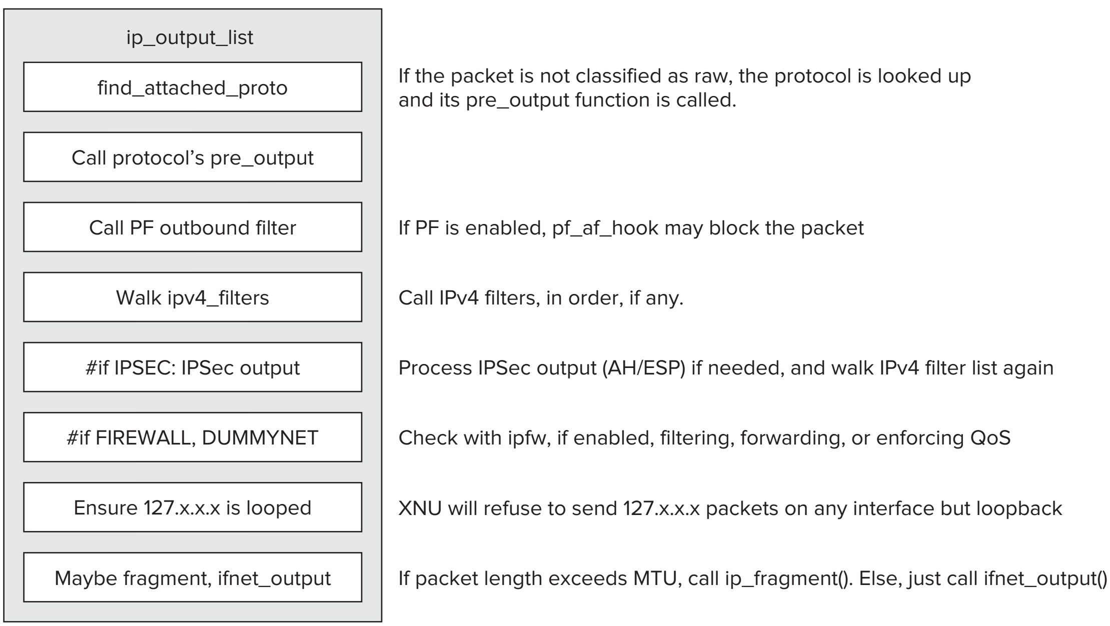


FIGURE 17-10: The flow of IP's ip_output_list()

The flow is not yet done. As shown in Figure 17-11, `dlil_output()` finds the interface's attached protocol (so it can call its `pre_output` function, if any). It then verifies with the MAC framework that the packet may be transmitted (by a callout to `mac_ifnet_check_transmit`), calls the interface's "framer" function (to create the link layer header), and calls any interface filters (discussed later) to potentially intercept prior to sending. If all goes well, a call to the interface's `if_output` handler (which for a "real" interface is handled by its driver kext) performs the actual send operation (for IOKit drivers, this calls `IONetworkController::outputPacket`). For packets classified as "raw," the protocol `pre_output` and framer steps are skipped.

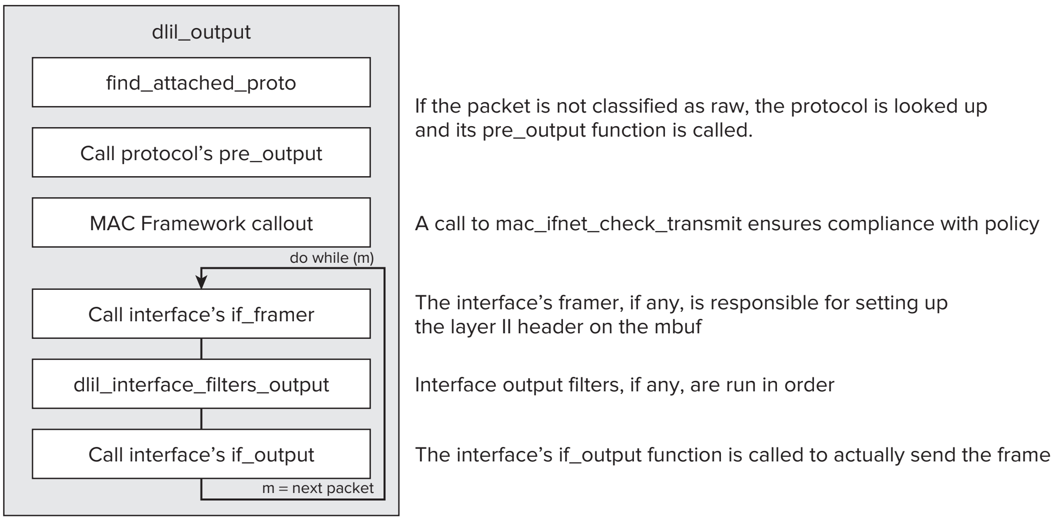


FIGURE 17-11: The flow of dlil_output()

PACKET FILTERING

Relatively few developers need to write full network drivers. Filtering packets, however, is commonplace. Whether for security or insecurity purposes, being able to inspect a host's traffic in real time offers unprecedented power. The network space is an arena wherein two major forces vie for supremacy: In the blue corner, the anti-virus and firewall providers, who seek to secure the host by inspecting both ingress and egress traffic. In the red corner, the malware and spyware "providers" who establish covert channels in the network, by means of which they can both eavesdrop as well as usurp control of the host. It is only fitting, therefore, that a section be devoted to the exciting realm of packet filtering.

BSD has a host of filtering mechanisms. Each offers its own abilities, both advantageous and disadvantageous. XNU, as an implementation of BSD, supports all these technologies, and they are detailed next. For certain tasks, picking a particular mechanism over another may be preferable. Table 17-14 illustrates the different abilities of these mechanisms.

TABLE 17-14: Comparison of Filter Techniques

ABILITY	SOCKET FILTERS	IPFW/PF	IP FILTERS	INTERFACE FILTERS	BPF
Mode	Kernel	User	Kernel	Kernel	User
Technique	API hook	Firewall	Firewall	Firewall	Packet filter
OSI layer	V (Session)	III (Network)	III (Network)	II (Data Link)	II (Data Link)
Packet Injection	Yes	No	No	Yes	Yes
Counterpart	Windows: Winsock SPI Linux: Socket hooking	Linux: IPTables	Linux: Netfilter hooks	Linux: BRTables	(Ported to Linux)

The kernel APIs are meant to be accessed from Network Kernel Extensions (NKEs), and Apple Developer's NKE Programming Guide[17] documents the filters (socket, IP and interface) very well. Another discussion can be found in Halvorsen & Clarke's book[18]. Nonetheless, we review them here briefly here, alongside the other mechanisms, which are not described in either.

Socket Filters

The highest level in which filters can be placed is that of the socket itself. The kernel implementation of sockets, described previously, allows a kernel extension to associate a socket filter using a special KPI. The KPI has been significantly slimmed down from its earlier incarnations, and covers a subset of the user mode socket API calls.

A socket filter is implemented as a struct `sflt_filter`. This structure, alongside the KPI functions exposed for setting, attaching and detaching it from a socket, is defined in the well documented `bsd/sys/kpi_socketfilter.h`. These functions (all return `errno_t)` are shown in Table 17-15:

TABLE 17-15: Socket Filter KPIs Exposed in bsd/sys/kpi_socketfilter.h

SOCKET KPI CALL	PURPOSE
`sflt_register` `(const struct sflt_filter *f,` `int domain,` `int type,` `int protocol);` **`sflt_unregister`** `(sflt_handle handle)`	Register a socket filter for specified *`domain`*, *`type`* and *`protocol`*. To unregister, use the filter's *`handle`* field.

SOCKET KPI CALL	CORRESPONDING API CALL
sflt_attach(socket_t so, sflt_handle *h*); **sflt_detach**(socket_t so, sflt_handle *h*);	Attach/Detach socket filter specified in handle *h* to/from socket *so*.
sock_inject_data_in (socket_t so, const struct sockaddr *from, mbuf_t data, mbuf_t control, sflt_data_flag_t flags); **sock_inject_data_out** (socket_t so, const struct sockaddr *to, mbuf_t data, mbuf_t control, sflt_data_flag_t flags);	Inject *data* mbuf into socket so's input or output stream. On unconnected (e.g. UPD) sockets, the caller may specify the fake sockaddr address (*from*/*to*).

The struct sflt_filter itself consists of a handle, flags, and a collection of function pointers, which are callbacks that will be invoked by the socket calls for registered socket filters. The annotated structure is shown in Listing 17-19:

LISTING 17-19: The XNU socket filter implementation

```
struct sflt_filter {
   sflt_handle            sf_handle; // accessible to apps using SO_NKE setsockopt(2)
   int                    sf_flags;  // SFLT_GLOBAL, SFLT_PROG or SFLT_EXTENDED
   char                  *sf_name;
   sf_unregistered_func   sf_unregistered;
   sf_attach_func         sf_attach;      // called on successful sflt_attach()
   sf_detach_func         sf_detach;      // called on successful sflt_detach()

   sf_notify_func         sf_notify;      // called with an sflt_event_t specifying
                                          // connect/disconnect/bound/buffers full/etc
   sf_getpeername_func    sf_getpeername; // called on getpeername(2)
   sf_getsockname_func    sf_getsockname; // called on getsockname(2)
   sf_data_in_func        sf_data_in;     // called before data is delivered to thread
   sf_data_out_func       sf_data_out;    // called before data is queued for sending
   sf_connect_in_func     sf_connect_in;  // called for incoming connections - accept
   sf_connect_out_func    sf_connect_out; // called for outgoing connections - connect
   sf_bind_func           sf_bind;        // called on bind(2)
   sf_setoption_func      sf_setoption;   // called on setsockopt(2)
   sf_getoption_func      sf_getoption;   // called on getsockopt(2)
   sf_listen_func         sf_listen;      // called on listen(2)
   sf_ioctl_func          sf_ioctl;       // called on ioctl(2)
```

continues

LISTING 17-19 *(continued)*

```
        /*
         * The following are valid only if SFLT_EXTENDED flag is set.
         * Initialize sf_ext_len to sizeof sflt_filter_ext structure.
         * Filters must also initialize reserved fields with zeroes.
         */
    struct sflt_filter_ext {
           unsigned int        sf_ext_len;
           sf_accept_func      sf_ext_accept;     // called before accept(2) returns
           void               *sf_ext_rsvd[5];    /* Reserved */
     } sf_ext;
#define sf_len          sf_ext.sf_ext_len
#define sf_accept       sf_ext.sf_ext_accept
};
```

The callbacks specified effectively cover all the socket APIs. Their prototypes match those of the corresponding user mode calls, with some subtle differences (e.g. the `int socket` is replaced by the kernel's `socket_t`, and the user mode `char *` buffers are replaced by the lower level `mbufs`).

The socket filter can be registered as a global filter (using the `SFLT_GLOBAL` flag), which will attach it to all sockets created from that point onward, or as a programmatic filter (`SFLT_PROG`), which will be attached only upon a specific application request. To request attachment, user mode applications can use the Apple specific `SO_NKE` `setsockopt(2)`.

Apple Developer has a well documented example in TCPLogNKE[19], which the reader is encouraged to peruse.

ipfw(8)

BSD-based kernels, like Linux, are not without a built-in firewalling functionality. What Linux refers to it as "`iptables`" BSD calls "`ipfw`." In BSD the mechanism can also be extended to layer II (for example, "`brtables`"), but this is not the case in XNU.

> *ipfw has been deprecated in favor of the more powerful PF mechanism (described next). It is included here for completeness, and still exists in Lion, but will likely be removed in an upcoming release.*

Controlling Parameters from User Mode

The `ipfw` mechanism can be controlled in a very fine-grained manner using a single command — `ipfw(8)` (or `ip6fw(8)` for IPv6), which enables root to define the rules and their default action. In addition, the mechanism exports several `sysctl(8)`-visible parameters, listed in Table 17-16:

TABLE 17-16: sysctl Variables for ipfw and heir Defaults in XNU.

NET.INET.IP.FW.* (NET.INET6.IP.FW.*)	DEFAULT VALUE	USED FOR
`autoinc_step`	100	Auto-increments value when creating dynamic (automatic) rules.
`curr_dyn_buckets`	N/A	Shows current number of hash buckets for dynamic rules.
`dyn_buckets`	256	Maximum number of buckets for dynamic rules (must be a power of 2).
`dyn_count`	N/A	Current number of dynamic rules. Always less than or equal to `dyn_max`, below.
`dyn_keepalive`	1	Automatically sends keep-alive packets for rules set to `keep-state`. These are sent from the kernel, and user mode remains oblivious to their existence.
`dyn_max`	4096	Maximum number of dynamic rules.
`dyn_ack_lifetime` `dyn_syn_lifetime` `dyn_fin_lifetime` `dyn_rst_lifetime`	300 20 1 1	Number of seconds controlling the lifetime of various stage TCP dynamic rules.
`dyn_udp_lifetime`	5	Number of seconds controlling the UDP rules.
`static_count`	N/A	Number of static rules.
`enable*`	1	Enables/disables ipfw globally.
`debug*` `verbose*` `verbose_limit*`	0 1 0	Generates debug messages, optionally verbose, and up to *`verbose_limit`* messages (note that `verbose_limit 0` effectively disables `verbose`).

Variables with a (*) also exist separately in the `net.inet6.ip6.fw` namespace.

Note that the `ipfw(8)` man page, a verbatim copy of BSD's, is wrong on several of these values. The man page further mentions the `net.link.ether.ipfw` and `bridge_ipfw` variables for layer II firewalling, but they are not supported in XNU.

The PF Packet Filter (Lion and iOS)

With Lion, Apple has integrated another BSD packet filtering mechanism, PF, into XNU. PF source code has actually been part of XNU from earlier Snow Leopard versions, but has been #ifdef'd out, and enabled only in iOS. PF is a one-stop interface for firewalling, and like `ipfw(8)`, offers the system administrator a simple utility — `pfctl(8)` to manage its rulebase. A quick way to see whether PF is enabled is to check for the existence of a `/dev/pf` file, as follows:

```
root@Padishah:~  # ls -l /dev/pf
crw------- 1 root wheel 7, 0 Nov 23 06:54 /dev/pf   # 8,0 on Lion
```

`pfctl(8)` opens the PF device, and manages rules by issuing corresponding `ioctl(2)` calls — `DIOCADDRULE`, `DIOCGETRULE`(S), and `DIOCCHANGERULE`. PF also enables user mode to view logged packets in an elegant way. Instead of looking at log files, an administrator can use `ifconfig(8) to` create the `pflog(4)` pseudo-interface. A user mode process can then bind to the interface, which will replicate all logged packets. A common use of this is to use `tcpdump(1)` or other packet capturing tools this way (see the manual page for an example).

The PF filter callouts (via `pf_af_hook()`) can be seen in Figures 17-8 (input) and 17-10 (output), respectively. PF is well documented in the corresponding man page (`man pfctl` on Lion and later), and in its own book[20]. Also, because PF is a fairly rigorous and non-extensible mechanism, it is not elaborated on here.

> *A classic buffer overflow in older versions of PF was used by the jailbreaker comex in his "spirit" jailbreak. The bug is now classified as CVE-2010-3830[21], or by its more verbose name, "iOS < 4.2.1 packet filter local kernel vulnerability," and a detailed discussion of it can be found at Sogeti's site[22]. In a nutshell, this bug allows an arbitrary overwrite (specifically, decrement) of kernel space memory by opening* `/dev/pf` *and issuing a* `DIOCADDRULE ioctl`*. Even though* `/dev/pf` *requires root privileges to open, comex was able to construct a two-staged exploit, with the first stage obtaining root via geohot's boot ROM exploit, and dropping the second stage to be executed by* `launchd(8)` *each time the iDevice is booted. As with the NDRV exploit discussed earlier in this chapter, the kernel memory overwrite provides the "untethered" part of the exploit by disabling code signing checks and memory write protections.*
>
> *Following the exploit, Apple fixed the* `DIOCADDRULE` *and* `DIOCGETRULE` *handlers. The changes were incorporated into OpenBSD, as well. Nonetheless, this is yet another example of how Apple's reliance on third-party code inherits with it third-party security vulnerabilities.*

IP Filters

Whereas firewalling allows for a rather limited accept/deny/drop functionality, filtering enables more detailed packet inspection, and even modification. BSD includes an IP filtering mechanism not unlike Linux's NetFilter (IPTables). The IP filters are invoked by the stack as callouts from specific points.

This mechanism is very powerful, and power corrupts. Indeed, IP filtering is commonly used in malware rootkits — Dino Dai Zovi's "Machiavelli"[23] uses the IPFilter framework in its rootkit component.

The ipf_filter Structure

An IP filter, called `ipf_filter` throughout the kernel, is basically two callback functions: one for filtering inbound traffic (`ipf_input`), and one for the outbound traffic (`ipf_output`). Additionally, an `ipf_detach` function can be used to handle filter detachment. A filter can also have a free text name and a "cookie." This "cookie" is an opaque, void pointer and may be used to pass a structure or some other argument to the filter functions (See Listing 17-20).

LISTING 17-20: The IPFilter and opaque IPFilter from bsd/netinet/kpi_ipfilter.c

```
/*!
        @typedef ipf_filter
        @discussion This structure is used to define an IP filter for
                use with the ipf_addv4 or ipf_addv6 function.
        @field cookie A kext defined cookie that will be passed to all
                filter functions.
        @field name A filter name used for debugging purposes.
        @field ipf_input The filter function to handle inbound packets.
        @field ipf_output The filter function to handle outbound packets.
        @field ipf_detach The filter function to notify of a detach.
*/
struct ipf_filter {
        void            *cookie;    // opaque value, caller defined, passed to functions
        const char      *name;
        ipf_input_func  ipf_input;  // Handles input packets     (see below)
        ipf_output_func ipf_output; // Handles output packets    (see below)
        ipf_detach_func ipf_detach; // Handles filter detachment (see below)
};

struct opaque_ipfilter;
typedef struct opaque_ipfilter *ipfilter_t;
```

The kernel maintains two filter lists: `ipv4_filters` and `ipv6_filters`. An additional filter list — `tbr_filters` — is used for defunct filters are to be removed. All three lists are opaque, however, and filters should only be manually added to the first two lists by a call to `ipf_addv4` or `ipf_addv6`, respectively.

Implementing Filter Functions

A filter can choose to implement either ingress or egress function (or both), and can optionally specify a detach function. The functions adhere to a set interface, as shown in Listing 17-21.

LISTING 17-21: Interface filter function prototypes (from bsd/netinet/kpi_ipfilter.h)

```
typedef errno_t(*ipf_input_func)(void *cookie,mbuf_t *data,int offset,u_int8_t
protocol); (*ipf_output_func)(void *cookie,
mbuf_t *data,  ipf_pktopts_t options);
typedef void (*ipf_detach_func)(void *cookie);
```

The input and output functions get the data to be filtered, along with a cookie value, which is the pointer value specified during filter creation. The filters can then do whatever processing is required, returning 0 to signal the packet is ok (normal processing), `EJUSTRETURN` to instruct the stack to drop the packet, but not free the `mbuf`. Any other non-zero value, will instruct the stack to drop the packet, and free the `mbuf` as well.

Filter Callout Locations

Once installed, user-specified filters are called out from the IP stack at two specific locations:

- **Packet input:** The IP protocol input functions (`ip_proto_dispatch_in` in `bsd/netinet/ip_input.c` for IPv4 and `ip6_input` in `bsd/netinet6/ip6_input.c` for IPv6) iterate over the corresponding filter list (`ipv[46]_filters`) and call the `ipf_input` member function, if set.
- **Packet output:** The IP protocol output functions (`ip_output_list` in `bsd/netinet/ip_output.c` for IPv4, and `ip6_output` in `bsd/netinet6/ip6_output.c` for IPv6) similarly iterate over the filter list and call the `ipf_output` member function, if set. The IPv4 handler actually calls the filters on two separate occasions, one for multicast and one for normal packets, but the two cases are mutually exclusive.

Listing 17-22 shows how the filter list is walked from `ip6_input()`:

LISTING 17-22: Walking ipv6_filters, from ip6_input() (bsd/netinet6/ip6_input.c)

```
/*
 * Call IP filter
 */
 if (!TAILQ_EMPTY(&ipv6_filters)) {
     ipf_ref();
     // Walk the v6 filter list   (v4 is very similar)
     TAILQ_FOREACH(filter, &ipv6_filters, ipf_link) {
            if (seen == 0) {
                if ((struct ipfilter *)inject_ipfref == filter)
                   seen = 1;
              } else if (filter->ipf_filter.ipf_input) {
                 // If an input filter exists, execute it on this mbuf
                 errno_t result;
                 result = filter->ipf_filter.ipf_input(
                 filter->ipf_filter.cookie, (mbuf_t*)&m, off, nxt);
                 // If filter returns "EJUSTRETURN", packet is intercepted
                 if (result == EJUSTRETURN) {
                     ipf_unref();
                     goto done;   // packet dropped, mbuf is not freed
                    }
           if (result != 0) {
                    ipf_unref();
                    goto bad;      // packet dropped, mbuf is freed
               }
           }
         }
     ipf_unref();
 }
```

Interface Filters

The lowest level in which filters can be placed is that of the network interface. These filters are conceptually similar to socket and IP filters, but the lower level allows the filter to intercept and manipulate the packets before any further processing by upper layers.

An interface filter is a `struct iff_filter`, defined in `bsd/net/kpi_interfacefilter.h` as shown in Listing 17-23:

LISTING 17-23: An interface filter, annotated

```
struct iff_filter {
   void                    *iff_cookie;  // argument to filter functions
   const char              *iff_name;    // filter name (not really useful)
   protocol_family_t       iff_protocol; // 0 (all packets) or specific protocol
   iff_input_func          iff_input;    // optional filter for input packets, or NULL
   iff_output_func         iff_output;   // optional filter for output packets, or NULL
   iff_event_func          iff_event;    // optional filter for interface events,or NULL
   iff_ioctl_func          iff_ioctl;    // optional filter for ioctls on interface
   iff_detached_func       iff_detached; // required callback when filter is detached
};
```

The various filters all receive the interface (`ifnet_t`). The input and output filters receive the packet an `mbuf` chain. As with IP filters, the filter functions are expected to return 0 (accept), `EJUSTRETURN` (drop), or any non-zero value (drop, free). The filters are invoked by DLIL using `dlil_interface_filters_[input|output]()` prior to actually receiving or sending the frame (as shown in Figure 17-7 for the receive path, right before the call to `find_attached_proto()`).

The Berkeley Packet Filter

Low-level packet filters may not require protocol-level packet processing and prefer to work on the packets themselves, gaining even more efficiency in the process. McCanne and Van Jacobson (known for PPP compression and the traceroute algorithm) addressed this need by developing the BSD Packet Filter (BPF) back in 1993 and presenting it in a UseNIX paper[24]. BPF has since become a standard, powering many a network monitor (notably, TCPDump and libPCab-related tools). Because XNU's networking is based on BSD's, it has integrated BPF, as well. The code is contained in `bsd/net`, as shown in Table 17-17:

TABLE 17-17: BPF Implementation Files in XNU

BSD/NET FILE	USED FOR
`bpf.c`	The BPF supporting logic, `ioctls`, and `/dev` interface
`bpf_filter.c`	The BPF state machine
`bpf.h`	General definitions for `structs` and `ioctl` codes
`bpf_compat.h`	Compatibility hacks (`#defines`) for `malloc` and `free`
`bpf_desc.h`	Defining descriptors associated with BPF devices: `bpf_d` and `bpf_if`

BPF is structured around the notion of a "filter machine." The machine is a state machine with no loops or backward branches and limited opcodes. Ensuring no loops is critical, because the code runs in the kernel whenever a packet is processed and under tight constraints. The filter may inspect, but not modify any packets, though packets may be injected onto an interface.

To get started, a user mode program opens one of the `/dev/bpf#` devices. Each device can be attached to an underlying interface† with a given BPF program. There are usually four such files — `/dev/bpf0` through `/dev/bpf3` — but more files can be dynamically created as the need arises, up to `bpf_maxdevices` (set to 256, and also exported through `sysctl kern.debug`). Clients normally iterate over all devices and grab the first one available.

Controlling BPF is done exclusively through `ioctl(2)` calls. First, the BPF device has to be attached to an underlying interface (with a `BIOCSETIF ioctl`). Next, options may be set on the device, as shown in Table 17-18.

TABLE 17-18: BPF ioctls Related to Setting Options

BPF IOCTL	USED FOR
`BIOCSBLEN`	Sets buffer len. Called prior to attachment with `BIOCSETIF`. This buffer size must be adhered to in future `read(2)` calls.
`BIOCSRSIG`	Rather than block `read(2)`, this sends a signal (default: `SIGIO`) to process on packet availability.
`BIOCSSEESENT`	If set to non-zero, `read(2)` also returns (`SEE`) outgoing (`SENT`) packets from the underlying device, rather than just returning incoming ones.
`BIOCIMMEDIATE`	Returns immediately on packet availability, rather than blocking until a timeout or the buffer is full. Setting this overrides `BIOCSRTIMEOUT` (see next entry)
`BIOC[GS]RTIMEOUT`	Gets/sets timeout value, after which the `read(2)` operation will return. Setting this overrides `BIOCIMMEDIATE` (see preceding entry).
`BIOCPROMISC`	Sets underlying interface to promiscuous mode. Interface will deliver all frames, not just those matching its own hardware Address (or broadcast/multicast) to the kernel. This is useful for monitoring over hubs, for example.

To start reading from a device, a BPF program is defined by the client and set to execute on the interface by a `BIOCSETF ioctl(2)`. From that point onward, the client can simply employ standard `read(2)` system calls to retrieve packets (according to the options set in Table 17-18. The BPF program is thus key in determining which packets will be received on the device. Only packets matching the filter will be made available on the file descriptor.

†Only interfaces whose initialization code called `bpfattach()` and provided an `ifnet_set_bpf_tap` callback may be attached in this manner, though all common interfaces call `bpfattach()`, as do the ones initialized from Apple's kexts. Because this code is present in IONetworkingFamily, all the subclasses automatically become BPF-enabled

Building a BPF Program

A BPF program constitutes a program-within-a-program written in a format that can be understood by the BPF machine. The program is a `struct bpf_program`, which is constructed as an array of `bf_len bpf_insn structs`. Each `bpf_insn` represents a BPF instruction, defined as shown in Listing 17-24.

LISTING 17-24: The BPF instruction structure

```
/*
 * The instruction data structure.
 */
struct bpf_insn {
        u_short         code;   // The instruction op code
        u_char          jt;     // Conditions: Branch on argument eval true
        u_char          jf;     // Conditions: Branch on argument eval false
        bpf_u_int32     k;      // Argument for instructions. Depends on code
};

/*
 * Macros for insn array initializers.
 */
#define BPF_STMT(code, k) { (u_short)(code), 0, 0, k }
#define BPF_JUMP(code, k, jt, jf) { (u_short)(code), jt, jf, k }
```

Six "opcodes" can be used to inspect the incoming packets. The opcodes are understood by the BPF machine, which is a simple abstraction containing an instruction pointer, an accumulator register (for simple arithmetic), an index register, and limited memory. The machine is extremely limited, but considering its intended usage, is well suited to the task at hand of inspecting packets.

The `bpf(3)` manual page elaborates on the actual opcodes and patterns; the interested reader is advised to turn there for a more complete reference. Rather than repeat more of the same, this book turns to a practical example.

Experiment: Constructing a Sample BPF Program

Listing 17-25 demonstrates a sample generic filter for IPv4 packets, matching a specific protocol and port.

LISTING 17-25: A filter program to capture frames matching a specified protocol and port

```
int installFilter(int   fd,
         unsigned char  Protocol,
             unsigned short Port)
{
    struct bpf_program bpfProgram = {0};

    /* dump IPv4 packets matching Protocol and Port only */
    /* @param: fd - Open /dev/bpfX handle.               */

    /* As an exercise, you might want to extend this to IPv6, as well */
```

continues

LISTING 17-25 *(continued)*

```
    const int IPHeaderOffset = 14;

    /* Assuming Ethernet II frames, We have:
     *
     *    Ethernet header = 14 = 6 (dest) + 6 (src) + 2 (ethertype)
     *    Ethertype is 8-bits (BFP_P) at offset 12
     *    IP header len is at offset 14 of frame (lower 4 bytes).
     *       We use BPF_MSH to isolate field and multiply by 4
     *    IP fragment data is 16-bits (BFP_H) at offset  6 of IP header, 20 from frame
     *    IP protocol field is 8-bts (BFP_B) at offset 9 of IP header, 23 from frame
     *    TCP source port is right after IP header (HLEN*4 bytes from IP header)
     *    TCP destination port is two bytes later)
     */

    struct bpf_insn insns[] = {
     BPF_STMT(BPF_LD  + BPF_H   + BPF_ABS, 6+6), // Load ethertype 16-bits (12 (6+6)
                                                 // bytes from beginning)

     BPF_JUMP(BPF_JMP + BPF_JEQ + BPF_K, ETHERTYPE_IP, 0, 10),
                          // Compare to requested Ethertype or jump(10) to reject

     BPF_STMT(BPF_LD  + BPF_B   + BPF_ABS, 23), // Load protocol(=14+9 (bytes from IP))
                                                // bytes from beginning

     BPF_JUMP(BPF_JMP + BPF_JEQ + BPF_K  , Protocol, 0, 8), // Compare to requested
                                                            // or jump(8) to reject

     BPF_STMT(BPF_LD  + BPF_H   + BPF_ABS, 20), // Move 20 (=14 + 6)  We are
                                                // now on fragment offset field

     BPF_JUMP(BPF_JMP + BPF_JSET+ BPF_K, 0x1fff, 6, 0), // Bitwise-AND with 0x1FF and
                                                        // jump(6) to reject if true

     BPF_STMT(BPF_LDX + BPF_B   + BPF_MSH, IPHeaderOffset), // Load IP Header Len (from
                                                // offset 14) x 4 , into Index register

     BPF_STMT(BPF_LD  + BPF_H   + BPF_IND, IPHeaderOffset), // Skip past IP header
                                          // (off: 14 + hlen, in BPF_IND), load TCP src

     BPF_JUMP(BPF_JMP + BPF_JEQ + BPF_K  , Port, 2, 0), // Compare src port to requested
                                                       // Port and jump to "port" if true

     BPF_STMT(BPF_LD  + BPF_H   + BPF_IND, IPHeaderOffset+2),
  // Skip two more bytes (off: 14 + hlen + 2), to load TCP dest
/* port */

     BPF_JUMP(BPF_JMP + BPF_JEQ + BPF_K  , Port, 0, 1), // If port matches, ok.
                                                        //  Else reject
/* ok: */

     BPF_STMT(BPF_RET + BPF_K, (u_int)-1),      // Return -1 (packet accepted)
```

```
/* reject: */

     BPF_STMT(BPF_RET + BPF_K, 0)              // Return 0  (packet rejected)
    };

    // Load filter into program
    bpfProgram.bf_len = sizeof(insns) / sizeof(struct bpf_insn);
    bpfProgram.bf_insns = &insns[0];

    return(ioctl(fd, BIOCSETF, &bpfProgram));
}
```

To install this filter, write a small "driver" program that opens `/dev/bpfX` (by either iterating through the defined BPF devices, or arbitrarily choosing X to be one of 0, 1, 2, or 3.). The program should set the following `ioctl()`s:

- **`BIOCSETIF`**: The ioctl accepts a `struct ifreq`, though you only need to set (`strncpy`) the `ifr_name` to be the name of the underlying device (`en0`, and so on), and pass the `struct` by reference.
- **`BIOCSEESENT`**: Set this if you want to see outbound, as well as inbound frames.
- **`BIOCIMMEDIATE` or `BIOCSRTIMEOUT`**: Set this to get your `read(2)` loop to return on frame reception, or immediately.
- **`BIOCPROMISC` (optional)**: Sets promiscuous mode. Use this if you are in a shared environment (hub) or are also using VM guests in your Mac. This enables you to see traffic not intended for your host.

After setting the `ioctl()`s, you can simply start a read loop (remember the buffer size passed must match the BPF buffer len, so use `BIOCGBLEN` or `BIOCSBLEN`). Frames will be delivered as one or more `bpf_hdr` structures, up to the amount of bytes read. The structure contains a `bh_hdrlen` field, which denotes the BPF header size. Immediately following it will be the frame, of `bh_caplen` bytes.

> *Not relying on* `sizeof(struct bpf_hdr)` *is important, because of compiler alignment directives. Advancing to the next frame using* `BPF_WORDALIGN` *is also important, for the same reasons.*

If you are feeling adventurous, compile this program for iOS — you might need to copy over some OS X includes (notably, `<net/bpf.h>`). The program does, however, compile cleanly, and makes for a nice TCPdump clone (though you can always get the latter from Cydia). You can download a fully working tool, which is based on one possible solution to this exercise, from the book's companion website.

TRAFFIC SHAPING AND QOS

BSD offers, in additional to its built-in firewall, a Quality of Service (QoS) traffic shaper mechanism known as `dummynet(4)`. This mechanism relies on the `ipfw` structures described earlier in this chapter, and is in fact controlled from the system command `ipfw(8)`.

The Integrated Services Model

Defined in RFC 1633, Integrated Services (`IntSrv`) takes a different approach to QoS. Packets are still differentiated, but are not classified into logical "flows." A "flow" consists of a *traffic specification* (`TSpec`), which like the `DiffSrv` code point, is defined based on packet-specific attributes. In addition, however, a *reservation specification* (`RSpec`) defines parameters for the flow itself, namely bandwidth reservation, maximum acceptable delay, and acceptable packet loss.

BSD defines a "pipe" for integrated services. The pipe parameters can be adjusted with the `ipfw(8)` subcommand `pipe config` by specifying the number and the specific parameter — usually `bw` (bandwidth) or delay. Note, that this subcommand is not available in `ip6fw(8)`.

The Differentiated Services Model

Defined in RFC2474, Differentiated Services (`DiffSrv`) is a packet classification mechanism which assigns one of 64 "code points" to an IP packet based on properties such as its source, destination, protocol, or transport layer attributes (commonly, its ports). The 64 code points can then be used to place egress packets into one of several queues, and then route packets by queue. Each second is divided into equal shares, but an unequal number of shares is given to each queue. So, although each queue still maintains its own first-in-first-out (FIFO) ordering, the queue itself may be processed more or less frequently than others.

This approach is hence called Weighted Fair Queuing (WFQ). The fairness stems from the fact that, rather than prioritizing packets, this approach guarantees that even lowly-classified packets get treatment (although somewhat more infrequently). BSD kernels actually extend WFQ by using an improved algorithm called Worse-Case WFQ.

Differentiated services are provided by the "queue," which you can configure to hold a maximum number of packets, or overall bytes. The queues can also be set to implement the RED (Random Early Detection) or gRED (a "gentle" variant), to preemptively drop packets on specific thresholds.

Implementing dummynet

The dummynet mechanism is implemented in a single file, `bsd/netinet/ip_dummynet.c`, and uses three heaps:

- **`ready_heap`**: Used for fixed-rate pipes
- **`wfq_ready_heap`**: Used in implementing the worst-case WFQ
- **`extract_heap`**: Used to maintain packets that are intentionally delayed

These heaps are all defined in `bsd/netinet/ip_dummynet.h` (See Listing 17-26).

LISTING 17-26: THE DUMMYNET HEAP IMPLEMENTATION FROM BSD/NETINET/IP_DUMMYNET.H

```
struct dn_heap_entry {
    dn_key key ;        /* sorting key. Topmost element is smallest one */
    void *object ;      /* object pointer */
} ;
```

```
struct dn_heap {
    int size ;
    int elements ;
    int offset ; /* XXX if > 0 this is the offset of direct ptr to obj */
    struct dn_heap_entry *p ;   /* really an array of "size" entries */
} ;
```

Every interval (usually 1 ms), the `dummynet()` function is called, incrementing ticks.

Controlling Parameters from User Mode

Similar to controlling the `ipfw` mechanism, in addition to the `ipfw(8)` command, which is used to create the pipes or the queues from its rules and configure them, several `sysctl(8)`-visible parameters are available, as listed in Table 17-19.

TABLE 17-19: sysctl Parameters Pertaining to dummynet(4) Traffic Shaping

NET.INET.IP.DUMMYNET.*	DEFAULT VALUE	USED FOR
`hash_size`	64	Default value of buckets in queues and flows.
`red_avg_pkt_size`	512	Average size of a packet.
`red_max_pkt_size`	1500	Maximum size of a packet (as per MTU).
`red_lookup_depth`	256	Accuracy of computing the RED algorithm.
`debug`	0	Enables debug output.
`expire`	1	Automatically removes dynamic pipes if they become idle (that is, no traffic).
`max_chain_len`	16	Maximum number of pipes or queues per bucket. They are automatically removed upon `max_chain_len x hash_size`.
`searches` *`search_steps`*	0 0	Number of queue searches and search steps.
`ready_heap` *`extract_heap`*	N/A	Current sizes of ready and extract heaps.

*Parameters in italic are not specified in the manual pages.

SUMMARY

This chapter detailed, in great depth, the inner workings of the XNU network stack. Though closely resembling that of BSD, the XNU stack has some notable extensions in its implementation. The stack has a multitude of filtering mechanisms at every one of its layers (sockets, IP and interfaces), as well as support for QoS. Most importantly, it is "pluggable" in the sense that kernel extensions can register their own callbacks with specific protocol implementations, as is in fact done by `IONetworkingFamily` and friends.

The next chapters will discuss how these kernel extensions are created and handled. Chapter 18 explains the basic concepts of structure of all extensions, and Chapter 19 devotes itself to those of a specific type, IOKit.

REFERENCES AND FURTHER READING

1. Stevens, "Sockets and XTI programming," Vol. 1
2. Stevens, "TCP/IP Illustrated," Vol. 1–3
3. Kong, Joseph. *Designing BSD Rootkits*: *An Introduction to Kernel Hacking*. No Starch Press, 2007
4. Article TS1629, "Well known TCP and UDP ports used by Apple software products," `http://support.apple.com/kb/TS1629`
5. RFC1035 — "Domain Names – Implementation and Specification" `http://www.ietf.org/rfc/rfc1035.txt`
6. Apple Developer. Apple Filing Protocol Reference — `https://developer.apple.com/library/mac/#documentation/Networking/Reference/AFP_Reference/Reference/reference.html`
7. Network-cmds and the route(8) command — `http://opensource.apple.com/source/network_cmds/network_cmds-356.8/route.tproj/route.c`
8. Apple's EAPOL implementation — `http://opensource.apple.com/tarballs/eap8021x/`
9. Esser, Stefan "iOS Kernel Exploitation," `https://media.blackhat.com/bh-us-11/Esser/BH_US_11_Esser_Exploiting_The_iOS_Kernel_WP.pdf`
10. RFC2367 - Key Management Sockets http://www.ietf.org/rfc/rfc2367.txt
11. The Kame Project — "IPv6 and IPsec stack for use in BSD-based operating systems" `http://www.kame.net`
12. RFC3056 — "Connection of IPv6 Domains via IPv4 Clouds" `http://www.ietf.org/rfc/rfc3056.txt`
13. RFC4380 — "Teredo" `http://www.ietf.org/rfc/rfc4380.txt`
14. Miredo for OS X: http://www.remlab.net/miredo/
15. RFCGI — RFC2893 — "Transition Mechanisms for IPv6 Hosts and Routers" `http://www.ietf.org/rfc/rfc2893.txt`
16. Network-cmds and the netstat(8) command — `http://opensource.apple.com/source/network_cmds/network_cmds-356.8/netstat.tproj/inet.c`
17. Apple Developer, "Network Kernel Extensions Programming Guide," `http://developer.apple.com/library/mac/documentation/Darwin/Conceptual/NKEConceptual/NKEConceptual.pdf`

18. Halvorsen & Clarke "iOS and OS X Kernel Programming" Apress, 2011

19. Apple Developer. TCPLogNKE sample code — `https://developer.apple.com/library/mac/#samplecode/tcplognke/Introduction/Intro.html#//apple_ref/doc/uid/DTS10003669`

20. Hansteen, Peter, *The Book of PF*: *A No-Nonsense Guide to the OpenBSD Firewall*, Second Edition. No Starch Press, 2010

21. CVE-2010-3830, `http://cve.mitre.org/`

22. Sogeti, ESEC Labs `http://esec-lab.sogeti.com/post/2010/12/09/CVE-2010-3830-iOS-4.2.1-packet-filter-local-kernel-vulnerability`

23. Machiavelli — `http://www.blackhat.com/presentations/bh-usa-09/DAIZOVI/BHUSA09-Daizovi-AdvOSXRootkits-SLIDES.pdf`

24. McCanne and Van Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture," `http://www.tcpdump.org/papers/bpf-usenix93.pdf`

18

Modu(lu)s Operandi — Kernel Extensions

XNU provides a rich ecosystem of a kernel, having all the necessary services — scheduling, memory management, I/O, and more. Yet, no kernel can completely accommodate the vast range of hardware and peripheral devices available. Nor can any kernel, even monolithic ones, claim to be fully complete.

Enter: kernel extensions. Like shared libraries or DLLs in user mode, these are kernel modules, which may be dynamically inserted or removed on demand, often from user mode. XNU, in both OS X and iOS, makes use of modules to load its various device drivers, and to augment kernel functionality with entirely self-contained subsystems.

This chapter explores the mechanics of kernel extensions. We first discuss the design perspective, and then delve into intrinsic details of the various APIs. The chapter provides also provides insight into the undocumented happenings behind the APIs.

EXTENDING THE KERNEL

Virtually every contemporary operating system architecture acknowledges that, although a kernel is usually self-contained and must be able to provide the full set of APIs expected by user mode, crafting a kernel that is statically linked is virtually impossible. Such a kernel would imply a very rigid structure, which would not be extensible in any way: That, which was compiled in time, would be available, yet no additional functionality could be added.

With the multitude of devices available and the many offerings of new buses and device classes, compiling a single kernel that would contain all the necessary device drivers is unfeasible. Additionally, some operating system designs allow third-party developers access to extend and enhance their kernels or otherwise allow the insertion of code into kernel mode.

As necessity is the mother of invention, extensibility is that of modular design. Just as user mode has DLLs (in Windows) or shared objects (in UNIX), so does kernel mode in the form of kernel modules, or — in XNU parlance — kernel *extensions*. Called kexts for short, kernel extensions are a fundamental building block of XNU as much as the core itself. In fact, it is not uncommon to find more kernel-mode code resulting from module insertion than the original kernel core.

Although the nomenclature might be different, the idea behind kexts is exactly the same as that of Windows' `.sys` files (in `%systemroot%\system32\drivers`) and Linux's `.ko` files (usually in `/lib/modules or elsewhere`). All three file types are relocatable code that is dynamically linked with specific symbols the kernel sees fit to export. Kexts require only one well-known entry point, which usually handles all the initialization tasks the extension requires, and from that point can execute any code the developer wants.

A kext runs in kernel mode, and therefore has full access to kernel space. The developer can use any function that the kernel defines as exportable and even functions that are defined private — although the latter usually involve some form of hacking or reverse engineering. Global kernel variables and structures may also be queried and even set, making kexts highly popular for all sorts of kernel-level development. Profiling, system call hooking, and other functionality can be achieved in kernel mode.

Because kernel modules offer so much power, they pose an even greater risk. If the kernel is set to accept code of foreign origin, determining the intent — or malicious intent — of such code prior to actual insertion is impossible. Furthermore, once the code is loaded into the kernel, it is effectively the same, for all intents and purposes, as code from the kernel proper. This means the stability, and, even more so, the security of the entire operating system can be compromised. Indeed, most modern-day malware comes in the form of malicious modules, also known as "rootkits."

In iOS, in particular, there is another dimension of risk. Apple seems to have no desire whatsoever to open up the kernel development space to anyone but its own cadre. As a system, iOS is hardened in both user and kernel mode to discourage any type of modification. So, although kexts are used extensively to provide support for the various i-Devices, they are "fused" into the kernelcache by Apple when the iOS is built for each device (although kexts do load on the fly, from the kernelcache).

Securing Modular Architecture

Because a modular architecture harbors both significant benefits as well as huge risks, contemporary operating systems continue to allow and promote it, but impose certain limitations on its use, lest it be subverted for malicious means. There are two approaches for securing the architecture.

Code Signing

Code signing is the preferred approach and is the standard adopted by most systems. A good example is Windows, which (as of Windows Vista in its 64-bit edition) prevents any type of driver from loading unless it possesses a valid digital signature. Prior to transferring control to the module entry point, the kernel validates the signature on the code in the form of an attached certificate. The certificate must be signed with a private key, whose public key is known to the kernel, or by a chain of trust leading to such a key.

Code signing cannot vouch for code purity of purpose, but it can validate the origin of the code. Because signing the code involves the developer identifying to the signer, any attempted malware — once caught — would disqualify said developer, and would provide liability for any damages.

Apple uses code signing ubiquitously in iOS, yet signs no code but its own. The validation key is embedded deep in ROM, and from the early stages of iBoot, code that is not signed by Apple cannot be loaded. This makes it impossible to tamper with an iOS software update, which, (as was demonstrated in Chapter 5), is but a simple zip file. Any attempted patching of the update will result in the update being rejected. Indeed, only by patching the signature check in pre-A5 i-Devices can custom firmware images be loaded onto the device.

Pre-Linking

Pre-linking is the approach used by Apple in OS X and iOS. Rather than loading the kernel, and then loading the kexts in some order, the boot loader instead loads a *kernelcache file*. This file contains the kernel, pre-linked with select extensions. The result is essentially the same as having had the kernel dynamically load the extensions, but it offers two advantages:

- Loading time is much faster, because the process of dynamic linking involves resolving symbols in both the kernel and the module during runtime. Pre-linking allows the resolving to be done once, and the kernel image to be loaded with the modules already in, when the link addresses have been fully resolved.
- The kernelcache may be signed, and even encrypted (as is the case on iOS). Once the kernelcache is loaded, all further kext loading could potentially be disabled (though in practice, it isn't). This would ensure that no code can find a legitimate way into the iOS kernel.

> *As hardened as it is, even the iOS kernel has been subverted — a necessary step in the jail-breaking process, which is discussed in Chapter 5. This, however, was done by injecting code into the kernel due to a security vulnerability, and not by any "official" mechanism the kernel extensions provide.*

KERNEL EXTENSIONS (KEXTS)

When not linked into a kernelcache, kexts can be found in their standalone form populating `/System/Library/Extensions`. The vast majority of the kexts here are device drivers, which are detailed in depth in Chapter 19. The kexts found in this directory vary depending on the Mac model. Bear in mind, also, that not all of these kexts may be in use. To see which ones are actively loaded, use the `kextstat(8)` command, shown in Output 18-1.

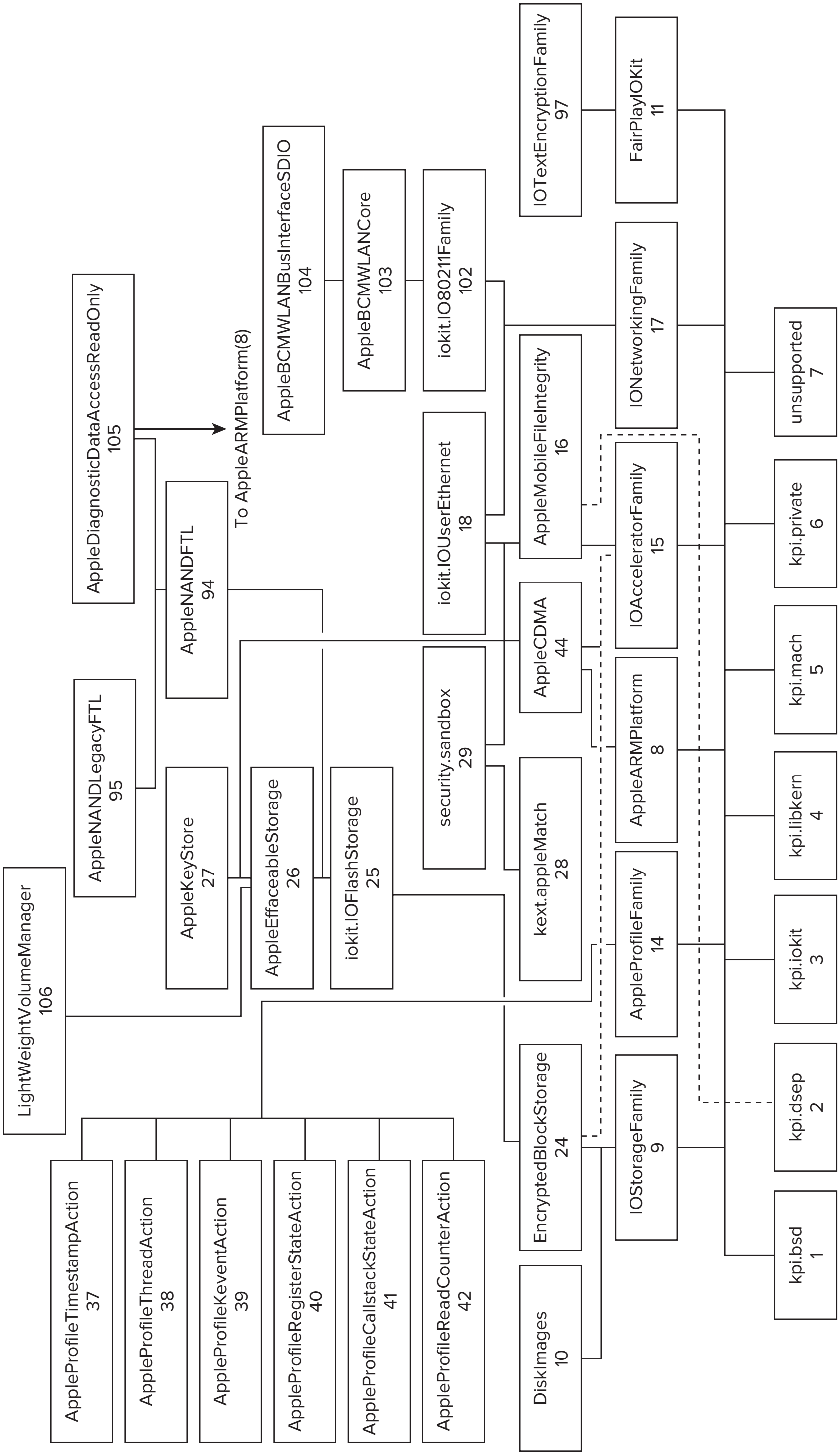
OUTPUT 18-1: Output of kextstat(8) from a Lion OS

```
Index Refs Address          Size     Wired    Name (Version) <Linked Against>
1  82 0xffffff7f80742000 0x683c   0x683c   com.apple.kpi.bsd (11.0.1)
2   6 0xffffff7f8072e000 0x3d0    0x3d0    com.apple.kpi.dsep (11.0.1)
3 106 0xffffff7f8074c000 0x1b9d8  0x1b9d8  com.apple.kpi.iokit (11.0.1)
4 111 0xffffff7f80738000 0x9b54   0x9b54   com.apple.kpi.libkern (11.0.1)
5  99 0xffffff7f8072f000 0x88c    0x88c    com.apple.kpi.mach (11.0.1)
6  33 0xffffff7f80730000 0x4938   0x4938   com.apple.kpi.private (11.0.1)
7  55 0xffffff7f80735000 0x22a0   0x22a0   com.apple.kpi.unsupported (11.0.1)
8  21 0xffffff7f809bc000 0x7000   0x7000   com.apple.iokit.IOACPIFamily (1.4)<7 6 4 3>
9 30 0xffffff7f80821000 0x1d000  0x1d000  com.apple.iokit.IOPCIFamily (2.6.5)<7 6 5 4 3>
...
82  2 0xffffff7f809c3000 0xc000   0xc000  com.apple.driver.AppleSMC (3.1.1d2)<8 7 5 4 3>
...
96  0 0xffffff7f812b9000 0x5000   0x5000  com.apple.Dont_Steal_Mac_OS_X (7.0.0)<82 7 ...
...
```

`kextstat(8)` *looks a little bit different on Lion than on previous versions of OS X. This is due to two reasons:*

- *The built-in kernel APIs in Lion have their VMSize and Wired fields correctly filled. On previous versions, their values were left at zero.*
- *Lion has fewer kernel APIs. Prior to Lion, the kernel exposed the (now obsolete)* `com.apple.kernel.*` *APIs for kexts to rely on, but these were declared deprecated as of Tiger (10.4), and have finally been removed as the feline evolved (though they are still present in 32-bit kernels and in iOS).*
- *The cydia version of kextstat (if you try it on iOS) is woefully broken, as it relies on deprecated APIs* (`kmod_get_info`) *which are unavailable in iOS. The book's companion websites offers a version that works well. But — more on that later.*

Kexts may be layered on top of one another. As Output 18-1 shows, each kext has a load index and a "references" field. The latter is used to determine how many dependents this kext has, and the former serves as an index to identify the kext in the list to its dependents. The values inside the angle brackets in each kext show the kexts it relies on, by index. A somewhat simplified and partial graphical representation of kext ordering is shown in Figure 18-1.



* -This graph is simplified by omitting dependencies which exist both directly and indirectly. That is, if a kext is dependent directly on another, but also independently (through another kext) on the same kext, the direct dependence is omitted. Even with this simplification, the graph is so big some kexts (particularly those which rely on `AppleARMPlatform`, for hardware) have been omitted. Lines are differently styled or broken if they do not intersect (i.e. on different planes). Full list of kexts is in Output 18-5.

FIGURE 18-1: Partial simplified representation of kexts in iOS 5

The first seven (or before Lion, twelve) load indices, which make up the foundation in Table 18-1, aren't real kexts; rather, they are "pseudo-kexts," or kernel built-in components. Their component version is the same as the Darwin version.

TABLE 18-1: Kernel Interfaces

KERNEL PROGRAMMING INTERFACE	REPRESENTS
`com.apple.kpi.bsd`	The kernel's BSD personality. This supersedes `com.apple.kernel.bsd`.
`com.apple.kpi.dsep`	Mandatory Access Control (MAC) Framework. This is a new interface, whose primary clients are the Sandbox.kext, FSCompression, quarantine (in OS X) and AppleMobileFileIntegrity (in iOS).
`com.apple.kpi.iokit`	The I/O Kit framework. This supersedes `com.apple.kernel.iokit`.
`com.apple.kpi.libkern`	The kernel runtime library. This supersedes `com.apple.kernel.libkern`.
`com.apple.kpi.mach`	The kernel's Mach personality. This supersedes `com.apple.kernel.mach`.
`com.apple.kpi.private`	Kernel internal APIs, which are not meant to be exported to non-Apple kexts.
`com.apple.kpi.unsupported`	Unsupported/deprecated APIs.

You can find all the pseudo-kexts in the `/System/Library/Extensions/System.kext/PlugIns` directory, yet they contain no code. In fact, they contain only one section — a symbol table — because their code is already implemented in the kernel. These are often referred to as the Kernel Programming Interfaces (KPIs). The XNU sources (`libsa/bootstrap.cpp`) also list four other kexts:

- `com.apple.iokit.IONVRAMFamily`
- `com.apple.driver.AppleNMI`
- `com.apple.iokit.IOSystemManagementFamily`
- `com.apple.iokit.ApplePlatformFamily`

Yet these, too, aren't actual kexts, and their respective directories contain only an `Info.plist`.

Kexts declare their dependency on other kexts — pseudo or real — in the `OSBundleLibraries` property of their main property list, as you will see in the next section.

A particularly intriguing kext is "`Dont Steal Mac OS X.kext`", also commonly referred to as DSMOS, shown earlier in Output 18-1. This kext is untouchable — its accompanying (intimidating)

LICENSE file strictly forbids any tampering with, disabling, or destroying it. Many a hackintosh has had its boot process delayed inevitably "waiting for DSMOS." For obvious reasons, this book cannot detail much about the DSMOS kext; suffice to say that it is used in decrypting code from various binaries, like the Finder, as discussed in Chapter 3. As noted in Chapter 11, which discussed Mach virtual memory internals, Apple has modified Mach and added its own memory pager (apple_protected_pager) to deal with DSMOS-protected memory, and that part remains open source. iOS doesn't have this module, but uses the IOTextEncryptionFamily (and, indirectly FairPlayIOKit) instead.

Kext Structure

Kexts are bundles, and as such follow the generic bundle layout: A kext directory has a single subdirectory, Contents/, in which you can find the files shown in Table 18-2.

TABLE 18-2: Files in the Contents/ Subdirectory

FILE/DIRECTORY	CONTAINS
CodeDirectory	Code directory file for the kext
CodeRequirements	Code requirement set for the kext
CodeResources	Code resources XML file specifying hashes and rules for files in kext
CodeSignature	Code signature for kext — usually contains Apple's digital certificate
Info.plist	Bundle manifest property list
MacOS	Directory containing actual kext binary — a file of type BUNDLE (Mach-O type 8) or KEXTBUNDLE (Mach-O type 11) for 64-bit
_CodeSignature	Directory containing the Code* files, which are actually symbolic links to this directory
version.plist	Kext version information, in a property list

Somewhat infrequently, a kext may contain other, related kexts — as in the case of kexts implementing IORegistry families (most IO*Family.kext). In those cases, the related kexts are nested in a PlugIns subdirectory. Also in some cases (e.g. IOSCSIArchitectureModelFamily.kext, webdavfs.kext, or ufs.kext), kexts may contain various resources — internationalization files, related user-mode binaries, and even icons. As you can expect, those are all found in a Resources subdirectory.

Like any bundle, the kext's Info.plist property list is of special importance. It is mandatory, and contains specific fields without which the kext cannot be loaded. Table 18-3 shows the fields mandatory in any kext:

TABLE 18-3: Mandatory Fields in Kext Plists

PLIST PROPERTY	USED FOR
`CFBundleExecutable`	Identifying the actual kext executable inside the bundle. This is, by convention, a file in the `MacOS/` subdirectory, with the same name as the kext itself.
`CFBundleIdentifier`	Uniquely identifying the kext name during runtime. This is the standard reverse DNS notation. Apple recommends `com.company.driver.*` for an I/O Kit driver, and `com.company.kext` for a generic kext.
`CFBundleVersion`	Kext version number, in the form of *Major.Minor.Fix.*
`OSBundleLibraries`	Required kernel libraries and other kexts on which this one depends.

The `Info.plist` can also specify several additional, optional properties, as shown in Table 18-4:

TABLE 18-4: Optional Fields in Kext Plists

PLIST PROPERTY	USED FOR
`OSBundleAllowUserLoad`	Boolean specifying that non-privileged users can load this kext. The default is `FALSE`.
`OSBundleCompatibleVersion`	Specifying which API versions this kext exports. This is the "other side" of `OSBundleLibraries`, as other kexts will specify this version to link to.
`OSBundleRequired`	Specifying this kext is required to mount the root filesystem on whatever device (`Root`), on a local device (`Local-Root`) or a network device (`network-root`). May also specify that this kext is required for console support (`console`), or even when booting `–x` (`Safe-Boot`).

It's not uncommon to find `OSBundle*` properties further defined for specific architecture by appendix suffixes (in the case of OS X `_i386` and `_x86_64`). For I/O Kit drivers, the `Info.plist` contains a host of other properties (including the mandatory `IOKitPersonalities`), which are described in Chapter 19.

Kext Security Requirements

Because kexts contain code that is loaded into kernel memory, extra security considerations must be enforced to make sure that any arbitrary and potentially malicious code will not be accidentally loaded.

The requirements on kexts are thus:

- Kexts must be owned by the `uid` of root, and the `gid` of wheel.
- Permissions on the directories must be at most 755 — that is, `rwxrwxr-x`.
- Any files in the kext must be at most 644 (`rw-r--r--`).

Working with Kernel Extensions

Mac OS X provides several handy utilities to manipulate and provide information about kernel extensions, as shown in Table 18-5:

TABLE 18-5: Kext-related Commands

COMMAND	USEAGE
`kextd`	Dynamically loads kexts from user-space
`kextfind`	Query kext by myriad properties and criteria. Simulates operation of `kextd`, as it looks up kexts for dynamic loading
`kextlibs`	Resolves kext dependencies
`kextload`	A simple kext loader
`kextunload`	A simple kext unloader
`kextutil`	(Snow Leopard and later): The more advanced version of `kextload`, with far more options

These tools will be demonstrated in a simple exercise to create kexts.

Kernelcaches

Kernelcaches play an important part in both OS X and iOS. In OS X, they are used to speed up the boot process by providing a complete kernel, optimized for the specific platform the OS is executing in, with all the drivers pre-loaded. In iOS, they contain the only kexts that the kernel will load, and no others. This makes the iOS kernel far more secure and tamper resistant.

Kernelcaches follow the same general structure on both platforms, but are implemented a little bit differently in OS X and iOS, as shown in Table 18-6.

TABLE 18-6 Kernelcache Implementation

OS	/SYSTEM/LIBRARY/CACHES/..	CONTAINS
OS X	`com.apple.kext.caches/Startup`	Mach-O binary, potentially fat, with `complzss` beginning at relative offset 384
iOS	`com.apple.kernelcaches/kernelcache`	Kernelcache in IMG3 encrypted form, opening to a `complzss`, as in the preceding

The iOS kernelcache format (IMG3) and the simple `complzss` compression scheme were both previously discussed under "iOS Boot Images." in Chapter 6.

To unpack a kernelcache, you must first get rid of excess headers: On OS X, these are usually the fat header (if the kernelcache is a multi-architecture `i386/x86_64` binary) and the `lzss` compression. On iOS the kernelcache is a thin binary — only the ARM architecture is present. However, the kernelcache is encrypted, and you therefore must apply a precursor step of decrypting the cache, if you can obtain the IV and Key. This is shown in Output 18-2:

OUTPUT 18-2: Expanding a kernelcache

```
morpheus@Minion(/) $ cd /System/Library/Caches/com.apple.kext.caches/Startup
morpheus@Minion(.../com.apple.kext.caches/Startup)$ file kernelcache
kernelcache: Mach-O universal binary with 2 architectures
kernelcache (for architecture x86_64):  data
kernelcache (for architecture i386):    data

morpheus@Minion(.../com.apple.kext.caches/Startup)$ more kernelcache
"kernelcache" may be a binary file.  See it anyway? y
<CA><FE><BA><BE>^@^@^@^B^A^@... ^@^@^@^C^@<9C><90><84>^@<90>\<BC>^@^@^@^@complzss<AD>..

morpheus@Minion (.../Startup)$ lipo -thin x86_64 kernelcache /tmp/thincache

morpheus@Minion (.../Startup)$ more /tmp/thincache
complzss<AD><D2>…

morpheus@Minion (.../Startup)$ complzss -o 384 /tmp/thincache> /tmp/uncompressed_cache
morpheus@Minion (.../Startup)$ file /tmp/uncompressed_cache
/tmp/uncompressed_cache: Mach-O 64-bit executable x86_64
morpheus@Minion (.../Startup)$ ls -l /tmp/uncompressed_cache /mach_kernel
-rw-r--r--  1 root  wheel  23851008 Sep  4 19:46 /tmp/uncompressed_cache
-rw-r--r--@ 1 root  wheel  15564456 May  7 07:23 /mach_kernel
```

Recall, the `0xCAFEBABE` is the fat header of the file. Soon after it is the `complzss` header, which in this case spans 384 bytes. At that offset, the compressed image begins, which can be expanded into a thin binary.

If you look at the binary and compare it to your `mach_kernel`, as in the example in Output 18-2, you will see a significant difference in size. This is the size of all the kernel extensions loaded into the `__PRELINK_TEXT` segment. Whereas the `mach_kernel` in the root has an empty segment, the kernelcache makes use of this segment by putting all the necessary kernel extensions in it. Using otool once more, this time to dump the `PRELINK_TEXT` segment (`otool -s __PRELINK_TEXT __text`), reveals the segment has additional Mach-O binaries, the kexts, loaded in. You can recognize the kexts by their Mach-O signature — `0xFEEDFACE` (32-bit) or `0xFEEDFACF` (64-bit)[1] as shown in Output 18-3:

OUTPUT 18-3: Isolating kexts in the kernelcache's PRELINK_TEXT section.

[1]On Intel architecture, remember that endian-ness makes the signature appear to be `ce fa fe ed` or `cf fa fe ed`, and therefore you should grep accordingly.

```
morpheus@Ergo(/)$ otool -s __PRELINK_TEXT __text IOS-5.0.0b5.kernel | grep feedface
80347000        feedface 0000000c 00000009 0000000b
80348000        feedface 0000000c 00000009 0000000b
8034c000        feedface 0000000c 00000009 0000000b
80363000        feedface 0000000c 00000009 0000000b
8036b000        feedface 0000000c 00000009 0000000b
80371000        feedface 0000000c 00000009 0000000b
80377000        feedface 0000000c 00000009 0000000b
80378000        feedface 0000000c 00000009 0000000b
8037a000        feedface 0000000c 00000009 0000000b
803a2000        feedface 0000000c 00000009 0000000b
… total of 137 packed kernel extensions..
```

But how does the kernel know just what these kexts are? You saw that in a standalone form, each kext as a bundle contains a property list file, `Info.plist`. The same applies for a kernelcache, but in this case, the `Info.plist` files are packed separately in a `__PRELINK_INFO __info` segment. If you use otool on this segment, you will see it is ASCII text. It also is not just any text, but a massive `Plist`, containing an array of `dict`s, each representing one of the kexts loaded. If you use the book's companion ***jtool*** (or `segedit(1)`) to extract the `PRELINK_INFO` segment from the iOS 5 decrypted kernel, you would see something similar to Output 18-4:

OUTPUT 18-4: kextcache __PRELINK_INFO segment, restored to XML format

```
morpheus@Ergo (../iOS)$ jtool -e PRELINK_INFO kernel.5.0.1.iPod4
Processing kernel.5.0.1.iPod4
Mach-O 32-bit executable for ARMv7; 11 load commands spanning 2076 bytes
Extracting segment@0x10420224, 523911 bytes into kernel.5.0.1.iPod4.__PRELINK_INFO
morpheus@Ergo (../iOS)$ more PRELINK_INFO kernel.5.0.1.iPod4
<dict><key>_PrelinkInfoDictionary</key>
   <array>
     <dict>
       <key>CFBundleName</key><string>MAC Framework Pseudoextension</string>
       <key>_PrelinkExecutableLoadAddr</key><integer size="64">0x80346000</integer>
       <key>_PrelinkKmodInfo</key><integer ID="5" size="32">0x0</integer>
       <key>_PrelinkExecutableSize</key><integer size="64">0x28c</integer>
       <key>CFBundleDevelopmentRegion</key><string ID="7">English</string>
       <key>CFBundleVersion</key><string>11.0.0</string>
       <key>_PrelinkExecutableSourceAddr</key><integer size="64">0x80346000</integer>
       <key>CFBundlePackageType</key><string>KEXT</string>
       <key>CFBundleShortVersionString</key><string>11.0.0</string>
       <key>OSBundleCompatibleVersion</key><string>8.0.0b1</string>
       <key>OSKernelResource</key><true/>
       <key>_PrelinkExecutableRelativePath</key><string>MACFramework</string>
       <key>CFBundleInfoDictionaryVersion</key><string ID="15">6.0</string>
       <key>CFBundleExecutable</key><string>MACFramework</string>
       <key>OSBundleAllowUserLoad</key><true/>
       <key>CFBundleIdentifier</key><string>com.apple.kpi.dsep</string>
       <key>CFBundleSignature</key><string ID="18">????</string>
       <key>OSBundleRequired</key><string>Root</string>
```

continues

OUTPUT 18-4 *(continued)*

```
        <key>CFBundleGetInfoString</key>
            <string>MAC Framework Pseudoextension, SPARTA Inc,11.0.0</string>
        <key>_PrelinkBundlePath</key>
      <string>/System/Library/Extensions/System.kext/PlugIns/MACFramework.kext</string>
       <key>_PrelinkInterfaceUUID</key><data>d1F0yq5vQTeuZGj2Y5s5dg==</data>
</dict>
<dict>
   <key>CFBundleName</key><string>Private Pseudoextension</string>
   <key>_PrelinkExecutableLoadAddr</key><integer size="64">0x80347000</integer>
   <key>_PrelinkKmodInfo</key><integer IDREF="5"/>

       … (output truncated - there's over 520KB of XML) …
```

Note that the prelinked `Info.plist` sections contain additional keys that are not present (and not needed) in standalone kexts. These are easily identifiable because of the `_Prelink` prefix. They are not formally documented by Apple, but their use is as shown in Table 18-7:

TABLE 18-7: Plist File Properties

PLIST PROPERTY	USED FOR
`_PrelinkExecutableSourceAddr`	The address in memory in which this kext can be found when loading the kernel. This is the address in which the kext's Mach-O header can be expected from the `__PRELINK_TEXT` section (compare with the output of otool).
`_PrelinkExecutableLoadAddr`	The address in memory where this kext will be loaded. In the case of a prelinked kernel, equating this value with the source address just makes sense.
`_PrelinkExecutableSize`	Size of the kext in bytes.
`_PrelinkExecutableRelativePath`	Where this kext would be, relative to the `_PrelinkBundlePath`.
`_PrelinkBundlePath`	Where this kext would be, had it been on disk.
`_PrelinkInterfaceUUID`	Used for the core pseudo-extensions. A Base 64 – encoded unique identifier.

Kernelcaches are created on OS X dynamically — and the root directory still contains a copy of `mach_kernel`. On iOS, however, the kernelcache is one of the files provided by Apple. Therein also lies the difference between the iOS distributions of the various devices: The kexts required for a CDMA iPad, for example, differ from those of a GSM iPhone.

To view a list of kexts in the iOS kernelcache for yourself, you can run the decache shell script provided on the book's website — provided you have the decrypted, decompressed kernelcache. It will provide you information on the kexts, as well as selectively display their properties.

The iPod4, 1 kernel will list something similar to what's shown in Output 18-5, with some 143 pre-linked extensions in all:

OUTPUT 18-5: Output of decache on the decompressed iPod 4,1 kernelcache of iOS 5.0

```
morpheus@Ergo (/iOS)$ Tools/decache kernels/iPod4,1_5.0_9A334/kernelcache
MAC Framework Pseudoextension (System.kext/PlugIns/MACFramework.kext)
Private Pseudoextension(System.kext/PlugIns/MACFramework.kext)
I/O Kit Pseudoextension (System.kext/PlugIns/IOKit.kext)
Libkern Pseudoextension (System.kext/PlugIns/Libkern.kext)
BSD Kernel Pseudoextension (System.kext/PlugIns/BSDKernel.kext)
AppleFSCompressionTypeZlib (AppleFSCompressionTypeZlib.kext)
Mach Kernel Pseudoextension (System.kext/PlugIns/Mach.kext)
Unsupported Pseudoextension (System.kext/PlugIns/Unsupported.kext)
I/O Kit USB Family (IOUSBFamily.kext)
I/O Kit Driver for USB User Clients(IOUSBFamily.kext/PlugIns/IOUSBUserClient)
I/O Kit Storage Family (IOStorageFamily.kext)
AppleDiskImageDriver (IOHDIXController.kext)
AppleDiskImagesKernelBacked (IOHDIXController.kext/PlugIns/AppleDiskImagesKernelBacked)
FairPlayIOKit (FairPlayIOKit.kext)
AppleARMPlatform (AppleARMPlatform.kext)
AppleVXD375 (AppleVXD375.kext)
IOSlaveProcessor (IOSlaveProcessor.kext)
IOP_s5l8930x_firmware (IOSlaveProcessor.kext)
AppleDiskImagesUDIFDiskImage(IOHDIXController.kext/PlugIns/AppleDiskImagesUDIFDiskImage)
..
```

Note, not all kexts may necessarily be loaded (though most are). You can use the **`jkextstat`** tool, described later in this chapter, to see which kexts are actively loaded.

Multi-Kexts

Kernelcaches are just one of two forms of pre-linking available in OS X and iOS. The other is known as a multi-kext archive, or *mkext*. This file is really just an archive of two or more kexts, like a kernelcache, but without the kernel. Mkexts are unidentifiable by "file" and other utilities, but a visible ASCII "MKXTMOSX" signature in the first line of the binary format makes them stand out from other binaries. This header is documented in `libkern/mkext.h`, as shown in Listing 18-1:

LISTING 18-1: The mkext header, from libkern/mkext.h

```
* Core Header
*
* All versions of mkext files have this basic header:
*
* - magic & signature - always 'MKXT' and 'MOSX' as defined above.
* - length - the length of the whole file
* - adler32 - checksum from &version to end of file
* - version - a 'vers' style value
* - numkexts - how many kexts are in the archive (only needed in v.1)
* - cputype & cpusubtype - in version 1 could be CPU_TYPE_ANY
```

continues

LISTING 18-1 *(continued)*

```
*     and CPU_SUBTYPE_MULTIPLE if the archive contained fat kexts;
*     version 2 does not allow this and all kexts must be of a single
*     arch. For either version, mkexts of specific arches can be
*     embedded in a fat Mach-O file to combine them.
```

Mac OS X provides a "kextcache" tool to maintain kernelcaches and mkext files alike. Using `kextcache mkextunpack`, you can list or unarchive an `mkext`.

A Programmer's View of Kexts

From the programmer's perspective, a kext is just a kernel-mode object file, linking with the kernel-mode, rather than user-mode libraries. This means that many familiar functions from `<unistd.h>` and `<stdlib.h>` are no longer available. Also, kernel-mode brings other constraints — primarily in the form of severe memory restrictions, because kernel memory is, by default, wired memory and consumes physical RAM.

The most severe restriction kernel mode imposes is in system stability. Creating a kext is the easy part — the difficulty is in how to correctly code a kext, because even the most minor transgression in a kext can lead to a kernel panic. In kernel mode, no safety net exists like there is in user mode, and no well-defined process bounds to contain errors. Rather than kill an offending kernel thread, the kernel opts for harakiri, and kills itself.

Take out the warnings, however, and what remains is a relatively simple and straightforward process, involving the following steps:

1. Start XCode and choose Generic Kernel Extension from the System Plug-ins pane.
2. XCode defines the kext entry and exit points for you automatically. Both have the same prototype. The generated code will look something like Listing 18-2:

LISTING 18-2: The skeleton code generated for a new kernel extension

```
#include <mach/mach_types.h>

kern_return_t SampleKext_start(kmod_info_t * ki, void *d);
kern_return_t SampleKext_stop(kmod_info_t *ki, void *d);

kern_return_t SampleKext_start(kmod_info_t * ki, void *d)
{
    return KERN_SUCCESS;
}

kern_return_t SampleKext_stop(kmod_info_t *ki, void *d)
{
    return KERN_SUCCESS;
}
```

The two arguments are generally treated as opaque, though the `kmod_info_t` can prove quite useful if you want to enumerate all the kexts in the system (or do more insidious things like hide your kext).

3. Edit the `Info.plist` file either directly or through the XCode `plist` editor (the `plist` is under Supporting Files).
4. Compile, either through the GUI or, if you prefer CLI, using `xcodebuild(1)`. Although this command has many arguments, you can opt for the defaults, or selectively build for specific targets (`-target`) or configurations (`-configuration`).

Kexts can link with the `Kernel.Framework`, which is an empty framework (no binary) containing the kernel headers (exported from XNU during the build stage). In addition, the `Resources/` directory of this framework contains text files listing the supported KPIs for each architecture (including ARM).

Kernel Kext Support

Kexts are a unique part of XNU, because they represent a significant component that is neither part of Mach nor of BSD. Additionally, whereas most of the kernel is C, kext handling is performed in a portion of XNU which is C++. The same holds true for I/O Kit, which rests on kext support, as well.

Mach kmod Support

XNU's Mach layer was extended to support kernel modules. While the Mach layer is unaware of kexts, it does support a `kmod` object, representing a kernel module. Listing 18-3 shows `kmod_info`, defined in `osfmk/kern/kmod.h`.

LISTING 18-3: The definition of the kmod_info_t , which abstracts kexts

```
#define KMOD_MAX_NAME    64

typedef struct kmod_info {
    struct kmod_info  * next;
    int32_t             info_version;           // version of this structure
    uint32_t            id;
    char                name[KMOD_MAX_NAME];
    char                version[KMOD_MAX_NAME];
    int32_t             reference_count;        // # linkage refs to this
    kmod_reference_t  * reference_list;         // who this refs (links on)
    vm_address_t        address;                // starting address
    vm_size_t           size;                   // total size
    vm_size_t           hdr_size;               // unwired hdr size
    kmod_start_func_t * start;
    kmod_stop_func_t  * stop;
} kmod_info_t;
```

It is this `kmod_info_t`, which every kext gets as a parameter for its entry point. When a kext is created, XCode initializes a `kmod_info_t` for the kext, using a macro, `KMOD_DECL_EXPLICIT`, which it generates in the XCode `DerivedData/` directory under `<moduleName>_info.c` file. This is shown in Listing 18-4:

LISTING 18-4: Automatically generated info for kexts

```
#include <mach/mach_types.h>

extern kern_return_t _start(kmod_info_t *ki, void *data);
extern kern_return_t _stop(kmod_info_t *ki, void *data);
__private_extern__ kern_return_t sampleKext_start(kmod_info_t *ki, void *data);
__private_extern__ kern_return_t sampleKext_stop(kmod_info_t *ki, void *data);

__attribute__((visibility("default")))
   KMOD_EXPLICIT_DECL(com.technologeeks.osx.sampleKext, "1.0.0d1", _start, _stop)
__private_extern__ kmod_start_func_t *_realmain = sampleKext_start;
__private_extern__ kmod_stop_func_t *_antimain = sampleKext_stop;
__private_extern__ int _kext_apple_cc = __APPLE_CC__ ;
```

Up until Snow Leopard, `osfmk/kern/kmod.c` used to contain a fair amount of `kmod` handling code, including calls such as `kmod_create`, `kmod_destroy`, and others. At present, however, all these calls return a `KERN_NOT_SUPPORTED` value, with the exception of `kmod_get_info()`, which is a Mach host trap, defined in user mode's `<mach/mach_host.h>`. This still works for 32-bit clients, as shown in Listing 18-5:

LISTING 18-5: kmod_get_info() falling through to kext_get_kmod_info for 32-bit clients

```
kern_return_t
kmod_get_info(
    host_t host __unused,
    kmod_info_array_t * kmod_list KMOD_MIG_UNUSED,
    mach_msg_type_number_t * kmodCount KMOD_MIG_UNUSED)
{
#if __ppc__ || __i386__
 if (current_task() != kernel_task && task_has_64BitAddr(current_task())) {
     NOT_SUPPORTED_USER64();
     return KERN_NOT_SUPPORTED;
 }  return kext_get_kmod_info(kmod_list, kmodCount);
#else
    NOT_SUPPORTED_KERNEL();
    return KERN_NOT_SUPPORTED;
#endif /* __ppc__ || __i386__ */
}

// kext_get_kmod_info is defined in libkern/OSKextLib.cpp:
/*********************************************************************
* Compatibility implementation for kmod_get_info() host_priv routine.
* Only supported on old 32-bit architectures.
*********************************************************************/
#if __i386__
kern_return_t
kext_get_kmod_info(
    kmod_info_array_t      * kmod_list,
    mach_msg_type_number_t * kmodCount)
{
    return OSKext::getKmodInfo(kmod_list, kmodCount);
}
#endif /* __i386__ */
```

Indeed, on a 32-bit system, a quick and dirty implementation of `kextstat(8)` can be coded as shown in Listing 18-6:

LISTING 18-6: kextstat(8)-style output of struct kmod_info_t's. Compile with –arch i386.

```
#include <mach/mach.h>
#include <mach/mach_host.h>

// Quick kextstat(8) like utility - using the 32-bit APIs of kmod_get_info();
// Compile with -arch i386

void main()
{

   mach_port_t            mach_host;
   kern_return_t          rc;
   mach_msg_type_number_t modulesCount = 0;
   kmod_args_t            modules;
   int                    i;
   kmod_info_t           *mod;

   mach_host = mach_host_self();
   rc = kmod_get_info (mach_host,
                      &modules,
                      &modulesCount);

   if (rc != KERN_SUCCESS)
    {
      mach_error ("kmod_get_info",rc);
      exit(2);
    }


  printf("Got %d bytes - %d modules\n", modulesCount, modulesCount/sizeof(kmod_info_t));

  mod = (kmod_info_t *) modules;
  for (i = 0; i < modulesCount / sizeof(kmod_info_t); i++)
    {
       printf("%d\t", mod->id);
       printf("%s\t", mod->name);
       printf("%x\t", mod->address);
       printf("%x\n", mod->size);

       // break after kpi.bsd, which is also #1
       if (mod->id ==1) break;
       mod++; // increments by sizeof(kmod_info_t)
    }

}
```

The `kmod` architecture, however, is considered deprecated, and the code in the previous listing will fail (claiming "service not supported") on 64-bit OS X, or iOS (which is why the Cydia-supplied `kextstat` fails). The APIs exposed by `libKern` must be used in these cases, and they are discussed next.

libKern

While kmod_info_t still serves as the basic structure for kexts, most of the kext handling logic has been moved to the libkern directory and has been rewritten in C++. The logic for maintaining kexts is now in libkern/c++/OSKext.cpp and is exposed to user mode via the I/O Kit framework.

In OS X, Most of the interfacing with kexts is done by a dedicated daemon, kextd(8). This daemon, (which resides in /usr/libexec, with its ilk), serves as a bridge between user mode and the kernel, assisting both in loading kexts and resolving dependencies. It registers host special port #15 (HOST_KEXTD_PORT) when started from Launchd(1), and communicates with user mode clients over Mach messages (MIG subsystem 70000). The IOKit framework exposes KextManager APIs that work with kextd (and hide the the Mach messages to it), as well as non-manager ones that interface with the kernel directly (intended for use by kextd itself). The latter APIs are defined in the the kext.subproj of the open source IOKitUser package, and are listed in Table 18-8.

TABLE 18-8: libKern's OS Kext APIs

API FUNCTION	USER FOR
OSKextLoad(OSKextRef aKext); **OSKextLoadWithOptions** (OSKextRef *aKext*, OSKextExcludeLevel *startExc*, OSKextExcludeLevel *addPExc*, CFArrayRef *personalityNames*, Boolean *delayAutounloadFlag*);	Loading a kext into the kernel. This function is not meant to be used outside kextd(8).
OSKextUnload(OSKextRef *aKext*, Boolean *termSvcAndRmvPrsnlt*);	The core functionality of kextunload(8).
OSKextStart(OSKextRef *aKext*); **OSKextStop**(OSKextRef *aKext*);	Start or stop a kext by calling its start or stop routines, respectively.
Boolean OSKextIsStarted (OSKextRef aKext);	Return true if a kext has been started.
CFDictionaryRef OSKextCopyLoadedKextInfo(CFArrayRef *kextIdentifiers*, CFArrayRef *infoKeys*)	Returns a dictionary of all loaded kexts. The core functionality of kextstat(8). New in Lion and iOS 4.3. Deprecates Snow Leopard/ iOS 3.*x*'s OSKextCreateLoadedKextInfo.

The kextd is (for obvious reasons) not present in iOS. The APIs for direct kext loading and listing, however, still are (but don't be surprised if they disappear soon after this book sees print). A kextstat(8)-like utility, similar to the one in Listing 18-7, would look like the following:

LISTING 18-7: Using the IOKit-exposed OSKext APIs to provide kextstat(8)-like functionality

```
 /* A simple implementation of kextstat(8) which actually works on iOS, as well:
  * All the work is done by OSKextCopyLoadedKextInfo.
  *
  * Compile with -framework IOKit -framework CoreFoundation
  */

#include <CoreFoundation/CoreFoundation.h>

void printKexts(CFDictionaryRef dict)

   // Simple dump of an XML dictionary
   CFDataRef xml = CFPropertyListCreateXMLData(kCFAllocatorDefault,
                                              (CFPropertyListRef)dict);
   write(1, CFDataGetBytePtr(xml), CFDataGetLength(xml));
   CFRelease(xml);
}

int main (int argc, char **argv)
{

  // OSKextCopyLoadedKextInfo does exactly that, i.e. obtains loaded kext
  // information from kernel, and return it as a CoreFoundation "dictionary" object.
  CFDictionaryRef kextDict =
       OSKextCopyLoadedKextInfo(NULL, // CFArrayRef kextIdentifiers,
       NULL);                         //CFArrayRef infoKeys)


   printKexts(kextDict);

}
```

The code in Listing 18-6 merely dumps the dictionary returned by `OSKextCopyLoadedKextInfo()` as an XML plist. The book's companion website contains a more complete version, called `jkextstat`, offering `kextstat(8)` compatible output, as shown in Output 18-6:

OUTPUT 18-6: jkextstat on iOS 5, from the author's iPod Touch 4G

```
root@Podicum (~)# jkextstat
0 __kernel__
1 kpi.bsd
2 kpi.dsep
3 kpi.iokit
4 kpi.libkern
5 kpi.mach
6 kpi.private
7 kpi.unsupported
8 driver.AppleARMPlatform <1 3 4 5 6 7>
9 iokit.IOStorageFamily <1 3 4 5 6 7>
```

continues

OUTPUT 18-6 *(continued)*

```
10 driver.DiskImages <1 3 4 5 6 7 9>
11 driver.FairPlayIOKit <1 3 4 5 6 7>
12 driver.IOSlaveProcessor <3 4>
13 driver.IOP_s5l8930x_firmware <3 4 12>
14 iokit.AppleProfileFamily <1 3 4 5 6 7>
15 iokit.IOCryptoAcceleratorFamily <1 3 4 5 7>
16 driver.AppleMobileFileIntegrity <1 2 3 4 5 6 7 15>
17 iokit.IONetworkingFamily <1 3 4 5 6 7>
18 iokit.IOUserEthernet <1 3 4 5 6 16 17>
19 platform.AppleKernelStorage <3 4 7>
20 iokit.IOSurface <1 3 4 5 6 7 8>
21 iokit.IOStreamFamily <3 4 5>
22 iokit.IOAudio2Family <1 3 4 5 21>
23 driver.AppleAC3Passthrough <1 3 4 5 7 8 11 21 22>
24 iokit.EncryptedBlockStorage <1 3 4 5 9 15>
25 iokit.IOFlashStorage <1 3 4 5 7 9 24>
26 driver.AppleEffaceableStorage <1 3 4 5 7 8 25>
27 driver.AppleKeyStore <1 3 4 5 6 7 15 16 26>
28 kext.AppleMatch <1 4>
29 security.sandbox <1 2 3 4 5 6 7 16 28>
30 driver.AppleS5L8930X <1 3 4 5 7 8>
31 iokit.IOHIDFamily <1 3 4 5 6 7 16>
32 driver.AppleM68Buttons <1 3 4 5 7 8 31>
33 iokit.IOUSBDeviceFamily <1 3 4 5>
34 iokit.IOSerialFamily <1 3 4 5 6 7>
35 driver.AppleOnboardSerial <1 3 4 5 7 34>
36 iokit.IOAccessoryManager <3 4 5 7 8 33 34 35>
37 driver.AppleProfileTimestampAction <1 3 4 5 14>
38 driver.AppleProfileThreadInfoAction <1 3 4 6 14>
39 driver.AppleProfileKEventAction <1 3 4 14>
40 driver.AppleProfileRegisterStateAction <1 3 4 14>
41 driver.AppleProfileCallstackAction <1 3 4 5 6 14>
42 driver.AppleProfileReadCounterAction <3 4 6 14>
43 driver.AppleARMPL192VIC <3 4 5 7 8>
44 driver.AppleCDMA <1 3 4 5 7 8 15>
45 driver.IODARTFamily <3 4 5>
46 driver.AppleS5L8930XDART <1 3 4 5 7 8 45>
47 iokit.IOSDIOFamily <1 3 4 5 7>
48 driver.AppleIOPSDIO <1 3 4 5 7 8 12 47>
49 driver.AppleIOPFMI <1 3 4 5 7 8 12 25>
50 driver.AppleSamsungSPI <1 3 4 5 7 8>
51 driver.AppleSamsungSerial <1 3 4 5 7 8 34 35>
52 driver.AppleSamsungPKE <3 4 5 7 8 15>
53 driver.AppleS5L8920X <1 3 4 5 7 8>
54 driver.AppleSamsungI2S <1 3 4 5 7 8>
55 driver.AppleD1815PMU <1 3 4 5 7 8 31>
56 iokit.AppleARMIISAudio <1 3 4 5 7 22>
57 driver.AppleEmbeddedAudio <1 3 4 5 7 8 22 31 56>
58 driver.AppleCS42L59Audio <3 4 5 8 22 31 56 57>
59 driver.AppleEmbeddedAccelerometer <3 4 5 7 8 31>
```

```
60 driver.AppleEmbeddedGyro <1 3 4 5 7 8 31>
61 driver.AppleEmbeddedLightSensor <3 4 5 7 8 31>
62 driver.AppleEmbeddedUSB <1 3 4 5 7 8>
63 driver.AppleS5L8930XUSBPhy <1 3 4 5 7 8 62>
64 iokit.IOUSBFamily <1 3 4 5 7>
65 driver.AppleUSBEHCI <1 3 4 5 7 64>
66 driver.AppleUSBComposite <1 3 4 64>
67 driver.AppleEmbeddedUSBHost <1 3 4 5 7 62 64 66>
68 driver.AppleUSBOHCI <1 3 4 5 64>
69 driver.AppleUSBOHCIARM <3 4 5 8 62 64 67 68>
70 driver.AppleUSBHub <1 3 4 5 64>
71 driver.AppleUSBEHCIARM <3 4 5 8 62 64 65 67 70>
72 driver.AppleS5L8930XUSB <1 3 4 5 7 8 62 64 65 67 68 69 71>
73 driver.AppleARM7M <3 4 8 12>
74 driver.EmbeddedIOP <3 4 5 12>
75 driver.AppleVXD375 <1 3 4 5 7 8 11>
76 iokit.IOMobileGraphicsFamily <1 3 4 5 7 8>
77 iokit.IODisplayPortFamily <1 3 4 5 6 7 22>
78 driver.AppleDisplayPipe <1 3 4 5 7 8 76>
79 driver.AppleRGBOUT <1 3 4 5 7 8 76 77 78>
80 driver.AppleTVOut <1 3 4 5 7 8>
81 driver.AppleAMC_r2 <1 3 4 5 7 8 11 21 22>
82 driver.AppleSamsungDPTX <3 4 5 7 8 77>
83 iokit.IOAcceleratorFamily <1 3 4 5 7 8>
84 IMGSGX535 <1 3 4 5 7 8 83>
85 driver.H2H264VideoEncoderDriver <1 3 4 5 7 8>
86 driver.AppleJPEGDriver <1 3 4 5 7 8>
87 driver.AppleH3CameraInterface <1 3 4 5 7 8>
88 driver.AppleM2ScalerCSCDriver <1 3 4 5 7 8 45>
89 driver.AppleCLCD <1 3 4 5 7 8 76 78>
90 driver.AppleSamsungMIPIDSI <1 3 4 5 7 8>
91 driver.ApplePinotLCD <1 3 4 5 7 8>
92 driver.AppleSamsungSWI <1 3 4 5 7 8>
93 driver.AppleSynopsysOTGDevice <1 3 4 5 7 8 33 62>
94 driver.AppleNANDFTL <1 3 4 5 7 9 25>
95 driver.AppleNANDLegacyFTL <1 3 4 5 9 25 94>
96 AppleFSCompression.AppleFSCompressionTypeZlib <1 2 3 4 6>
97 IOTextEncryptionFamily <1 3 4 5 7 11>
98 driver.AppleBSDKextStarter <3 4>
99 nke.ppp <1 3 4 5 6 7>
100 nke.l2tp <1 3 4 5 6 7 99>
102 iokit.IO80211Family <1 3 4 5 6 7 17>
103 driver.AppleBCMWLANCore <1 3 4 5 6 7 8 17 102>
104 driver.AppleBCMWLANBusInterfaceSDIO <1 3 4 5 6 7 8 47 103>
105 driver.AppleDiagnosticDataAccessReadOnly <1 3 4 5 7 8 94>
106 driver.LightweightVolumeManager <1 3 4 5 9 15 24 26>
107 driver.IOFlashNVRAM <1 3 4 5 6 7 25>
108 driver.AppleNANDFirmware <1 3 4 5 25>
109 driver.AppleImage3NORAccess <1 3 4 5 7 8 15 108>
110 driver.AppleBluetooth <1 3 4 5 7 8>
111 driver.AppleMultitouchSPI <1 3 4 5 7 8>
112 driver.AppleUSBMike <1 3 4 5 8 22 33>
113 driver.AppleUSBDeviceMux <1 3 4 5 6 7 33>
```

The free tool provides many additional features improving on the original, such as XML and experimental graph output (similar to Figure 18-1), as well as recursively following kext dependencies — for both OS X and iOS.

Behind the Scenes of Kext Loading

The APIs we have seen so far are all user mode APIs. This is no surprise, as the initiative for loading a kext comes from user mode — whether from a system process, such as `launchd(8)`, in reaction to a detected hardware change, or from the administrator, by manually using one the kext utilities. The actual loading of the kext, however, involves kernel memory operations, and can only be performed in kernel mode.

To bridge the divide, kext loading relies on Mach messages. All kext operations are encapsulated as serialized XML in the `ool_descriptors` of Mach `kext_request` messages (message #425). These messages, which are part of the `host_priv` subsystem (discussed in Chapter 9), naturally require access to the host's privileged port. Recall, that Mach messages eventually involve the `mach_msg_trap`, which moves from user mode to kernel mode.

Using the companion website's Mach message snoop tool will reveal the serialized XML, for example as in Output 18-7, associated with a kext unload:

OUTPUT 18-7: Serialized unload kext_request message:

```
OSKextUnloadKextWithIdentifier("kextName", //CFStringRef kextIdentifier,
                               true);    // Boolean
                                terminateServiceAndRemovePersonalities);


<dict>
  <key>Kext Request Predicate</key><string>Unload</string>
  <key>Kext Request Arguments</key>
  <dict>
    <key>TerminateIOServices</key><true/>
    <key>CFBundleIdentifier</key><string>kextName</string>
  </dict>
</dict>
```

Likewise, snooping OS X's `kextstat(8)` yields the following:

```
<dict>
   <key>Kext Request Predicate</key>
       <string>Get Loaded Kext Info</string>
   <key>Kext Request Arguments</key>
       <dict><key>CFBundleIdentifier</key><array></array></dict>
</dict>
```

The header file `libkern/libkern/kext_request_keys.h` provides a listing of all the various request "keys" or predicates, which are all textual. They are listed in Table 18-9:

TABLE 18-9: Predicates for kext_request

PREDICATE	PRIVILEGED	USE
Get Loaded Kext Info	No	Get currently loaded kext information
Get Kernel Image	No	Get sanitized kernel image
Get Kernel Load Address	No	Get load address of kernel (for debugging)
Get All Load Requests	No	Get status of all kext load requests since boot
Get Kernel Requests	Yes	Retrieve list of all kext load requests, including those from kernel space
Load	Yes	Load one or more kexts
Start	Yes	Start a kext
Stop	Yes	Stop a kext
Unload	Yes	Unload (remove) a kext

The privileged predicate are reserved for `kextd` use, though up to an including Lion they can be used by any root process. The kernel may occasionally initiate requests back to user mode (i.e. `kextd`), as well. These requests include Send Resource, to ask `kextd` to retrieve a file resource belonging to a kext, and Kext Load Request, which asks `kextd` to load a kext from disk, and send it to the kernel. Additionally, `kextd` can get notifications from the kernel for kext loading and unloading.

Experiment: Viewing kext_request Messages Issues by kextd

Using gdb, you can view both `mach_msg()`s sent to and from `kextd` on an OS X system. To start, find the PID of `kextd`, and attach to it using gdb `-p`, as shown in Output 18-8:

OUTPUT 18-8: Attaching to kextd with gdb

```
root@Simulacrum (/)# ps -ef | grep kextd
    0  11       1   0  5:46PM ??         0:00.12 /usr/libexec/kextd
    0  4217  4214   0  5:48PM ttys007    0:00.01 grep kextd
root@Simulacrum (/)# gdb -p 11
GNU gdb 6.3.50-20050815 (Apple version gdb-1817) (Thu Apr  5 20:54:43 UTC 2012)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "x86_64-apple-darwin".
/Users/mahmood1/4197: No such file or directory
```

continues

OUTPUT 18-8 *(continued)*

```
Attaching to process 11.
Reading symbols for shared libraries . done
Reading symbols for shared libraries
.................................................................. done
Reading symbols for shared libraries + done
0x00007fff8642e6ae in mach_msg_trap ()
```

The `kextd(8)` will be in broken into in `mach_msg_trap()` — not surprising, as this is the blocking system call in the heart of its message loop. Add a breakpoint on `kext_request`, and continue:

```
(gdb) break kext_request
Breakpoint 1 at 0x7fff86421770
(gdb) c
Continuing.
```

In another terminal (and, if you can, another window), run `kextload(8)`, and load some harmless module, such as the NTFS driver (`kextload /System/Library/Extensions/ntfs.kext`). You should see `kextd(8)` break on `kext_request`, as it receives a message on its host special port, and relays it as a `kext_request` to the kernel. Likewise, `kextload(8)` will hang, since it is waiting on `kextd`'s reply. Printing the value of the RDX register as a string will reveal the message, as shown in Output 18-9:

OUTPUT 18-9: Displaying kext MIG messages

```
 (gdb) x/6s $rdx        # First request is a Get Loaded Kext Info, on the NTFS.kext
0x7f8c8a00d200:  "<dict><key>Kext Request Predicate</key>
                  <string>Get Loaded Kext Info</string>
                  <key>Kext Request Arguments</key><dict>
                  <key>Kext Request Info Keys</key>
                  <array><string>CFBundleIdentifier</string><string>CF"...
0x7f8c8a00d2c8:  "BundleVersion</string><string>OSBundleCompatibleVersion
</string><string>OSBundleIsInterface</string><string>OSKernelResource</string>
<string>OSBundleCPUType</string><string>OSBundleCPUSubtype</string>"...
0x7f8c8a00d390:  "<string>OSBundlePath</string><string>OSBundleUUID</string>
<string>OSBundleStarted</string><string>OSBundleLoadTag</string>
<string>OSBundleLoadAddress</string><string>OSBundleLoadSize</string> "...
0x7f8c8a00d458:  "SBundleWiredSize</string><string>OSBundlePrelinked</string>
<string>OSBundleDependencies</string><string>OSBundleRetainCount</string>
</array><key>CFBundleIdentifier</key><array><string>com.apple.kpi.li"...
0x7f8c8a00d520:  "bkern</string><string>com.apple.kpi.private</string>
<string>com.apple.kpi.unsupported</string><string>com.apple.kpi.mach</string>
<string>com.apple.kpi.bsd</string><string>com.apple.filesystems.ntfs</s"...
0x7f8c8a00d5e8:  "tring></array></dict></dict>"
(gdb) c
Continuing.

Breakpoint 1, 0x00007fff86421770 in kext_request ()
(gdb) x/6s $rdx                  # Actual load request is in MultiKext form
0x10b1eb000:     "MKXTMOSX"
```

As further exercise, try and break inside `kext_request`, to intercept the kernel's reply. You could try to break on the incoming `mach_msg` from `kextload` (or, alternatively, run `kextload` under gdb as well).

SUMMARY

This chapter discussed Kernel Extensions — KEXTs, and kernelcaches. Both are important concepts in the OS X and iOS kernel space, as they provide the flexibility required by the kernel to support third party devices and enhancements. In the right hands, KEXTs offer the developer the ability to add functionality to the kernel, and provide device drivers, primarily using I/O Kit, as is shown in the next chapter. In the wrong hands, the functionality of a KEXT — injecting code directly into kernel space — can be abused to no end, providing a fulcrum for rootkits and malware to quite literally move the kernel.

REFERENCES

1. Apple Developer, "Kernel Programming Guide," `http://develeoper.apple.com/`
2. Apple Developer, "Kernel Extensions Programming Topics," `http://developer.apple.com`

19

Driving Force — I/O Kit

Unlike other operating systems, XNU is unique in its offering of a complete runtime environment for device drivers. Even more unique is that this environment enables developers to code in C++ rather than C, which has traditionally been, alongside assembly, the language of choice for kernel programming.

XNU's device driver environment is called the I/O Kit, and it is a proprietary component developed by Apple. It is neither part of Mach, nor BSD (nor, for that matter, the legacy OS 9). Its roots are in NeXTSTEP's DriverKit though it has advanced considerably since then. It is a largely self-contained environment, meaning that developers can code and rely solely on the I/O Kit APIs, remaining largely ignorant of the Mach or BSD layers. By enabling C++, I/O Kit brings to developers the power of object orientation, chiefly subclassing and function overriding, which transforms the device driver development process into a much more efficient one. Driver developers need not implement everything from scratch, but can actually subclass existing drivers, inheriting some already-implemented features to save time, while overriding and providing different implementations for others.

I/O Kit also offers its own user mode set of APIs, the I/O Kit Framework, which provides advanced features such as kernel notifications and kernel-to-user (and vice versa) communications.

This chapter covers I/O Kit, dealing with its low-level implementation, which is part of the XNU open source. I/O Kit is already well documented by Apple Developer references[1,2], and the reader is encouraged to read these for the driver API specifics. Rather than discuss drivers of various types as other books do[3], we focus on the framework itself, and the implementation of the features widely required by all drivers: memory allocation, interrupt handling, and others.

> *This chapter applies to iOS as it does to OS X, since I/O Kit is part of iOS, and is in fact widely used by Apple for all the device drivers. Due to the restrictionson iOS, however, developing third-party drivers for Apple's i-Devices is extremely hard (not to say impossible). This makes the term "iOS Kernel Programming" virtually non-existent outside Apple's own circles. Even on a jailbroken device, kext and I/O Kit support is (intentionally) limited. Also remember there are very few public kernel symbols to link the drivers with. Apple doesn't want anyone messing around with its prized embedded OS, even more so when it involves the kernel.*

INTRODUCING I/O KIT

I/O Kit is quite unique in its design. While all other operating systems certainly have device drivers, most are doomed to be written in C, and don't have their own runtime environment. Few exceptions exist, notably Windows' NDIS and the new Windows Driver Foundation architecture, but none is as extensive and as object oriented as I/O Kit.

Device Driver Programming Constraints

Device drivers are the primary reason why developers opt to abandon the relative safety of user mode and delve into the hazardous realms of kernel programming. Under normal conditions, user mode code is simply unable to directly access hardware, due to ring (or on ARM, CPSR) restrictions. Although user mode driver frameworks exist, most notably for USB, they are fairly limited, and often don't live up to the requirements of high-throughput devices, such as disks or display adapters.

Device drivers, however, operate under the tightest set of requirements possible. By virtue of living in the kernel, they inherit all the restrictions of kernel mode: limited wired memory, no user mode APIs, and a very narrow margin of error, with nearly every bug potentially resulting in a kernel panic. Due to the drivers' interfacing with hardware, however, the margin of error becomes even narrower still. Device drivers often have to deal with interrupts from their devices, which are the most critical parts of kernel code, and introduce even further complications dealing with concurrency and code reentrance. To further complicate things, every operating system has its own device driver model, resulting in a very steep learning curve, which often proves to be a slippery one, as well.

As such, it is somewhat a relief for developers, in that sense, to be presented with I/O Kit as the API environment of choice for OS X. Object orientation makes plenty of sense when one considers that devices can be thought of as instances of their respective classes. While I/O Kit requires a certain paradigm shift from the usual view of device driver programming, its features make the shift and adaptation well worth it. These features are discussed next, but before plunging into the details, we first need to lay out a few clear foundations.

What I/O Kit Is

Before we introduce the internals of I/O Kit, it makes sense to clearly define what I/O Kit is and is not.

A (Nearly) Self-Contained Environment

I/O Kit is a nearly self-contained runtime environment for drivers. The closest non–OS X comparable runtime is NDIS (Network Driver Interface Specification), which is widely used on Windows to provide a model and an environment for network device drivers. The NDIS APIs wrap those of Windows, and a fully NDIS-compliant driver can also run on Linux's NDISWrapper.

I/O Kit has not been implemented anywhere but OS X and iOS (though, in theory, it can be). It is, however, a full environment, and an I/O Kit driver can theoretically rely solely on the I/O Kit APIs, which wrap those of the underlying Mach[1]. Indeed, the I/O Kit APIs for creating threads, allocating memory, and many other common tasks are merely thin wrappers over the Mach APIs. Listing 19-1 shows an example of this in `IOCreateThread`, which wraps Mach's `kernel_thread_start`:

LISTING 19-1: I/O Kit thread creation and exit APIs, from I/O Kit/Kernel/IOLib.cpp

```
IOThread IOCreateThread(IOThreadFunc fcn, void *arg)
{
     kern_return_t   result;
     thread_t                thread;

     result = kernel_thread_start((thread_continue_t)fcn, arg, &thread);
     if (result != KERN_SUCCESS)
               return (NULL);

     thread_deallocate(thread);

     return (thread);
}


void IOExitThread(void)
{
    (void) thread_terminate(current_thread());
}
```

In terms of performance, the overhead from I/O Kit is fairly small (in many cases, direct fall-through calls such as `IOExitThread()` can be optimized by the compiler). Using the I/O Kit APIs hides the underlying Mach APIs, making drivers potentially forward compatible even if Mach is someday changed or altogether removed.

An Object-Oriented Environment

I/O Kit drivers are objects instantiated and derived from certain base classes. These base classes are, for the most part, provided by Apple. The topmost class — the abstract `OSObject` — is akin to C++'s or Java's basic idea of an "object." Though `OSObject` cannot be instantiated (because it is abstract), everything is a type of `OSObject`. The true power, however, comes from its descendants, which form a complex class hierarchy spanning well over a hundred classes. A developer can find the class that is closest to his or her own required driver and pick up from there, effectively reusing code that is generic enough to be in the class itself.

[1] Theoretically, as more often than not drivers, even Apple's own, stray outside the I/OKit APIs.

For example, consider an Ethernet driver. Your own specific driver for a proprietary multi-gigabit Ethernet would still share common logic with the lowliest of the 10 Mbps cards. Namely, Ethernet frame encapsulation, MAC address handling, and many other features are invariant, being part of the low-level Ethernet protocol. Implementing these in a driver from scratch would consume valuable time, and worse, might introduce bugs. Reusing tested code shortens the development time considerably and lends itself to more solid, robust code, which is especially important for drivers.

Specifically Designed for Drivers

I/O Kit provides support for many aspects of programming that are specific to working with devices — primarily plug 'n' play, and power management. Another important architectural idea is that of driver layering, which enables the stacking of device drivers on top of one another.

Work Loop Driven

I/O Kit offers a work loop model, which is somewhat similar to Objective-C's Run loop (or Mach's message loop). In a nutshell, a work loop is a message handling loop which continuously processes events. Using a work loop greatly simplifies concurrency issues, and can often alleviate the need for locks, which may impact performance.

Registry Based

Unlike other driver environments, in I/O Kit everything is accounted for — objects referenced, classes registered, and more — and is managed in the I/O Registry, which is a multi-layered hierarchical database tracking both the objects and their interrelations. This registry is maintained in kernel memory, and can be queried from within an I/O Kit driver or from user mode using the `ioreg(8)` command, which will be discussed later in this chapter.

User (Mode) Friendly

I/O Kit offers APIs for user mode access, and in fact you can implement some drivers, such as those of USB devices, entirely in user mode. The I/O Kit registry is also readily accessible from user mode (as will be shown later in this chapter), allowing the user mode program to query hardware configuration and parameters.

Implemented in a subset of C++

Because I/O Kit is C++ based, it draws on some of the language's useful compile time features, such as:

- **Namespaces:** I/O Kit drivers can use C++ namespaces to wrap their functions and symbols, which helps avoid global symbol conflicts in the kernel.
- **Name mangling:** I/O Kit symbols are mangled, which embedding of the C++ level prototype information (namespace, return value and arguments) in the function name. This feature actually comes in very handy when inspecting the iOS kernel symbols: A name demangler (for example, HexRays' IDA-Pro or the free `http://demangler.com`) can quickly recover the prototype from the otherwise weird-looking symbol.

What I/O Kit Isn't

For all its capabilities, I/O Kit is still not a perfect environment. It has some shortcomings. Specifically:

A Full C++ Environment

I/O Kit is implemented in C++, but the C++ is a restricted subset of the C++ you probably know and love (or hate) from user-land. In particular, it does not offer the following features:

- **Templates:** These compile-time features of C++ are not present in I/O Kit, so using the familiar template `< >` on data structures is impossible. There is no STL support.
- **Exceptions:** One of C++'s most powerful features is structured exception handling. I/O Kit will have none of that, so the `try`/`catch` blocks must be left behind. The kernel stack is limited, because the kernel generally does not place exception handlers on kernel mode code.
- **Standard constructors:** These can't be used in I/O Kit because the only way to fail in a constructor is to throw an exception, and I/O Kit does not support exceptions. Instead, object construction is split into two — a new operator (essentially a simple wrapper over `malloc`) and an `init()` function, which prepares the object.

A Full-Featured API

The I/O Kit APIs are good, but not that good. Because there is no full C++ runtime, the only runtime functionality is provided by a custom library called libkern. In order to be fully compliant with I/O Kit, a developer is expected to use only the libkern APIs. A developer might find using those limited, as it requires getting used to the I/O Kit primitives (e.g. `OSArray`, `OSDictionary`), rather than the familiar data types of C++.

Another problem that arises is the minor transgression into Mach or BSD space. As stated before, the aim of I/O Kit is to be fully self-contained, but it somewhat falls short of that. Even Apple's own examples sometimes use data types or functions that are in Mach headers. This requires the developer to be cognizant of some Mach primitives after all, and may hinder portability if I/O Kit is ever ported out of Apple's systems.

The Most Flexible of Programming Models

An I/O Kit driver must implement a very specific lifecycle, which marks a significant departure from normal driver callbacks that are well known from other operating systems. The lifecycle is quite complex, and a developer needs to know what callback to implement under what specific conditions.

All about code

I/O Kit drivers aren't just binaries. Being kexts, they must contain the mandatory `Info.plist`. Being I/O Kit drivers, the `Info.plist` is expected to contain I/O Kit-specific directives, without which the driver cannot function. It is not uncommon for a developer to spend frustrating hours debugging a driver that failed to load before realizing the problem is a typo in the driver's property list.

LIBKERN: THE I/O KIT BASE CLASSES

I/O Kit's foundation, the libkern C++ runtime, defines the primitive classes that are available for use in all I/O Kit drivers. These primitives, which correlate somewhat with those of CoreFoundation, are defined in XNU's `libkern/libkern/c++` directory (in `.h` files) and implemented in the `libkern/c++` directory, in simple files, one per class. This is shown in Table 19-1:

TABLE 19-1: I/O Kit Primitives Provided by libkern

LIBKERN/ I/O KIT CLASS	CORRESPONDING COCOA/CARBON CLASS	USED FOR
`OSObject`	`NSObject`	The parent class of all there is. Everything in I/O Kit inherits from this (with the exception of `OSMetaClass`), and by doing so automatically obtains reference counting logic and other top-level methods.
`OSMetaClass`	N/A	An abstract class used extensively in I/O Kit to provide RTTI services, in place of C++ RTTI, which is unsupported.
`OSArray`	`CFArray`	An array of `OSObject`s.
`OSBoolean`	`CFBoolean`	A primitive boolean type. Simple wrapper over a private `bool` value.
`OSCollection` `OSCollectionIterator`	N/A	An abstract collection object and its iterator. The latter inherits from `OSIterator`.
`OSData`	`CFData`	An opaque array of bytes.
`OSDictionary`	`CFDictionary`	An associative array. This is functionally the same as a Perl or Java hash, or Objective-C's `CFDictionary` object.
`OSIterator`	N/A	Abstract base class for iterators.
`OSKext`	N/A	A class defining a kernel extension.
`OSNumber`	`CFNumber`	A number — integer, float, or double.
`OSOrderedSet` `OSSet`	`CFSet`	An ordered and an unordered set, respectively. Both inherit from `OSCollection`.
`OSString`	`CFString`	A C-String wrapper.
`OSSymbol`	N/A	Unique, reusable symbols (for example, hard-coded strings).

The `libkern/c++` directory also contains support files (`OSRuntime.c` and `OSRuntimeSupport.cpp`) that are used during libkern's initialization as well as serialization functions (`OSSerialize`/`OSUnserialize`) to allow the writing and reading of objects from XML property lists.

OSObject

All classes but one in I/O Kit's extensive hierarchy trace back to one ancestor, called `OSObject`. This is the same "object" ancestor that can be found in Java and C++ and is akin to the `NSObject` of Cocoa. Inheriting from `OSObject` involves a slight change in the programming model. Due to the lack of exception support, constructors may no longer be used to initialize the newly created objects. Instead, object instantiation is now split into two phases: the allocation of memory for it (which is done, as always, using the `new` operator), and the initialization, which is carried out by a separate `init()` function. It is the responsibility of a client creating an object to follow the `new` operator by a call to `init()`, and to check the return value of the latter. If `init()` returns false, the object cannot be used, and must be freed.

Quite a few I/O Kit classes implemented static factory methods, which perform the work of new and init in the same function. These follow a loose convention of "with," allowing for multiple factory methods which take different arguments.

Another slight change in the model is the alleviation of the need to explicitly call `free` or `delete` to dispose of an object. In fact, these are disallowed. Instead, `OSObjects` maintain reference counts, which can be incremented (with `retain`) or decremented (with `release`). Code is expected to use only those two methods, with `release` automatically freeing and deleting the object when the reference count drops to zero. The object's `free()` is still supported as the anti-function of its `init()`, and for user-defined objects should be overridden to counteract any initializations on allocations performed during `init()`.

OSMetaClass

I/O Kit doesn't support the standard C++ RunTime Type Identification (RTTI). It offers a similarly powerful mechanism, however, in its `OSMetaClass`.

The `OSMetaClass` is an abstract class and is not meant to be used directly. It does, however, require that special macros be used to enable its RTTI features. These macros include the following:

- **`OSDeclareDefaultStructors`**: This is used to emit the prototypes of the default constructors and destructors (hence, "Structors") for I/O Kit objects. Virtually all I/O Kit objects have this in their header file. Abstract classes use `OSDeclareAbstractStructors`, instead. The macros take two arguments — the driver class name and its superclass.
- **`OSDefineMetaClassAndStructors`**: This is similarly used in the class implementation. Abstract classes use `OsDefineMetaClassAndAbstractStructors` — The suffix `WithInit` may be appended to both, for macros that also include the initialization function.

THE I/O REGISTRY

I/O Kit maintains an up-to-date database on all of its objects and the interrelations between them. This database resides in memory and is known as the *I/O Registry*. This should not be confused with Windows' registry, which is arguably somewhat similar, but with far reaching differences.

The I/O Kit registry is multi-planar. Quite simply, this means that it exists in three dimensions (unlike most graphs, which are bi-dimensional) and can be examined in one of several planes. Registered objects are like lines, which cut through the planes, and may exist in some, and be missing from others. As a consequence, their relationships with other objects are dependent on which plane they are viewed in. An object may be connected to its parent on one plane, but not another.

Table 19-2 lists the planes that are currently defined.

TABLE 19-2: Currently Defined Planes

PLANE	USED FOR
`IOService`	The default plane, wherein all objects have some connection to a parent.
`IOACPIPlane`	The ACPI-enabled devices, as exported by `AppleACPIPlatform.kext`. Not applicable on iOS, which does not support ACPI.
`IODeviceTree`	The Device Tree, as constructed by EFI (or iBoot) and exported by the `IOPlatformExpert`.
`IOPower`	Devices that respond to power management events. Devices are connected in this plane if a power failure in one affects another. Drivers can selectively opt-in to this plane if they require power management by calling `PMInit()` and then asking their provider to `joinPMTree()`. (You can find more on that topic in the "I/O Kit Power Management" section.)
`IOUSB`	USB devices. This hierarchy is based on the USB devices' own hierarchy. Usually not found on iOS, but may be created dynamically; for example, when an i-Device is connected to Apple's digital camera kit.
`IOFireWire`	Firewire buses and devices, if any. Like USB, the hierarchy is based on the internal hierarchy of devices connected. Not applicable on iOS or any Macs that do not support FireWire (for example, MacBook Air).

As noted in Table 19-2, planes may also be created dynamically. This is rarely done outside I/O Kit's initialization, but one example is iOS's USB host support, which is enabled when Apple's digital camera kit's adapter is attached to, say, an iPad. Observant hackers have long noticed that the "kit" is nothing more than a adapter that transforms an iPad into a USB 2.0 host (albeit in a limited manner — USB devices cannot draw power, which limits most hard disks, but lightweight devices like keyboards can, in fact, be connected).

The defined planes are maintained under the root entry, in the `"IORegistryPlanes"` property (`kIORegistryPlanesKey` in `I/O Kit/I/O Kit/I/O KitKeys.h`). A quick way to find out what planes are defined on a given system is by using `ioreg(8)` and singling out the `"IORegistryPlanes"` key, as shown in Listing 19-2. As noted in Table 19-2, the iMacs, Minis, and Pros also have an `"IOFireWire"` plane.

LISTING 19-2: Viewing registry planes on a MacBook Air and on an iPad 2.

```
#
# Macbook Air
#
morpheus@Ergo (~)$ ioreg -l -w 0 | grep IORegistryPlanes
  |   "IORegistryPlanes" = {"IOACPIPlane"="IOACPIPlane","IOPower"="IOPower",
"IODeviceTree"="IODeviceTree", "IOService"="IOService","IOUSB"="IOUSB"}
#
#... and, on a jailbroken iPad (with ioreg installed from Cydia)
#
root@Padishah (/) # ioreg -l -w 0 | grep RegistryPlanes
  |   "IORegistryPlanes" = {"IODeviceTree"="IODeviceTree","IOService"="IOService",
"IOPower"="IOPower"}
```

The `ioreg(8)` command is really an all-in-one utility for all things I/O Registry–related. Because it is a command-line utility, it is very useful. As shown in Listing 19-2, it can be used with myriad switches. The `-l` switch is used to list properties (which `"IORegistryPlanes"` is), and `-w 0` disables the truncation of output on terminal window boundary). This command can also be compounded with the powerful `grep(1)` to quickly single-out only the class, instance, or property of interest. GUI-oriented developers might prefer `IORegistryExplorer`, which is part of XCode, and can also show live registry changes such as the addition and removal of devices, as shown in Figure 19-1.

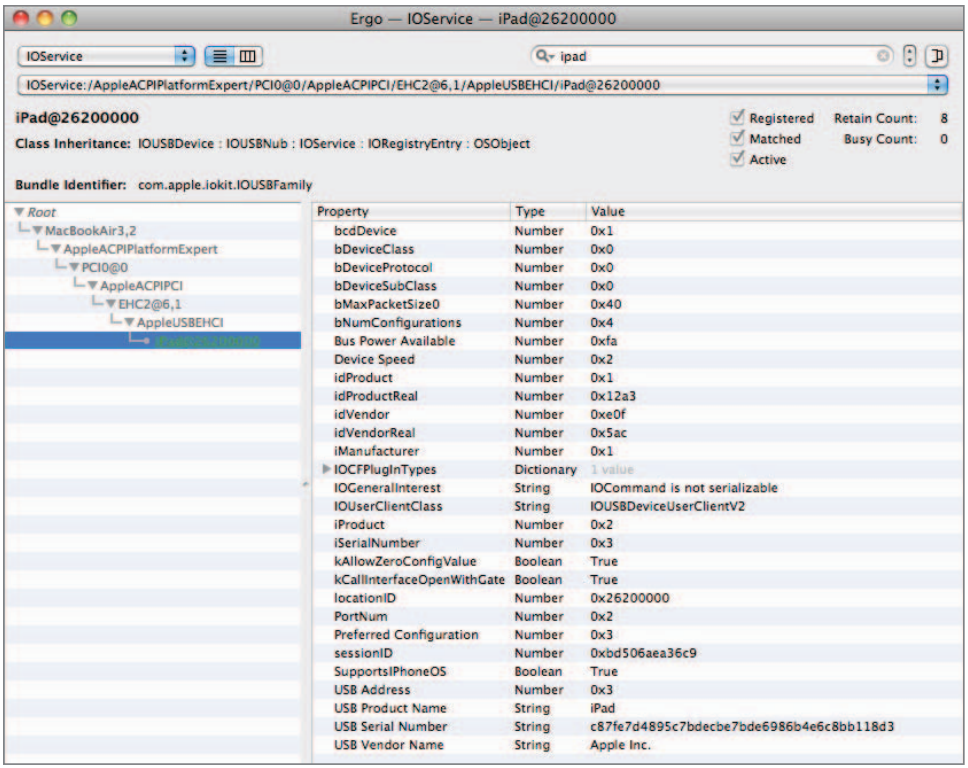


FIGURE 19-1: IORegistryExplorer showing the connection of an iPad to a MacBook Air

In each plane, the objects are organized in a hierarchical tree structure. Each object can be found by a path-like specification, which is reminiscent of the Solaris or Linux Device tree (and, in the case of the `IODeviceTree` plane, follows it). In addition, each object has a unique path designating its class inheritance, tracing back to `OSObject`. Remember that I/O Kit does not allow multiple inheritance; therefore, both the existence and uniqueness of this inheritance path are assured.

IORegistryEntry

The `IORegistryEntry` class is used as a parent class for those objects that have representation in the I/O Registry. It is a simple container of the object's properties, which are stored as an `OSDictionary` object. The class is not meant to be directly inherited from. The parent class for I/O Kit objects is `IOService`, a subclass of this one. By virtue of inheritance, however, all drivers are also automatically registered.

`IORegistryEntry` contains some 70 or so functions that deal with the implementation of the `IORegistry` and its various planes. The initialize method implements a singleton by either initializing or returning the global `gRegistryRoot` (which can also be obtained by a call to `IORegistryEntry::getRegistryRoot()`). The root also holds the various I/O planes (in the `gIORegistryPlanes` dictionary). The `IORegistryPlane` class itself is also defined (in the same `.cpp` and `.h` files), though its only useful method is `serialize()`. New planes can be created at any time by `IORegistryEntry::makePlane()`, though as noted earlier this is fairly rare outside initialization. The `IORegistryEntry` class is responsible for implementing the registry objects' interface: getting and setting properties, managing hierarchy, and associating with an I/O plane. By inheriting from it (via `IOService`), a driver gets all these services "for free."

IOService

The direct (and only) descendant of `IORegistryEntry` is `IOService`. It is also the ancestor of all drivers, both Apple supplied and third party. Though most drivers aren't direct subclasses of `IOService`, they are still its eventual descendants, and inherit from it the set of functions they are capable of using (such as power management, interrupt handling, and so on) and in some cases, expected to implement (such as the driver standard callbacks). This is described in more detail later in the "I/O Kit Kernel Drivers" section.

The common ancestry of all I/O Kit classes comes in handy during various registry walking and enumeration tasks. This is shown next.

I/O KIT FROM USER MODE

I/O Kit drivers can communicate with user mode through APIs offered by the `I/O Kit.Framework`, and its `IOKitLib` APIs. This framework is solely intended for user mode, as kernel mode I/O Kit components are expected to use the `IOKit/` subdirectory of `Kernel.Framework`. User mode applications can use the APIs to interface with I/O Kit drivers in the kernel, as well as the I/O Kit components themselves, most notably the I/O Registry.

All I/O Kit functions rely on a special host port, which I/O Kit refers to (and obtains by a call to) `IOMasterPort()`. This function is really just a simple wrapper over the `host_get_io_master()`

function, which obtains the `IO_MASTER_PORT` special port from `mach_host_self()`. (Special ports are discussed in Chapter 10.) Alternatively, applications can use `kIOMasterPortDefault` as a constant value in place of the master port, which causes I/O Kit to look up the port internally. Communications between user mode and I/O Kit kernel components and drivers is carried over Mach messages, generated as subsystem 2800 by MIG (as can be seen in `System/Library/Frameworks/IOKit.framework/Headers/iokitmig.h`. The implementations of these routines in the kernel are in `iokit/Kernel/IOUserClient.cpp`.

One additional kernel function is `iokit_user_client_trap`, otherwise known as Mach trap #100. This trap (also implemented in `iokit/Kernel/IOUserClient.cpp` and defined in `IOKitUser`'s `IOTrap.s` for i386) can be used through the `IOKit` framework's exported `IOConnectTrap[0-6]` calls. These calls are used to invoke driver registered functions which are external to I/O Kit, with up to 6 arguments. This mechanism is largely unused, aside from rare cases (e.g. `IOPMSetPMPreferences` in iOS), as the better `IOConnectCallMethod` and friends have been introduced in Leopard.

The IOKitLib APIs are well documented[4], and Apple maintains a developer-friendly guide for user mode developers[5]. These APIs are extremely powerful — this section provides an overview of some of them, while leaving others (even powerful ones, such as `IOConnectMapMemory`) to whet the voracious user's appetite.

I/O Registry Access

With the Master Port in hand, an application may send any number of I/O Kit requests. Commonly, these requests involve querying the I/O Registry. Listing 19-3 shows traversing the I/O Kit planes programmatically:

LISTING 19-3: Traversing I/O Kit's service plane in search of a specific device

```
//
// Simple I/O Kit Registry walker
// Compile with -framework IOKit

#include <stdio.h>
#include <mach/mach.h>
#include <CoreFoundation/CoreFoundation.h> // For CFDictionary

// In OS X, you can just #include <IOKit/IOKitLib.h>. Not so on iOS
// in which the following need to be included directly
#define IOKIT   // to unlock device/device_types..
#include <device/device_types.h> // for io_name, io_string

// from IOKit/IOKitLib.h
extern const mach_port_t kIOMasterPortDefault;

// from IOKit/IOTypes.h
typedef io_object_t     io_connect_t;
typedef io_object_t     io_enumerator_t;
typedef io_object_t     io_iterator_t;
typedef io_object_t     io_registry_entry_t;
typedef io_object_t     io_service_t;
```

continues

LISTING 19-3 *(continued)*

```
// Prototypes also necessary on iOS
kern_return_t IOServiceGetMatchingServices(
        mach_port_t     masterPort,
        CFDictionaryRef matching,
        io_iterator_t * existing );

CFMutableDictionaryRef IOServiceMatching(const char *name);

// Main starts here
int main(int argc, char **argv)
{
    io_iterator_t deviceList;
    io_service_t  device;
    io_name_t     deviceName;
    io_string_t   devicePath;
    char         *ioPlaneName = "IOService";
    int           dev = 0;

    kern_return_t kr;

    // Code does not check validity of plane (left as exercise)
    // Try IOUSB, IOPower, IOACPIPlane, IODeviceTree
    if (argv[1]) ioPlaneName = argv[1];

    // Iterate over all services matching user provided class.
    // Note the call to IOServiceMatching, to create the dictionary

    kr = IOServiceGetMatchingServices(kIOMasterPortDefault,
                                     IOServiceMatching("IOService"),
                                     &deviceList);

    // Would be nicer to check for kr != KERN_SUCCESS, but omitted for brevity

    if (kr){ fprintf(stderr,"IOServiceGetMatchingServices: error\n"); exit(1);}
    if (!deviceList) {  fprintf(stderr,"No devices matched\n"); exit(2); }

    while ( IOIteratorIsValid(deviceList) &&
            (device = IOIteratorNext(deviceList))) {

         kr = IORegistryEntryGetName(device, deviceName);
         if (kr)
            {
                fprintf (stderr,"Error getting name for device\n");
                IOObjectRelease(device);
                continue;
         }

         kr = IORegistryEntryGetPath(device, ioPlaneName, devicePath);

         if (kr) {
                // Device does not exist on this plane
                IOObjectRelease(device);
```

```
                        continue;
                        }

                // can listProperties here, increment device count, etc..
                dev++;
                printf("%s\t%s\n",deviceName, devicePath);
        }

        if (device) {
                fprintf (stderr,
                 "Iterator invalidated while getting devices. Did configuration change?\n");
               }
        return kr;
}
```

The first thing to notice in the listing is the abundance of declarations. OS X supplies `<IOKit/IOKitLib.h>` which defines all these, but the iOS SDK does not have this header. Nonetheless, the typedefs and functions are supported, so it's a simple matter of importing the declarations manually, and so this code can compile and link on iOS, as well. The program flow is simple to follow, and the I/O Kit function names are rather self-explanatory, but much occurs behind the scenes.

First, the call to `IOServiceMatching()` creates a matching dictionary for `IOService`. This matching dictionary is a `CFMutableDictionaryRef` (that is, a pointer to a non-constant `CFDictionary` object), constructed automatically to match on service name or subclass name. Specifying `IOService` as the class name means we are interested in a match of all classes (since it is the progenitor of nearly all other classes).

Every subsequent call to I/O Kit from `IOServiceGetMatchingServices()` internally calls a lowercased version (for example, `io_service_get_matching_services`), for which there is a corresponding kernel implementation, as created by the MIG (you can find the MIG `.defs` file in `osfmk/device/device.defs`, and their implementations in `iokit/Kernel/IOUserClient.cpp`). The communication is naturally carried out over Mach messages. Whereas all I/O Kit objects are opaque to user mode, the kernel functions can dereference them, and return specific fields (for example, `io_registry_entry_get_name`, `_get_path`, and so on). Likewise, the I/O Kit opaque iterator object, which is used to walk through the device collection, can be safely dereferenced in kernel mode to return the device handle.

Getting/Setting Driver Properties

Because device drivers in the I/O Kit model are objects, they have properties. These properties are visible in user mode and may be obtained and even modified by a user mode client. This approach makes for a simple, intuitive way to communicate with device drivers, rather than the traditional UNIX `ioctl(2)` interface.

To manipulate properties, I/O Kit offers several functions. `IORegistryEntryCreateCFProperties()` and `IORegistryEntryCreateProperty()` may be used to retrieve a copy of the driver's entire property table, or an individual property by name. To set the property list or individual properties, corresponding `Set` functions may be used. (The corresponding `Get` functions are deprecated, superseded by their `Create` counterparts). Listing 19-4 shows how you can extend Listing 19-3 to provide more of `ioreg(8)`'s functionality:

LISTING 19-4: A property getter function for an IOService

```
void listProperties(io_service_t     Service)
{

  CFMutableDictionaryRef propertiesDict;

  kern_return_t kr = IORegistryEntryCreateCFProperties( Service,
                                                        &propertiesDict,
                                                        kCFAllocatorDefault,
                                                        kNilOptions );
  if (!kr) { fprintf (stderr,"Error getting properties..\n"); return; }

  // If kr indicates success, we have the properties as a dict. From here,
  // it's just a matter of printing the CFDictionary, in this example, as XML

  CFDataRef xml = CFPropertyListCreateXMLData(kCFAllocatorDefault,
                                              (CFPropertyListRef)propertiesDict);
  if (xml) {
        write(1, CFDataGetBytePtr(xml), CFDataGetLength(xml));
        CFRelease(xml);
        }

}
```

Many drivers export useful information through the I/O Registry. One such example is battery status. iOS developers may be familiar with the `UIDevice` class and the `UIDeviceBatteryState`, which enable getting battery properties through Objective-C and the `UIKit` framework. Similar functionality can be obtained in a quick-and-dirty way directly from the I/O Registry, by inspecting the `AppleSmartBattery` class (in OS X) or `AppleD1xxxPMUPowerSource` (in iOS, 1946 on an iPad 2, 1816 on an iPod 4G). Though these are different classes, they export the `CurrentCapacity` and `MaxCapacity` properties. Dividing the former by the latter will obtain the battery percentage. Likewise, the `isCharging`/`fullyCharged` properties provide the corresponding Boolean status indications. The `IOKit` framework also provides the IOPowerSource APIs (in the `ps.subproj` of the `IOKitUser` package) to wrap the raw I/O Registry parameters in a nicer API.

Plug and Play (Notification Ports)

A client in user mode may ask I/O Kit to notify it of any I/O Registry changes, such as the arrival (addition) and departure (removal) of devices, or a change in the state of certain devices. This is useful for adding plug and play support for devices, such as starting iTunes (and possibly iPhoto) when an i-Device is inserted.

To request notifications, a client must first create a notification port. This is an `IONotificationPort` pointer (or `IONotificationPortRef`) returned by a call to `IONotificationPortCreate`. It's opaque in user mode, but is actually hiding a Mach port.

The notification port can be registered in I/O Kit's kernel component by `IOServiceAddMatchingNotification()` (for device arrival) or `IOServiceAddInterestNotification()` (for device state change). These functions internally call `io_service_add_notification` and `io_service_add_interest_notification`, respectively. Interest notifications have a message-type argument, which is a self-explaining constant from `IOMessage.h`, as shown in Listing 19-5:

LISTING 19-5: kIOMessage constants for interest notification messages

```
#define kIOMessageServiceIsTerminated      IOKit_common_msg(0x010) // removal
#define kIOMessageServiceIsSuspended       IOKit_common_msg(0x020)
#define kIOMessageServiceIsResumed         IOKit_common_msg(0x030)
#define kIOMessageServiceIsRequestingClose IOKit_common_msg(0x100)
#define kIOMessageServiceIsAttemptingOpen  IOKit_common_msg(0x101)
#define kIOMessageServiceWasClosed         IOKit_common_msg(0x110)
#define kIOMessageServiceBusyStateChange   IOKit_common_msg(0x120)
#define kIOMessageServicePropertyChange    IOKit_common_msg(0x130)
//
// These are considered deprecated
//
#define kIOMessageCanDevicePowerOff        IOKit_common_msg(0x200)
#define kIOMessageDeviceWillPowerOff       IOKit_common_msg(0x210)
#define kIOMessageDeviceWillNotPowerOff    IOKit_common_msg(0x220)
#define kIOMessageDeviceHasPoweredOn       IOKit_common_msg(0x230)
#define kIOMessageCanSystemPowerOff        IOKit_common_msg(0x240)

//
// These are wrapped by IOPMLib's IORegisterForSystemPower
//
#define kIOMessageSystemWillPowerOff       IOKit_common_msg(0x250)
#define kIOMessageSystemWillNotPowerOff    IOKit_common_msg(0x260)
#define kIOMessageCanSystemSleep           IOKit_common_msg(0x270)
#define kIOMessageSystemWillSleep          IOKit_common_msg(0x280)
#define kIOMessageSystemWillNotSleep       IOKit_common_msg(0x290)
#define kIOMessageSystemHasPoweredOn       IOKit_common_msg(0x300)
#define kIOMessageSystemWillRestart        IOKit_common_msg(0x310)
#define kIOMessageSystemWillPowerOn        IOKit_common_msg(0x320)
```

The notification port may be listened on directly, using the Mach message primitives, or — preferably — connected to a run loop construct. Run loops are a Core Foundation programming model, which implements message loops. When a message is received on the notification port, a user-supplied callback is invoked. A good example of this can be found in the `IOKitUser` package, which contains an example program called `ionotify.c`.

I/O Kit notifications are also used (in Lion and later) by `launchd(1)`, which can be set to listen for I/O Kit matching events (by specifying a `com.apple.iokit.matching` dictionary under `LaunchEvents`) and start programs on demand (as discussed in Chapter 7).

I/O Kit Power Management

Not all devices need power management support, but for those that do, this support is very important. Power management is paramount for Apple's i-Devices, which run on a battery and must use it efficiently, because an i-Device that runs out of battery is about as useful as a brick. (Come to think of it, less so, because you wouldn't go around throwing a $600 brick.)

Drivers can register for power notifications and both respond and affect system power state transitions. Drivers requiring this functionality can be found in the `IOPower` plane, and their lineage also doubles as their power dependency. This is described in Apple's I/O Kit Fundamentals, and is thus left out of scope for this work.

User mode applications can also request involvement in Power Management. This has, in fact, been possible since the advent of OS X, albeit not as documented as is the case with drivers. Applications can register for power notifications, and even prevent system sleep or shutdown using Power Management Assertions. These are similar in principle to Android's "wakelocks," which enable a user mode program to request a hold on the device, preventing it from going to sleep. Lion provides a command-line tool called `caffeinate(8)`, whose simple source[6] shows that it is merely a simple program to call `IOPMAssertionCreateWithDescription`. This is one of the many API calls exported through IOPMLib, shown in Table 19-3:

TABLE 19-3: IOP Code

FUNCTION	USAGE
`io_connect_t` **`IORegisterForSystemPower`** `(void *refcon, IONotificationPortRef *thePortRef, IOServiceInterestCallback callback, io_object_t * root_notifier );` `IOReturn` **`IODeregisterApp`** `(io_object_t * notifier )`	Register for power management notifications. This function creates an I/O notification port and registers an `kIOAppPowerStateInterest`. The port reference is returned in `thePortRef`, with an optional callback. The refcon is an opaque identifier which should be kept for de-registration.
`IOReturn` **`IOAllowPowerChange`** `(io_connect_t kernelPort, long notificationID );` `IOReturn` **`IOCancelPowerChange`** `(io_connect_t kernelPort, long notificationID )`	Respond by allowing or canceling a power change event.
`IOReturn` **`IOPMSleepSystem`** `(io_connect_t fb );` `IOReturn` **`IOPMSchedulePowerEvent`** `(CFDateRef time_to_wake, CFStringRef my_id, CFStringRef type);`	Request system sleep, or schedule sleep, wake up, shutdown, or power on.
`IOReturn` **`IOPMAssertionCreateWithName`**`( CFStringRef AssertionType, IOPMAssertionLevel AssertionLevel, CFStringRef AssertionName, IOPMAssertionID *AssertionID);` `IOReturn` **`IOPMAssertionRelease`** `(IOPMAssertionID AssertionID)`	Create a power management assertion, and specify a textual *`AssertionName`*. The *`AssertionType`* is one of `kIOPMAssertionTypeNoIdleSleep`, `kIOPMAssertionTypeNoDisplaySleep`, etc. The *`AssertionID`* should be retained until its eventual release.

FUNCTION	USAGE
IOReturn **IOPMCopyAssertionsByProcess** *(CFDictionaryRef *AssertionsByPID)*	Show processes holding assertions (used by `pmset -g`).

Driving IOPMLib behind the scenes are Mach messages (this book holds little surprises, even as it draws to its close). The `powermanagement` subsystem is subsystem 73000, and MIG is used to generate connections, notifications, and assertions. The full list of messages can be seen in the `IOKitUser` package's `pwr_mgt.subproj/powermanagement.defs`.

Other I/O Kit Subsystems

The `IOKitUser` package contains, along side power management, other interesting subprojects, including the kext `subproj` (discussed last chapter), USB, HID, and Graphics. The latter is especially important, as it allows access to the framebuffer (graphics device memory) by communicating with the kernel's `IOGraphicsFamily`. This is useful for all sorts of nifty graphics effects, CLUT manipulation and transparent overlays (such as those which appear when pressing the volume buttons on a Mac or an i-Device). Singh's book — *Mac OS X Internals: A Systems Approach* (Addison-Wesley Professional, 2006) — has a nice example of framebuffer rotation.

I/O Kit Diagnostics

Apple provides only two diagnostic utilities outside `ioreg(8)` and the graphical `IORegistry Explorer` bundled with Xcode. The only two utilities provided are `ioallocount` and `ioclasscount`.

ioalloccount(8)

`ioalloccount(8)` takes no arguments and presents the memory consumed by I/O Kit allocations, as shown in Listing 19-6.

LISTING 19-6-A: ioalloccount on OS X

```
morpheus@ergo (/)$ ioalloccount
  Instance allocation = 0x0031c9c8 = 3186 K
  Container allocation = 0x001f9ecd = 2023 K
  IOMalloc allocation = 0x01ed5238 = 31572 K
  Pageable allocation = 0x08e55000 = 145748 K
```

On an i-Device, the numbers are lower by an order of magnitude:

LISTING 19-6-B: ioalloccount on iOS

```
root@Padishah (/) # ioalloccount
   Instance allocation = 0x00154260 = 1360 K
  Container allocation = 0x002cadd7 = 2859 K
   IOMalloc allocation = 0x00e529c2 = 14666 K
   Pageable allocation = 0x016e1000 = 23428 K
```

ioclasscount(8)

`ioclasscount(8)` counts the instances of all registered I/O Kit classes and subclasses, providing an aggregate count. This means that top-level classes get counted when they, or any subclass of theirs, get instantiated. The classes counted include the libkern classes as well, which understandably have the most instances. For example, Listing 19-7 shows an `ioclasscount` on an iPad 2, sorted by the number of instances:

LISTING 19-7: ioclasscount, sorted by the number of instances

```
root@Padishah (/) # ioclasscount | sort -t'=' -n -k 2
AppleAKM8973S = 0
AppleANX9836 = 0
..
AppleARMCHRPNVRAM = 0
AppleARMCortexGeneralPurposeCounter = 0
..
_IOServiceJob = 0
AppleA5AE2 = 1
..
IOServicePM = 49
IOCommand = 53
IOWorkLoop = 61
AppleARMIISCommand = 64
IOPMemory = 75
IOSubMemoryDescriptor = 93
OSObject = 94
AppleSimpleUARTCommand = 96
IOServiceMessageUserNotification = 100
IODMACommand = 107
IOTimerEventSource = 119
_IOServiceInterestNotifier = 120
IOService = 126
OSKext = 157
IOCommandGate = 187
IOSurfaceDeviceCache = 274
IOSurfaceClient = 276
IOSurface = 281
IOMachPort = 348
IOGeneralMemoryDescriptor = 426
IOMemoryMap = 430
IOBufferMemoryDescriptor = 509
OSSet = 567
OSArray = 2393
OSData = 2431
OSSymbol = 3031
OSDictionary = 3575
OSString = 4634
OSNumber = 5357
```

Both `ioclasscount` and `ioalloccount` merely query the `I/O KitDiagnostics` property of the registry root, as you can see in Listing 19-8:

LISTING 19-8: Isolating the IOKitDiagnostics property from the I/O Registry

```
root@Padishah (/) # ioreg -w 0 -l | grep IOKitDiagnostics
 |   "IOKitDiagnostics" = {"Instance allocation"=1363612,"IOMalloc allocation"
=14976148,"Container allocation"=2885921,"Pageable allocation"=26894336
,"Classes"={"IOSDIODevice"=1,"IOApplePartitionScheme"=0,"IOFlashTranslationLayer"=1,
"IODPAudioDriver"=0,"AppleARMIODevice"=47,"AppleEmbeddedAudioPTTFunctionButton"=0,
"AppleProfileManualTriggerClient"=0,"IOHDIXHDDriveInKernel"=1,"AppleBCMWLANTxBuffer"=10,
"M2ScalerDARTVMAllocator"=0,"IOPlatformExpertDevice"=1,"AppleS5L8930XUSBPhy"=1,
"KDIEncoding"=1,"IORangeAllocator"=17,"IOMobileFramebuffer"=1, ...
```

`IOKitDiagnostics` is, in I/O Kit terms, a dictionary of five keys: the four allocation counts (displayed by `ioalloccount(8)`) and a "classes" key, which itself contains a dictionary with however many classes are registered as its keys (and the class instances themselves count as values of the respective keys).

I/O KIT KERNEL DRIVERS

As explained earlier in this chapter, I/O Kit drivers are objects derived from a common ancestor, `IOService`. The hierarchy under `IOService` is quite rich and extensive, and along the way drivers can become more specialized and suited for the devices or buses they are meant to handle.

I/O Kit drivers are classified as either "drivers" or "nubs." A nub is, quite simply, an adapter between two drivers, representing the devices to be controlled. Drivers create nubs for every device instance they manage. This is different than the UN*X model, in which the driver "object" is identified by a major number, and the specific devices are identified by minor numbers. That model is still supported, however, for those drivers which choose to create BSD device instances (in the `/dev` file system).

Driver Matching

I/O Kit maintains a Catalogue object[2] that represents the database of all known and registered driver personalities. In this context, the term *personality* refers to one or more facets of driver functionality declared in the driver's property list, as the value of the `<IOKitPersonalities>` key, which is itself a dictionary. Each personality must declare an `IOProviderClass` key (specifying the nub it can attach to). The Catalogue is bootstrapped by calling its `initialize` method, with values from `gIOKernelConfigTables`, a global array of strings containing the `IOPanicPlaform` and the `IOPlatformExpertDevice` entries (both in `iokit/Kernel/IOPlatformExpert.cpp`). The former is used to panic the system if no `IOPlatformDevice` matches, and the latter is instantiated as the root nub in `StartIOKit()`.

I/O Kit uses driver personalities to match drivers to new devices (more accurately, newly generated nubs of discovered devices). As the provider (for example, PCI or USB) discovers a new device it publishes the device using a call to `IOService::registerService()`, which starts the driver matching process (literally, by a call to `IOService::startMatching)`. This is a three-staged process, detailed in Figure 19-2. The process can be either synchronous (same thread) or asynchronous (in an I/O Kit created `IOConfigThread`).

[2] Apple/NeXT's driver people were chiefly British, apparently, as is the spelling of "Catalogue."

The first step of the process is referred to as class matching, and is a simple filtering step that enumerates all candidate drivers, by looking a match on their `IOProviderClass`. This, however, may return many candidates. The next step therefore, is passive matching, which needs to weed out those that are spurious and irrelevant by looking at their published personalities. Each driver personally specifies matching properties, which are either generic I/O Kit properties (listed in `iokit/IOKit/IOKitKeys.h`), or provider specific, for example PCI device identifiers (`IOPCIMatch`), USB types (such as `idVendor/idProduct`) and FireWire identifiers (`Unit_SW_Version/Unit_Spec_ID`). Virtual device drivers, which specify `IOResources` as their provider class, specify an `IOMatchProperty` to avoid matching all virtual devices. Drivers may specify an optional `IOProbeScore` property to ask to be tried first, and an `IOMatchCategory` property to specify which category they belong to. (Otherwise they are all classified into the same, unnamed category.)

The properties specified in the personality help the `IOProviderClass` filter the most matching driver(s), as all criteria should be matched. If a driver is of a more generic type, it can either specify less (or broader) matching criteria, or publish additional personalities. A good example of this can be found in VMWare Fusion's kext, whose `IOKitPersonalities` keys is shown in Listing 19-9. A wildcard match (and a high `IOProbeScore`) enables Fusion's vmioplug to be the first responder when USB devices are inserted, prompting the user to redirect the device to a running instance of a virtual machine.

LISTING 19-9: Example of an IOKitPersonalities key (from VMWare Fusion)

```
...
   <key>IOKitPersonalities</key>
      <dict>
          <key>UsbDevice</key>
           <dict>
              <key>CFBundleIdentifier</key>
              <string>com.vmware.kext.vmioplug</string>
              <key>IOClass</key>
              <string>com_vmware_kext_UsbDevice</string>
              <key>IOProviderClass</key>
              <string>IOUSBDevice</string>
              <key>idProduct</key>
              <string>*</string>
              <key>idVendor</key>
              <string>*</string>
              <key>bcdDevice</key>
              <string>*</string>
              <key>IOProbeScore</key>
              <integer>9005</integer>
              <key>IOUSBProbeScore</key>
              <integer>4000</integer>
           </dict>
...
```

After ordering all potential matches, the last step is active matching, wherein I/O Kit calls, in turn, the candidate drivers' `init()` and `probe()` methods (discussed later in the section, "The I/O Kit Driver Model") to obtain the active or live probe scores. The drivers are re-ordered by their probe scores and `IOMatchCategory` (if any), and I/O Kit proceeds to start the highest-ranking driver in each category. This gives a chance to the most suitable driver to claim the device. The process repeats until the first matching driver claims success (i.e. its `start()` method returns a true value).

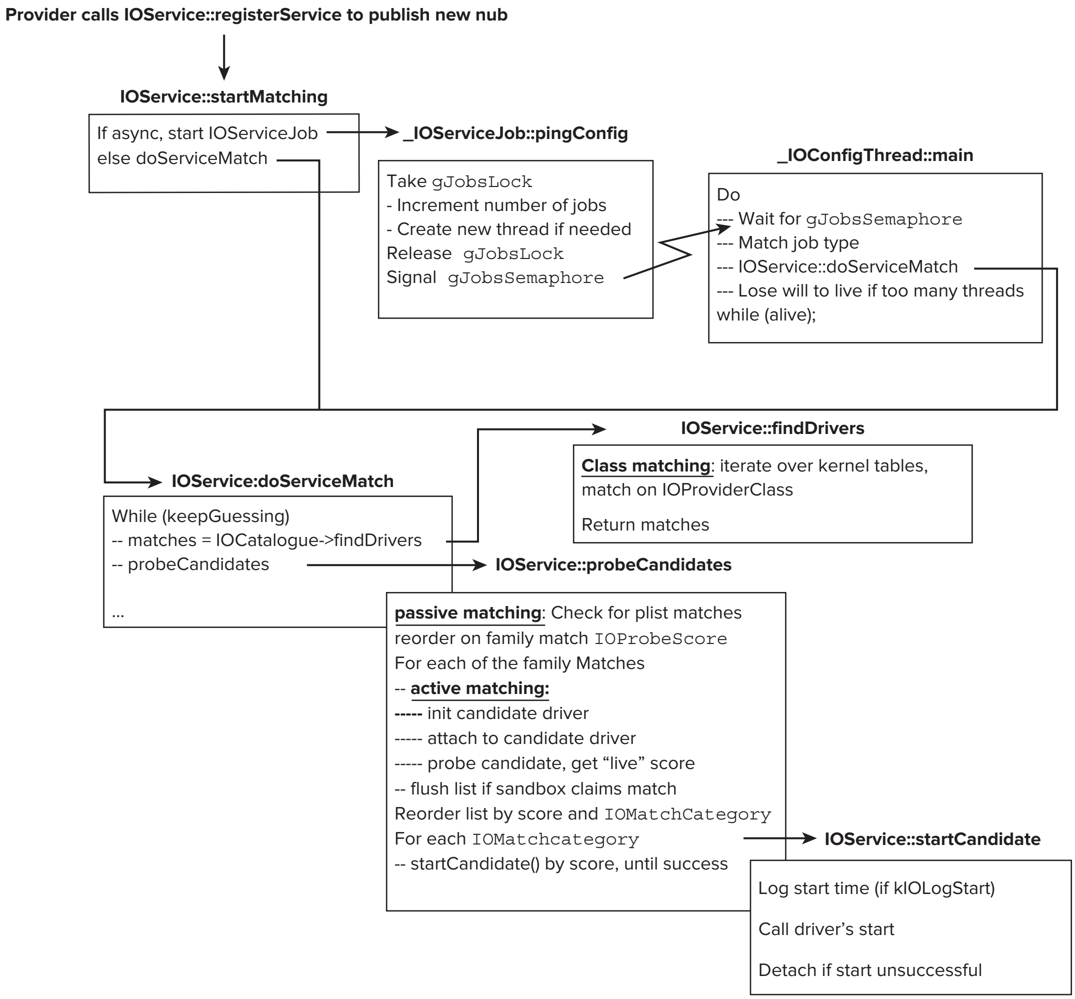


FIGURE 19-2: The I/O Kit matching process

Kernel components and other drivers can access the Catalogue programmatically and draw on its matching services. The `iokit/bsddev/IOKitBSDInit.cpp` file contains functions such as `IOCatalogueMatchingDriversPresent` (to perform a catalog search and return a Boolean indication if there are matching drivers) and `IOServiceWaitForMatchingResource` (to block its caller until a matching driver has been loaded), as well as others, which are mostly wrappers over methods from `IOService` and other I/O Kit classes.

The I/O Kit Families

Apple provides several "families," which defined abstract and concrete classes (all derived from `OSObject`). These classes implement the "typical" drivers of buses and generic device types. These include the ones shown in Table 19-4.

TABLE 19-4: The I/O Kit Generic Families

I/O KIT FAMILY	USED FOR
`IO80211Family`	Wireless Ethernet (802.11) devices
`IOACPIFamily`	Advanced Configuration and Power Interface
`IOAHCIFamily`	Advanced Host Controller Interface
`IOATAFamily`	IDE/ATA devices
`IOAudioFamily`	Generic family for all audio devices
`IOBDStorageFamily`	Bluray
`IOBluetoothFamily`	Bluetooth devices
`IOCDStorageFamily`	CD-ROM devices
`IODVDStorageFamily`	DVD-ROM devices
`IOFireWireFamily`	FireWire (IEEE 1394) devices
`IOGraphicsFamily`	Generic graphics adapters
`IOHIDFamily`	Human interface devices (keyboards, mice, the Apple Remote, and others)
`IONetworkFamily`	Generic network adapters
`IOPCIFamily`	Generic PCI devices
`IOPlatformPluginFamily`	Platform specific
`IOSCSIArchitectureModelFamily`	SCSI devices
`IOSCSIParallelFamily`	SCSI over parallel port interfaces
`IOSMBusFamily`	Intel's System Management Bus
`IOSerialFamily`	Serial port drivers
`IOStorageFamily`	Generic mass storage devices
`IOThunderboltFamily`	Thunderbolt devices (as of later Snow Leopard and Lion)
`IOUSBFamily`	Generic USB devices

Most of the families are in open source domain, as part of Darwin. This way, driver developers can draw on a large code base of examples, thereby taking a significant shortcut when developing I/O Kit drivers. The families greatly shorten the time required for development, and improve the overall stability and memory requirements of the I/O Kit drivers by calling on and reusing existing code. A driver is expected to find its nearest family member, and directly inherit from it. By doing so, much of the generic functionality can be obtained "for free." For example, a PCI device driver can take advantage of the pre-existing PCI bus logic, rather than having to re-create it from scratch. Apple Developer's I/O Kit Fundamentals guide provides detailed class hierarchies for each of its families, but we consider a specific example — that of `IONetworkingFamily` — next.

Case Study: IONetworkingFamily and adapting to DLIL

IONetworkingFamily is a wonderful example of the interoperability of I/O Kit with XNU's supporting DLIL (discussed in Chapter 17). It can be considered an adapter (in design pattern parlance, that is adapting one API to another), translating I/OKit's `IONetworkInterface` abstraction to that of the underlying DLIL's `ifnet`.

As an example, consider the case of Ethernet interfaces. `IONetworkingFamily` provides both `IONetworkInterface` (a "generic" interface abstraction) and its daughter class `IOEthernetInterface` (a more specific abstraction, but common to all Ethernet interfaces). Recall from Chapter 17, that during the initialization of XNU's interface "object," the `struct ifnet`, a driver must fill an `ifnet_init_params` structure. `IONetworkingFamily` provides the `initIfnetParameters` method, as shown in Figure 19-3:

IOEthernetInterface::initIfnetParams (struct ifnet_init_params)

```
super::initIfnetParams( params );
// fill in ethernet specific values
params->uniqueid = uniqueID->getBytesNoCopy();
params->uniqueid_len = uniqueID->getLength();
params->family = APPLE_IF_FAM_ETHERNET;
params->demux = ether_demux;
params->add_proto = ether_add_proto;
params->del_proto = ether_del_proto;
params->framer = ether_frameout;
params->check_multi = ether_check_multi;
params->broadcast_addr = ether_broadcast_addr;
params->broadcast_len = sizeof(ether_broadcast_addr);
```

IONetworkInterface::initIfnetParams

```
// Common shims to all interfaces
params->name = (char *)getNamePrefix();
params->type = _type;
params->unit = _unit;
params->output = output_shim;
params->ioctl = ioctl_shim;
params->set_bpf_tap = set_bpf_tap_shim;
params->detach = detach_shim;
params->softc           = this;
```

bsd/net/kpi_interface.h

```
struct ifnet_init_params {
const void *uniqueid;
u_int32_t   uniqueid_len;
const char  *name;
u_int32_t               unit;
ifnet_family_t          family;
u_int32_t               type;
ifnet_output_func       output;
ifnet_demux_func        demux;
ifnet_add_proto_func    add_proto;
ifnet_del_proto_func    del_proto;
ifnet_check_multi       check_multi;
ifnet_framer_func       framer;
void                    *softc;
ifnet_ioctl_func        ioctl;
ifnet_set_bpf_tap       set_bpf_tap;
ifnet_detached_func     detach;
ifnet_event_func        event;
const void      *broadcast_addr;
u_int32_t               broadcast_len;
```

FIGURE 19-3: The initIfNetParameters method in IONetworkFamily classes

Thanks to I/OKit's inheritance, `IOEthernetInterface` first calls on its parent class (`IONetwork Interface`) to set the common fields to all interfaces, such as the `ioctl` and BPF handlers. The Ethernet specific parameters (broadcast addresses, demux, framing, etc.) can then be set as well. Note, in particular, the setting of `ifnet` structure's `ifnet_*_func` pointers calls to the shims provided by I/O Kit. Between them, the two functions populate all the necessary fields of the `ifnet_init_params` structure.

This pattern is followed in the `attachToDataLinkLayer` method, which is responsible for allocating and attaching the underlying `ifnet` structure (and is responsible for calling `initIfnetParameters`), as shown in Figure 19-4:

IOEthernetInterface::attachToDataLinkLayer(IOOptionBits options,void *parameter)

```
ret=super::attachToDataLinkLayer (options, parameter);
if (ret == kIOReturnSuccess ) {
ifnet_set_baudrate( getIfnet(), 10000000); //FIXME..
bpfattach( getIfnet(), DLT_EN10MB, sizeof(struct ether_header));
}
```

IONetworkInterface::attachToDataLinkLayer

```
memset(&iparams, 0, sizeof(iparams));
initIfnetParams(&iparams);
if (ifnet_allocate( &iparams, &_backingIfnet))
    return kIOReturnNoMemory;
_syncToBackingIfnet();
if ((!ll_addr || (ll_addr->sdl_alen != 0)) &&
(ifnet_attach(_backingIfnet, ll_addr) == 0))
{
   ret = kIOReturnSuccess;
}
else{ // error condition, clean up
     ifnet_release(_backingIfnet);
     backingIfnet = NULL;
}
```

FIGURE 19-4: The attachToDataLinkLayer method in IONetworkingFamily classes

If you flip back a few pages and compare this to the UTUN case study in Chapter 17 (in particular, Figure 17-16), you will see that the very same functionality required for setting up an interface in that example has been matched by I/O Kit, through abstraction and object orientation.

`IONetworkingFamily` also ties to DLIL in two other important locations: packet reception and transmission. `IONetworkInterface::init` calls the `registerOutputHandler` method on the `IONetworkController`'s `outputPacket` function. The `IONetworkInterface::initIfnetParams` method, shown earlier, ties the underlying `struct ifnet`'s `ifnet_output` function to `IONetworkInterface`'s `output_shim`, which forwards the packet (read: mbuf) to the `outputPacket` handler. A driver is expected to override this function (whose default implementation merely drops all packets), and supply its own transmission logic.

Packet reception is implemented similarly: `IONetworkInterface` supplies two methods: `inputPacket` and `flushInputQueue`, which the implementing subclass is expected to call (from its work loop, when processing an interrupt). The inputPacket method passes the packet to BPF filters, if any, then enqueues it and calls `DLIL_INPUT`, passes the packet (i.e. mbuf chain) to `ifnet_input`. From there, processing continues as described in Chapter 17. This is shown in Figure 19-5:

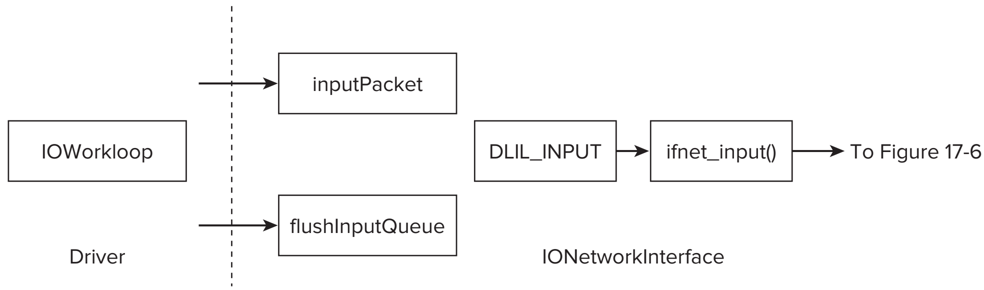


FIGURE 19-5: Packet reception in IONetworkFamily

The case study ends here, but the object orientation does not; Other families can inherit from `IONetworkingFamily`, and extend this functionality even further. Figure 19-6 depicts classes which rely on `IONetworkingFamily`. One important family branch is `IO80211Family`, which provides wireless Ethernet functionality. Apple's AirPort drivers (all as "plugins" of that family) inherit from `IO80211Interface` and `IO80211Controller`. To examine the implementation of a full Ethernet driver, check out Apple's Network Device Driver Programming Guide[7] and its AppleUSBCDCDriver[8].

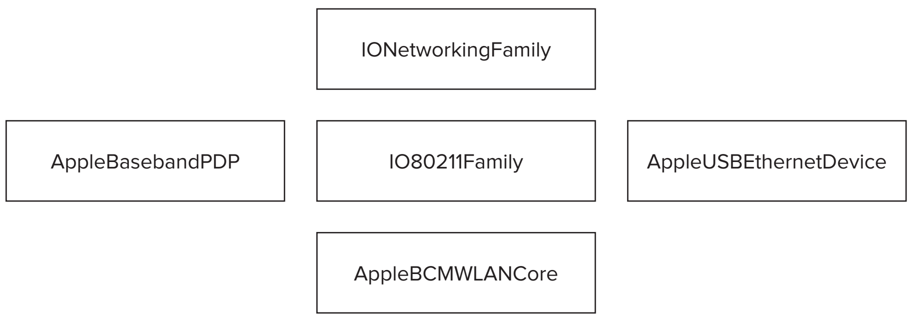


FIGURE 19-6: Descendants of IONetworkingFamily

The I/O Kit Driver Model

Irrespective of which family a driver is derived from, it is the eventual descendant of `IOService`. By virtue of this inheritance, an I/O Kit driver is expected to conform to a set interface and required to implement a very specific set of callbacks that correspond to milestones in its lifetime, as shown in Table 19-5:

TABLE 19-5: I/O Kit Driver Functions

FUNCTION (DRIVER ENTRY POINT)	CALLED WHEN
`bool init (OSDictionary * properties)`	The driver is first initialized.
`void free(void)`	The driver is unloaded. This is the anti-function of `init()` and is expected to undo everything `init()` has done.
`bool attach (IOService *provider);`	The driver is being attached to a nub, for probing or activation.
`void detach (IOService *provider);`	The driver is being detached from a nub, after probing or following close.
`IOService *probe (IOService *provider,; int *score);`	I/O Kit performs a probe for the device in question, to see whether it exists. Return pointer to `IOService` object representing driver, and populate score. If this function is omitted, the driver's default score, from its Plist, is returned.
`bool start ( IOService *provider )`	The driver is started by I/O Kit. Marks driver as active. Driver can publish its nubs.
`bool stop (IOService *provider)`	The driver is stopped by I/O Kit. Marks driver as inactive. Driver is expected to recall any nubs published.
`bool open (IOService *forClient, IOOptionBits options, void * arg);`	Driver is opened for use.
`void close (IOService *forClient, IOOptionBits options);`	Driver is released.
`IOReturn message (UInt32 type, IOService * provider, void * argument = 0 )`	Notification messages from other drivers.

There is a very specific order to the function calls, however, which is what I/O Kit considers to be the driver's lifecycle, as shown Figure 19-7.

A driver automatically inherits the lifecycle functions from its superclass (`IOService`), but may implement them as well, effectively overriding them. To ensure safety, however, any such implementation is expected to call the corresponding implementation of the superclass (i.e. extending, rather than overriding the methods).

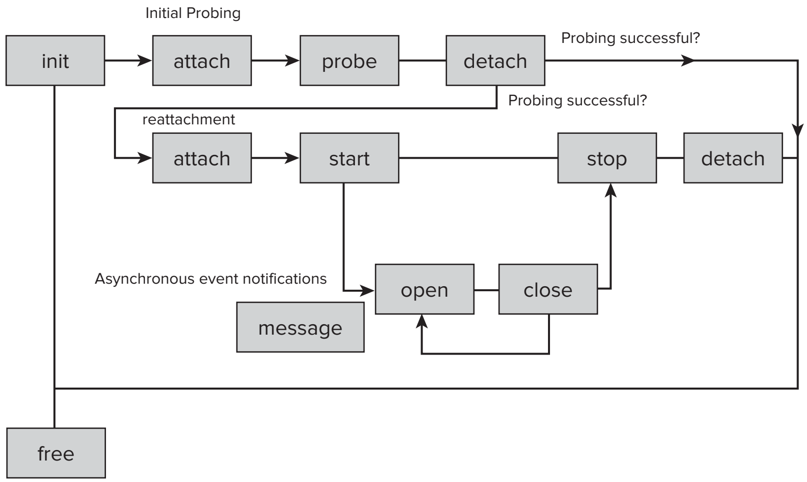


FIGURE 19-7: I/O Kit driver state machine

For example, consider `init()`: The driver is expected to implement its own initialization function, which is called when the driver is first loaded. This can be used for any driver-specific setup. Because the driver is a subclass of some other driver, it is expected to call its superclass `init` function first. This is usually something following the pattern in Listing 19-10:

LISTING 19-10: Sample I/O Kit driver init() function

```
bool sampleDriver::init(IOPhysicalAddress * paddr)
{
   bool rc = super::init(); // MUST call superclass before doing anything
   if (!rc) return (rc); // return FALSE to caller if super failed
   // Do own initialization
   return(false);
}
```

If the driver has nothing to do, the function body can either be left empty, or the function can be left unimplemented. Looking at the state machine, you can see another unusual trait of the I/O Kit callbacks, and that is in their coupling: A call to `init()` ensures an eventual call to `free()`, a call to `attach()` ensures a call to `detach()`, and `start()` is met by an eventual `stop()`.

> *By using the* `debug` *boot argument (or* `sysctl(8)` *on* `debug.iokit` *and* `debug.iotrace`*) you can ask XNU to log all IOKit operations. Specific flags are described in* `IOKit/IOKitDebug.h`*. Be careful with this, however! Setting all flags (*`0xFFFFFFFF`*) will likely cause a kernel panic.*

The IOWorkLoop

I/O Kit adopts the NeXT *runloop* model, familiar to user mode developers as the `CFRunLoop`. I/O Kit's version of the `runloop` is called `IOWorkloop`, and it follows the same basic idea: providing a single, thread-safe mechanism to handle all sorts of events that would otherwise be asynchronous. Access to the work loop is protected by a mutex, alleviating concerns of reentrancy and thread safety. Note, however, there is no guarantee that a work loop is, indeed, a thread. That is, the work loop iteration may be run in the context of another thread in the system. The work loop iteration is therefore always self-contained.

The driver can opt to join its provider's work loop (by calling `getWork Loop`), or create its own (by calling `IOWorkLoop::work Loop()`), which may be further exported to any of its subclasses. In practice most drivers opt to join their provider's. The driver can register any number of various event sources whose events it will handle by calling its `IOWork Loop::addEventSources` method. These are all subclasses of `IOEventSource`, and include the event sources shown in Table 19-6.

TABLE 19-6: Event Sources in IOWorkLoops

EVENT SOURCE	USED FOR
`IOCommandGate`	Commands from clients, or from power management
`IOInterruptEventSource` `IOFilterInterruptEventSource`	Interrupts, both dedicated and shared
`IOTimerEventSource`	Periodic timer events, watchdogs

The `IOWorkLoop` has a surprisingly simple and efficient implementation (at least, compared to earlier versions of OS X), using Mach continuations, as shown in Listing 19-11:

LISTING 19-11: The IOWorkloop implementation:

```
/* virtual */ void IOWorkLoop::threadMain()
{
restartThread:
    do {
        // Iterate through all work loop event sources. If we have none, bail.
        // runEventSources will also set "workToDo" to false, but the
        // IOWorkloop:signalWorkAvailable() may be called at any time and reset
        // it to true.
        if ( !runEventSources() )
            goto exitThread;

        IOInterruptState is = IOSimpleLockLockDisableInterrupt(workToDoLock);

        // If we get here and no more work (workToDo = FALSE), we check the
        // kLoopTerminate flag. If it is not set, we restart. Otherwise, we skip
        // this part and continue to exit.
        if ( !ISSETP(&fFlags, kLoopTerminate) && !workToDo) {
```

```
                assert_wait((void *) &workToDo, false);
                IOSimpleLockUnlockEnableInterrupt(workToDoLock, is);
                thread_continue_t cptr = NULL;

                // If possible, set threadMain as our own continuation and block
                // otherwise, leave continuation null and use "goto" for same effect
                if (!reserved || !(kPreciousStack & reserved->options))
                    cptr = OSMemberFunctionCast(
                            thread_continue_t, this, &IOWorkLoop::threadMain);
                thread_block_parameter(cptr, this);
                goto restartThread;
                /* NOTREACHED */
            }

            // At this point we either have work to do or we need
            // to commit suicide.  But no matter
            // Clear the simple lock and retore the interrupt state
            IOSimpleLockUnlockEnableInterrupt(workToDoLock, is);

        } while(workToDo);

exitThread:
    // We get here if no sources, or no more work and loop flags had kLoopTerminate
    thread_t thread = workThread;
    workThread = 0;     // Say we don't have a loop and free ourselves
    free();

    thread_deallocate(thread);
    (void) thread_terminate(thread);
}
```

Interrupt Handling

Although some device drivers are for virtual devices, the majority of drivers have to deal with real hardware, and — in doing so — with interrupts. I/O Kit does a fabulous job of hiding the interrupt handling logic of Mach from the driver developer, proving once more that ignorance is bliss. Rather than be bogged down in the quagmire of interrupt specifics, I/O Kit provides an object-oriented view of interrupts that is both efficient and intuitive.

The Driver View

The main object in the I/O Kit interrupt model is that of an `InterruptEventSource`, which, as is evident by Table 19-6 and the class name, is a subclass of `IOEventSource`. This is, as far as work loops are concerned, "just another" event source, enabling the driver to treat interrupts with the same work loop logic it applies to timers and event notifications.

The interrupts of the `InterruptEventSource`, however, aren't interrupts in the full sense of the word, but rather a safer kind of deferred interrupts. I/O Kit distinguishes between *primary* (direct) interrupts, wherein the handler runs with further interrupts blocked (effectively as part of Mach's interrupt handling) and secondary (indirect) interrupts where interrupts are enabled. In other words, secondary interrupts are signaled after a low-level handler acknowledges the interrupt, re-enables its line, and wakes up the driver's thread, to allow the driver's work loop to process the interrupt. This is somewhat akin to Linux's "bottom half" concept (in particular, the SoftIRQ), that Linux device drivers can schedule in the "top half" (the driver's interrupt service routine).

Direct interrupts are effectively the highest priority in the system, as they run in "raw" interrupt context, when the CPU processes the low-level trap which preempts the then-executing thread (i.e. as a call from iOS's `fleh_irq` or OS X's `interrupt()`, as discussed in Chapter 8). Apple strongly discourages the use of primary interrupts due to their time-critical nature, and documents them only briefly in the context of developing PCI drivers[9]. For all other purposes, Apple endorses the secondary interrupts. Secondary interrupts are much safer and are still of relatively high priority, but trail behind real time threads, timers, and paging events.

A special case to consider is when interrupt lines are shared between multiple interrupt sources. Drivers that are aware of that sharing can opt to register an `IOFilterInterruptEventSource`, instead of the usual `IoInterruptEventSource`. The filter interrupt event source constructor is provided with two callback functions: The first, to check whether their driver is indeed responsible for the device (returning a Boolean), and the second, to handle the interrupt if it is indeed within their responsibility (i.e. the filter returned true). The filter routine actually runs in the primary interrupt context, but is meant to merely check the interrupt source, and not process it. If the filter function returns `true`, the secondary interrupt is signaled and the handler function is invoked in the driver's work loop context:

A non-conforming I/O Kit driver may "cheat" and handle an interrupt in the primary context, by doing more work in the `IOFilterInterruptEventSource`'s filter function. To dissuade developers from doing so, Apple allows them to explicitly request a direct interrupt using the `IOService::registerInterrupt` method. The function is defined in `iokit/IOKit/IOService.h` as shown in Listing 19-12:

LISTING 19-12: IOService::registerInterrupt

```
/*!@function registerInterrupt
   @abstract Registers a C function interrupt handler for a device supplying interrupts.
   @discussion This method installs a C function interrupt handler to be called at
    primary interrupt time for a device's interrupt. Only one handler may be installed
    per interrupt source. IOInterruptEventSource provides a work loop based abstraction
    for interrupt delivery that may be more appropriate for work loop based drivers.
   @param source The index of the interrupt source in the device.
   @param target An object instance to be passed to the interrupt handler.
   @param handler The C function to be called at primary interrupt time when the
    interrupt occurs. The handler should process the interrupt by clearing the interrupt
     or by disabling the source.
   @param refCon A reference constant for the handler's use.
   @result An IOReturn code.
     kIOReturnNoInterrupt is returned if the source is not valid;
     kIOReturnNoResources is returned if the interrupt already has an installed handler.
  */

    virtual IOReturn registerInterrupt(int source, OSObject *target,
                                       IOInterruptAction handler,
                                       void *refCon = 0);
```

Let the driver beware, however: Executing in primary interrupt context is so time critical that even calls to `IOLog` are considered unsafe.

Behind the Scenes

The driver's view of interrupts shows just how well I/O Kit hides the underlying kernel logic supporting interrupts. Interrupt handling is not only among the most critical code paths in any kernel, but is highly machine dependent. Elegant object orientation abstracts these aspects, and enables Apple to share similar, if not identical logic between the two platforms. (See Figure 19-8.)

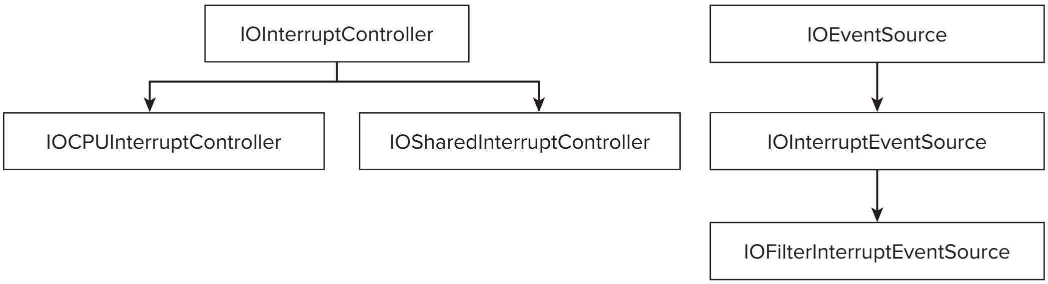


FIGURE 19-8: I/O Kit classes involved with interrupt handling

The `IOService::registerInterrupt()` method called by drivers for primary interrupts looks up the `IOInterruptController` instance. This is usually an instance of `IOCPUInterruptController`, or that of the Platform kext. The function then proceeds to call the controller's `registerInterrupt` method, passing along the `this` object reference and the arguments it was given.

`IOCPUInterruptController` ties I/O Kit to Platform Expert, but indirectly — that is, through the ml layer. When an interrupt is received, it is first handled by the machine specific handlers — `hndl_allintrs` on Intel, and `fleh_swi` on ARM. Chapter 8 discusses this low-level interrupt logic on both platforms, but stops short of discussing what happens when interrupts are passed to the Platform Expert.

As shown in Listing 8-4 and Figure 8-6, the Platform Expert's `PE_incoming_interrupt()` is invoked from the generic handler `interrupt(osfmk/i386/trap.c)` if the interrupt in question is found to be a device interrupt (and not a LAPIC one). The Platform Expert merely calls the corresponding interrupt handler from the `i386_interrupt_handler` structure. This is shown in Listing 19-13:

LISTING 19-13: Platform Expert Interrupt Handling, from pexpert/i386/pe_interrupt.c

```
struct i386_interrupt_handler {
        IOInterruptHandler      handler;
        void                    *nub;
        void                    *target;
        void                    *refCon;
};

typedef struct i386_interrupt_handler i386_interrupt_handler_t;

i386_interrupt_handler_t        PE_interrupt_handler;

void
PE_incoming_interrupt(int interrupt)
{
       i386_interrupt_handler_t        *vector;
      // Code also contains DTRACE/DEVELOPMENT INT5 hooks
```

continues

LISTING 13-13 *(continued)*

```
        vector = &PE_interrupt_handler;
        vector->handler(vector->target, NULL, vector->nub, interrupt);
    }
```

The `PE_interrupt_handler` is a singleton. The Platform Expert exports a special function, `PE_install_interrupt_handler`, which can be used to set its fields. This function is wrapped by `void ml_install_interrupt_handler` (`osfmk/i386/machine_routines.c`), which is also exported and invoked by `IOCPUInterruptController::enableCPUInterrupt`.

In iOS the structure is largely the same, with minor exceptions outside the scope of this book. Figure 19-9 shows the iOS disassembly of `void ml_install_interrupt_handler`, decompiled using the OS X source. This is aligned with `fleh_irq`, which is the (rough) equivalent in iOS of OS X's `interrupt()`, and inlines `PE_incoming_interrupt()`. Without getting bogged down in ARM assembly, suffice it to say that while the installation and invocation of the interrupt handler is not identical to OS X, it is nonetheless highly similar (did we not say that ignorance is bliss?)

```
; void ml_install_interrupt_handler(void *nub,
;        int source,
;        void *target,
;        IOInterruptHandler handler,
;        void *refCon);
;
0x8007B794 PUSH    {R4-R7,LR}
0x8007B796 ADD     R7, SP, #0xC
0x8007B798 STR.W   R8, [SP,#0xC+savedR8]!
0x8007B79C MOV     R5, R3 ; R5 = handler
0x8007B79E MOV     R8, R2 ; R8 = target
0x8007B7A0 MOV     R6, R1 ; R6 = source
0x8007B7A2 MOV     R4, R0 ; R4 = nub
 ; current_state = ml_get_interrupts_enabled
0x8007B7A4 BLX     _ml_get_interrupts_enabled
 ; PE_install_interrupt_handler (…) inline
 ; OS X uses vector = &PE_Interrupt_Controller.
 ; But iOS gets the vector from CPU data (R1)
 ;    vector->handler = handler;
 ;    vector->nub = nub;
 ;    vector->target = target;
 ;    vector->refCon = refCon;
0x8007B7A8 MRC     p15, 0, R1,c13,c0, 4
0x8007B7AC LDR     R2, [R7,#8]    ; 5th arg
0x8007B7AE LDR.W   R1, [R1,#0x4B8]; vector
0x8007B7B2 ADD.W   R3, R1, #0xC0
0x8007B7B6 STR.W   R5, [R1,#0xBC] ; handler
;
; One ARM inst stores nub,refcon, target
;
0x8007B7BA STMIA.W R3, {R4,R6,R8} ; C0,C4,C8
0x8007B7BE STR.W   R2, [R1,#0xCC] ; 5th arg
0x8007B7C2 MOVS    R2, #1
0x8007B7C4 STR     R2, [R1,#0x1C]
;
; Note, current_state is still in R0:
; ml_set_interrupts_enabled(current_state)
0x8007B7C6 BLX     _ml_set_interrupts_enabled
;
; initialize_screen(NULL, kPEAcquireScreen);
0x8007B7CA MOVS    R0, #NULL
0x8007B7CC MOVS    R1, kPEAcquireScreen
; ...
0x8007B7D6 B.W     _initialize_screen
```

```
fleh_irq: // q.v. interrupt(), osfmk/i386/trap.c
0x8007967C  SUB LR, LR, #4
; Set CPSR Interrupt flag
0x80079680  MRS SP, CPSR
0x80079684  BIC SP, SP, #0x100
0x80079688  MSR CPSR_x, SP
;
; ... lots of irrelevant stuff omitted
;
0x80079778 LDR   R8, =_kdebug_enable
0x8007977C LDR   R8, [R8]
0x80079780 MOVS  R8, R8 ; tests kdebug_enable
0x80079784 MOVNE R0, R5
0x80079788 BLNE  do_kdebug_EXCP_INTR_FUNC_START
0x8007978C BL    SCHED_STATS_INTERRUPT
;
; v->handler(v->target,.., v->nub, interrupt);
;
0x80079790 MRC  p15, 0, R9,c13,c0, 4
0x80079794 LDR  R4, [R9,#0x4B8] ; vector
0x80079798 STR  R5, [R4,#0xB8]
0x8007979C LDR  R3, [R4,#0x16C] ; Load count
0x800797A0 ADD  R3, R3, #1      ; Increment
0x800797A4 STR  R3, [R4,#0x16C] ; store count
0x800797A8 LDR  R0, [R4,#0xC8]  ; target
0x800797AC LDR  R1, [R4,#0xCC]
0x800797B0 LDR  R2, [R4,#0xC0]  ; nub
0x800797B4 LDR  R3, [R4,#0xC4]
0x800797B8 LDR  R5, [R4,#0xBC]  ; handler
0x800797BC BLX  R5 ; handler(target,...,nub,..)
;
; KERNEL_DEBUG_CONSTANT (MACHDBG_CODE( …
;
0x800797C0 MOVS R8, R8 ; test kdebug_enable
0x800797C4 BLNE do_kdebug_EXCP_INTR_FUNC_END;
; ...
```

FIGURE 19-9: ml_install_interrupt_handler and fleh_irq from iOS aligned

I/O Kit Memory Management

I/O Kit wraps Mach's kernel memory management calls with its own. Although Mach has its various memory management APIs (discussed in Chapter 12), the preferred mode of work is to use solely the I/O Kit `new` and `delete` operators, as well as the `IO*` wrappers.

The Memory management APIs offered by I/O Kit are shown Table 19-7.

TABLE 19-7: The I/O Kit Memory Allocation Methods

MEMORY MANAGEMENT API	WRAPS MACH API	USED FOR
`New` `Delete`	`kalloc` `kfree`	C++ objects
`IOMalloc` `IOFree`	`kalloc` `kfree`	I/O Kit `malloc()/free()` replacement
`IOMallocAligned` `IOFreeAligned`	`kernel_memory_allocate`	Allocates/frees memory with specific alignment requirements
`IOMallocContiguous` `IOFreeContiguous`	`kmem_alloc_contig`	Allocates/frees contiguous free memory (deprecated)
`IOMemoryDescriptor`	Various	Recommended (supersedes `IOMallocContiguous`)

Mixing and matching methods is obviously a bad idea, and each allocation must be freed with its matching function.

Additional classes such as `IODMACommand`, can be used for physical memory and DMA access. This class (which supersedes `IOMemoryCursor`) is itself a subclass of `IOCommand`, which is a generic class for controller related commands (such as ATA and SCSI).

BSD INTEGRATION

As discussed in this Chapter, I/O Kit presents a rich set of APIs to user mode. This, however, can lead to a problem when porting UN*X applications, which still use the BSD device interfaces of `/dev`. XNU therefore supports the traditional concepts of block and character devices (as well as network interfaces, as shown in Chapter 17), and even the BSD-specific structures of `bdevsw` and `cdevsw`.

Aside from a few in-memory devices, however, the logic in the kernel which supports these devices isn't XNU, but I/O Kit: In particular, the `IOStorageFamily.Kext`, which is responsible for handling mass storage devices, and the `IOSerialFamily.Kext`, which is responsible for serial ports, contain specialized classes, (called `IOMediaBSDClient` and `IOSerialBSDClient`, respectively. Lion's `CoreStorage.kext` likewise contains a `CoreStorageBSDClient`). These classes create and remove `/dev` entries on the fly when new volumes are attached or removed from the system. The end result is a dynamic /dev directory that reflects the current state of connected devices, albeit implemented differently than Linux's udevd. Example code from IOSerialBSDClient, which creates character devices for serial terminals, is shown in Listing 19-14:

LISTING 19-14: Initialization of BSD character devices in IOSerialBSDClient (IOSerialFamily-59)

```
// Provide a BSD layer compatible cdevsw structure, by populating all the
// system call handlers expected by BSD with those of the I/O Kit class
struct cdevsw IOSerialBSDClient::devsw =
{
    /* d_open     */ IOSerialBSDClient::iossopen,
    /* d_close    */ IOSerialBSDClient::iossclose,
    /* d_read     */ IOSerialBSDClient::iossread,
    /* d_write    */ IOSerialBSDClient::iosswrite,
    /* d_ioctl    */ IOSerialBSDClient::iossioctl,
    /* d_stop     */ IOSerialBSDClient::iossstop,
    /* d_reset    */ (reset_fcn_t *) &nulldev,
    /* d_ttys     */ NULL,
    /* d_select   */ IOSerialBSDClient::iossselect,
    /* d_mmap     */ eno_mmap,
    /* d_strategy */ eno_strat,
    /* d_getc     */ eno_getc,
    /* d_putc     */ eno_putc,
    /* d_type     */ D_TTY
};


// Constructor adds a devsw for TTYs
IOSerialBSDClientGlobals::IOSerialBSDClientGlobals()
{
     // ...
     // Initialization of various globals
     // ...
    fMajor = (unsigned int) -1;              // request dynamic major
    fNames = OSDictionary::withCapacity(4);
    fLastMinor = 4;                          // four minor devices
    fClients = (IOSerialBSDClient **)
                IOMalloc(fLastMinor * sizeof(fClients[0]));

    if (fClients && fNames) {
        bzero(fClients, fLastMinor * sizeof(fClients[0]));  // memset to zero
        fMajor = cdevsw_add(-1, &IOSerialBSDClient::devsw); // assign major
        cdevsw_setkqueueok(fMajor, &IOSerialBSDClient::devsw, 0); // enable
    }
    if (!isValid())
        IOLog("IOSerialBSDClient didn't initialize");
}
// Destructor removes the devsw added
IOSerialBSDClientGlobals::~IOSerialBSDClientGlobals()
{
 ...  // removal of all globals
```

```
    if (fMajor != (unsigned int) -1)
         cdevsw_remove(fMajor, &IOSerialBSDClient::devsw);
  ...
}

bool IOSerialBSDClient::createDevNodes()
{
       // ...
       // Create the device nodes
       //
        calloutNode = devfs_make_node(fBaseDev | TTY_CALLOUT_INDEX,
            DEVFS_CHAR, UID_ROOT, GID_WHEEL, 0666,
            (char *) calloutName->getCStringNoCopy() +
                     (uint32_t)sizeof(TTY_DEVFS_PREFIX) - 1);

        dialinNode = devfs_make_node(fBaseDev | TTY_DIALIN_INDEX,
            DEVFS_CHAR, UID_ROOT, GID_WHEEL, 0666,
            (char *) dialinName->getCStringNoCopy() +
                     (uint32_t)sizeof(TTY_DEVFS_PREFIX) - 1);
        if (!calloutNode || !dialinNode)
            break;

}
```

Thanks to I/O Kit inheritance, storage and serial devices can simply inherit from the Apple provided families, wherein all the BSD code is already nicely implemented and hidden.

SUMMARY

This chapter provided a thorough introduction to the wonderful world of I/O Kit, Apple's runtime environment for device drivers, which is a unique part of XNU. This chapter focused on I/O Kit from an architectural perspective, and not on the specific drivers. The various families, particularly USB and PCI, contain even more intricate and complicated classes than those hard coded into XNU. I/O Kit drivers can be accessed and queried from user mode over Mach messages, a property which forms the basis for many of Apple's frameworks (like IOSurface) which communicate with hardware.

REFERENCES AND FURTHER READING

1. Apple Developer, "I/O Kit Fundamentals," `https://developer.apple.com/library/mac/#documentation/devicedrivers/conceptual/IOKitFundamentals`
2. Apple Developer, "I/O Kit Device Driver Design Guidelines," `https://developer.apple.com/library/mac/#documentation/DeviceDrivers/Conceptual/WritingDeviceDriver/Introduction/Intro.html`
3. Halvorsen & Clarke, *OS X and iOS Kernel Programming*. APress, 2011

4. `I/O KitLib.h` — The user mode `I/O Kit.Framework` header
5. Apple Developer, "Accessing Hardware from Applications," `https://developer.apple.com/library/mac/#documentation/DeviceDrivers/Conceptual/AccessingHardware/`
6. Darwin Open Source, Caffeinate(8) source, `http://opensource.apple.com/source/PowerManagement/PowerManagement-271.25.8/caffeinate/caffeinate.c`
7. Apple Developer, "Network Device Driver Programming Guide," `https://developer.apple.com/library/mac/#documentation/DeviceDrivers/Conceptual/NetworkDriver/`. This guide has been "in a preliminary stage of completion" since 2008, but provides a good overview of interfacing with `IONetworkingFamily`.
8. Apple USB CDC Driver, `http://www.opensource.apple.com/darwinsource/tarballs/apsl/AppleUSBCDCDriver-314.4.1.tar.gz`
9. Apple Developer, "Writing PCI Drivers" and "Taking Primary Interrupts," `https://developer.apple.com/library/mac/#documentation/DeviceDrivers/Conceptual/WritingPCIDrivers/`

APPENDIX

Welcome to the Machine

Throughout this book, most of the samples of code are in C. Sometimes, however, especially in examples of code from the kernel core or from iOS, the excerpts are given in assembly. Maximum effort has been given to annotate the listings as much as possible, but in some cases you could find yourself wondering about the particular role or meaning of a register.

This appendix provides a bird's eye view of both Intel and ARM architectures and assembly languages. By no means anywhere near comprehensive, this appendix is not meant to replace the architecture manuals of Intel[1] (whose 64-bit architecture actually follows AMD[2]) and ARM[3] with their many pages of detail. The Intel architecture is fairly well documented, and at least one great reference exists for ARM[4]. This appendix, however, is meant to hopefully save you a time-consuming lookup of commonly used commands and registers, especially as it pertains to their usage in OS X and iOS.

DRAMATIS PERSONAE: REGISTERS

Virtually every CPU, irrespective of vendor, makes use of registers to hold immediate values of variables and constants required for various arithmetic and logical operations. The registers and their conventional purpose, however, differs between architectures.

Intel

Intel's current architecture dates back to the olden days of the 8086 and the 8-bit architecture. On 32-bit architectures, the program is limited to using only four general-purpose registers (EAX through EDX). In 64-bit architectures, R8 through R15 are added, and EAX through EDX can be used in 64-bit mode (i.e. as RAX through RDX).

Table A-1 lists the registers on the 64-bit architecture, and their traditional usage.

TABLE A-1: 64-Bit Registers on the Intel x86_64 Architecture

REGISTER	USED FOR
RAX	Accumulator. Used as a general purpose register. This is the only register that does not need to be saved by a function before use, and it is expected to hold the function's return value.
RBX	Base. Used as a general purpose register.
RCX	Counter. Used as a general purpose register. Some loop commands (REP) will decrement RCX and repeat as long as its value is not zero.
RDX	Data. Used as general purpose register.
RSI	Source Index for copy operations. Used in 64-bit architecture for parameter passing.
RDI	Destination Index for copy operations. Used in 64-bit architecture for parameter passing.
RBP	Base pointer (if enabled in program).
RSP	Stack Pointer.
R8-R15	General purpose registers. R8 and R9 used for parameter passing.
RIP	Instruction pointer. Points to the next program to execute.
CS	Code Segment. Also holds the Intel "ring" level in two bits: 00 (=0) through 11 (=3).
DS	Data Segment
ES	Extra Segment. Largely unused in OS X.
FS	Far Segment. Largely unused in OS X.
GS	General Segment. Kernel/User transition (using `swapgs` instruction).
SS	Stack Segment.

Other registers include the various table registers (IDTR, GDTR, etc.), but they are rarely of any interest outside of the very startup of XNU, wherein they are initialized.

Floating Point Registers

In addition to the common registers, Intel architectures also support floating-point optimized registers, called XMM registers. These are numbered XMM0 through XMM7. They are rarely used in the kernel, however, and are thus not of particular interest.

The EFLAGS/RFLAGS Register

There is an additional register in Intel architectures, known as the EFLAGS (32-bit) or RFLAGS (64-bit). Most of the 64-bit fields are "reserved," meaning they are (at least at present) unused. Figure A-1 presents the important flags in this register.

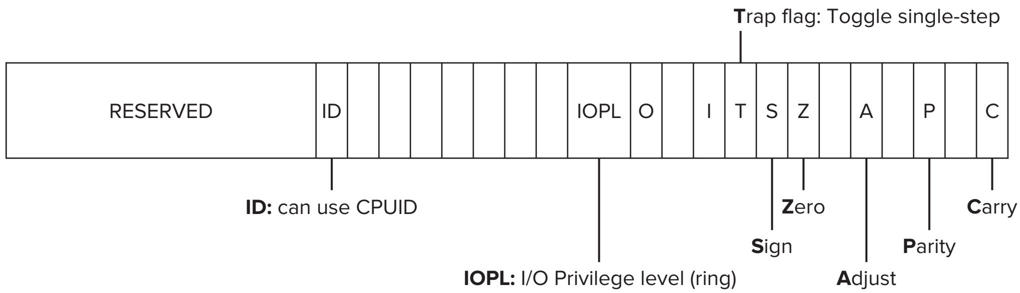


FIGURE A-1: Important flags in the EFLAGS register

The EFLAGS register can only be accessed only by means of a PUSHF (push flags) command through the stack. The machine level `ml_get_interrupts_enabled` function therefore has to resort to inline assembly, as shown in Listing A-1:

LISTING A-1: OS X's ml_get_interrupts_enabled (osfmk/i386/machine_routines.c)

```
/* Get Interrupts Enabled */
boolean_t ml_get_interrupts_enabled(void)
{
  unsigned long flags;
  __asm__ volatile("pushf; pop  %0" :  "=r" (flags));
  return (flags & EFL_IF) != 0;
}
```

The EFLAGS register can be set using POPF, but to Intel provides the STI/CLI assembly instructions for toggling the interrupt flag.

Control Registers

Intel architectures have additional Control Registers (CRs) and DebugRegisters (DRs). The latter are used by debuggers to set hardware breakpoints (that is, instruct the CPU to break on read, write, or execute access to a particular address), and are outside the scope of this book. The former, however, are particularly important. While user mode (Ring 3) has no access to them, kernel mode (Ring 0) actually relies on them for enforcing protected mode, virtual memory management, and other system tasks. The following list discusses the control registers and their usage:

- **CR0:** Miscellaneous flags controlling processor operation mode. The important ones are:
  - Bit 0 (PE) toggles real/protected mode
  - Bit 16 (WP) enables write protection on memory pages
  - Bit 31 (PG) enables paging (switches to virtual memory, and enables CR3)
- **CR1:** Unused.
- **CR2:** Address of last page fault.

- **CR3:** Used when CR0's PG bit is set. Holds the address of the page directory of the current process, i.e. a pointer to the virtual memory space of the current process. As a corollary, all threads of the same process share the same value of CR3.

  In 64-bit mode, unless otherwise stated (by the `-no_shared_cr3` boot argument), the kernel address space is mapped into all tasks. Entering and exiting kernel mode, therefore, is equivalent to switching between related threads.

- **CR4:** Miscellaneous flags controlling various extensions. Bit 5, for example, controls Physical Address Extensions.

ARM

ARM processors have traditionally had more registers than Intel available for the program's general purpose, though Intel's 64-bit has narrowed the gap. While there are technically 16 registers for general purpose (R0 through R15, as outlined in Table A-2), the last three are reserved for special functions, and the first four are used in argument passing, leaving 8 or 9 registers (depending on platform) used for the program.

TABLE A-2: Shows the Registers on a Typical ARM Processor

REGISTER	USED FOR
R0	Used as the first argument to functions, and expected to hold the function's return value on exit.
R1	Used as the second argument to functions with more than one argument, or as an additional 32-bit register to contain a 64-bit first argument. Volatile.
R2	Used as the third argument to functions with more than two arguments, or as the first 32-bits of a 64-bit second argument. Volatile.
R3	Used as the fourth argument to functions with more than three arguments, or as the second 32-bits of a 64-bit second argument. Volatile.
R4-R12/ V0-V8	General purpose. Must be saved by callee.
R7/FP	In some platforms (such as iOS), used as frame pointer (at all other times used as general purpose). Note `otool(1)` incorrectly calls R11 FP, though it is general purpose.
R9/	Reserved for special use in some platforms, such as iOS.
R13/SP	Traditionally used as the Stack Pointer.
R14/LR	Traditionally used as the Link Register, containing the return address of this function.
PC (R15)	The Instruction pointer. Unlike Intel's IP, this register may be set directly.

A special feature in ARM is *register banking*. Some registers are available in "shadow copies" when in different modes. More specifically, R13 and R14 are available in per-mode copies in all CPU modes, and R8 through R12 are available in Fast Interrupt (FIQ) mode. This makes it easy to switch CPU modes without having to explicitly save registers every time (somewhat similar to Intel's Model Specific Registers (MSRs))

Floating Point Registers

As in Intel, so in ARM — there are special registers for floating point operations. As with the Intel architecture, they are rarely used in kernel mode, but if you ever run into them, you'll recognize them from Table A-3:

TABLE A-3: ARM Floating-Point Registers

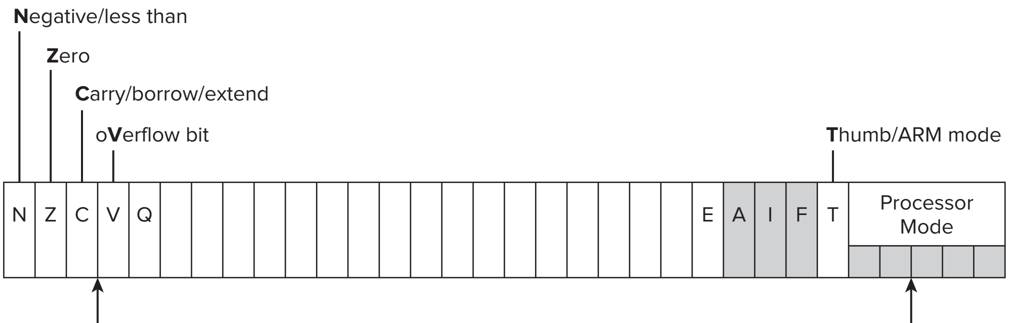
REGISTER	USAGE
S0-S15 D0-D7 Q0-Q3	Floating point registers. Two 16-bit Ss may be grouped together to form a 32-bit D, and two Ds may be grouped together to form a 64-bit Q. These can be used for floating point arguments, and are volatile.
S16-D31 D8-D15 Q4-Q7	Floating point registers, as above, but non-volatile (i.e. must be saved by callee).
S31-S63 D16-D31 Q8-Q15	Floating point registers, as above, but volatile, and only available on ARMv7 (which all modern i-Devices are).

Current Program Status Register

ARM CPUs use a special register, called the Current Program Status Register, in a way that is similar to Intel's EFLAGS. This register is a flags-only register that holds roughly the same flags as those in Intel.

Just as in the case of Intel's CPL bits (11-12) of EFLAGS, the CPSR dedicates bits to hold the current program's processor mode. As discussed in Chapter 8 (and in particular Table 8-1), the CPSR holds the processor state in its five least significant bits. These status flags are naturally not writable by code in any mode but supervisor mode, though when responding to an interrupt, fast interrupt, or trap, they are automatically set. A special case is the Thumb mode register, which is set automatically by the BX instruction (discussed later). (See Figure A-2.)

The CPSR can be read using the MRS command, and can be set using MSR, though the latter is not widely used. Instead, ARM offers a CPS command to change the processor state, and specifically set the `I` and `F` bits. The implementation of `ml_get_interrupts_enabled` in iOS therefore requires querying the CPSR (using MRS), as shown in Listing A-2:



Suffix	Values
Q/NE	0/1
CS/CC HS/HL	2/3
MI/PL	4/5
VS/VC	6/7
HI/LS	8/9
GE/GT LT/LE	A/C B/D
AL	E

Mode	Meaning
10000	_usr
10001	_fiq
10010	_irq
10011	_svc
10111	_abt
11011	_und
11111	_system

FIGURE A-2: The ARM CPSR flags

LISTING A-2: ml_get_interrupts_enabled in iOS

```
_ml_get_interrupts_enabled:
0x8007C26C    MRS  R2, CPSR          ; Read value of CPSR into R2
0x8007C270    MOV  R0, #1            ; Set R0 to be "1"
0x8007C274    BIC  R0, R0, R2,LSR#7 ; Isolate bit #8 ("I")
0x8007C278    BX              LR     ; returns R0
```

Similar to Intel, instead of having to set the interrupt flag through CPSR the specific assembly instructions of `CPSIE(nable)` and `CPSID(isable)` can be used to toggle interrupts. These instructions take an argument of `I` for normal IRQs and `F` or fast IRQs. This can be seen in the disassembly of `ml_set_interrupts_enabled`, which is left as an exercise to the interested reader.

Control Registers

Whereas Intel uses the CR registers for various process control tasks, ARM employs a *coprocessor*. This coprocessor is known as p15, and has its own registers. It is used for various low-level operations, including cache control, virtual memory, and multithreading support. Operations on the coprocessor are generally of the form of reading (MRC) or writing (MCR) to the coprocessor's registers.

Both the MRC and MCR commands follow the same general syntax:

```
MRC/MCR  p15,  Opcode, Reg,  C#1, C#2, Opcode2
```

Where:

- **p15**—This constant denotes coprocessor
- ***Opcode***—Operation to perform
- ***Reg***—Destination (MRC) or source (MCR) register
- **C##**, **C##**—Coprocessor control registers, as per Table A-4
- **Opcode 2**—Additional opcode, if required

SETTING: ABIS AND CONTEXTS

The processor executes code linearly (out-of-order execution notwithstanding). Developers, however, make use of functions and subroutines in order to improve code readability and efficacy. When the compiler emits code, it follows certain calling convention that dictate how the functions are to be called and which registers are used for passing the parameters and return values. When the compiler emits calls that interface with the operating system (namely, system call invocations), it must additionally pass system call numbers and parameters in a way that is mutually agreed upon with the operating system. Additionally, certain other conventions dictate floating-point usage, and data alignment. Collectively, all these are known as the Application Binary Interface, or ABI. Apple provides documentation for the ABIs used in both OS X[5] and iOS[6], but both documents refer to the standard architecture ABI documents by AMD (which originated the x86_64 standard) and ARM, respectively.

ABIs

Intel and ARM have different ABIs, but the principles are similar. In both, the calling conventions follow the same rough idea: Some registers are declared volatile, meaning their values are not expected to persist across a function call, whereas others are. A non-volatile register, however, is not necessarily a reserved register: Functions are expected to save non-volatile registers on entry and restore them on exit. So long as the non-volatile registers are correctly saved and restored, the caller has no idea (and really doesn't care, either) if they are used in whatever way. What follows, is that functions generally have a fixed prolog and epilog. This can be a useful anchor when trying to disassemble blocks of assembly which have no symbols.

When calling a function, the following conventions are adhered to:

- The calling function (caller) is expected to do the following:
    - Pass as many arguments as possible in the registers allocated for them
    - If there are less arguments than available registers, registers are unused
    - If there are more arguments than registers, any remaining arguments are passed on the stack

  - Save its return address, so the called function may return to its caller upon completion
  - Pass control to the called function by jumping to its address

The callee has more responsibilities than the caller:

- On entry (that is, in the prolog), the called function (callee) is expected to:
  - Save any registers it is going to use
  - If a frame pointer (Intel: RBP, ARM: R7) is used, set it
  - Save any floating point registers it may be using
  - Allocate space on the stack for local variables
- On exit, the callee is also expected to:
  - Deallocate space on the stack for local variables
  - Restore any floating point registers it may have been using
  - Restore any general purpose registers it may have been using
  - Restore the Frame Pointer, if used, and return to the return address specified by the caller

Comparing the same function call on Intel and ARM side by side shows this well.

Figure A-3 demonstrates a decompilation of `thread_call_allocate()`, with interleaved source code and implementation on both Intel and ARM. You are encouraged to use `otool(1)` or IDA to see this call, as it is exported on both platforms.

Unlike the Intel architecture, wherein the instruction pointer may only be set by a `JMP`, `CALL`, or `RET` instruction, ARM is more flexible: The PC may be set by a branch, but also by a `POP` (as in the previous example), or by a direct load (`LDR`), or even a simple move (`MOV`). Both Intel and ARM assembly opcodes are discussed in this appendix.

Context Switching

Another type of control transfer is *context switching*, the process of replacing the currently executing thread with another one. Unlike function calls, in which the caller premeditates the control transfer, this is an abrupt occurrence, which often happens unexpectedly (due to an interrupt), and which the thread is totally unaware of. It is, in effect, the same as pausing a movie, changing the channel, then — at some later point — resuming the movie.

Context switching in Mach is abstracted by the `machine_switch_context(osfmk/x86_64/Cswitch.s)` wrapper, which wraps the `Switch_Context` assembly logic. OS X's `Switch_Context`, as would be expected of an Intel architecture, saves all the registers and loads the previous state. Intel doesn't have a "save all registers" command, so this is done manually, as shown in Listing A-3 (i386 code is virtually identical, but with fewer registers).

```
thread_call_t thread_call_allocate(
                thread_call_func_t          func,
                thread_call_param_t         param0)
{
```

```
push    rbp
mov     rbp, rsp

push    r14
push    rbx
```

```
PUSH    {R4-R7,LR} ; 0xB5F0

MOV     R4, R1 ; R4 = param0
LDR     R1, =thread_call_zone

ADD     R7, SP, #0xC ; R7 = SP+12
```

```
thread_call_t call = zalloc(thread_call_zone);
```

```
mov     rbx, rsi ; rbx = param0
mov     r14, rdi ; r14 = func
mov     rdi, = thread_call_zone
call    _zalloc
; Now rax = call
```

```
MOV     R6, R0  ; R6 = func
LDR     R0, [thread_call_zone]
BL      _zalloc
; Now R0 = call
```

```
(call)->func = (call_entry_func_t)(func);
(call)->param0 = (call_entry_param_t)(param);
(call)->queue  = NULL;
```

```
; bzero (call, 60);
MOVS    R1, #0x3C ; size_t
MOV     R5, R0    ; R5 = call
MOV     R0, R5    ; void *
BLX     _bzero    ; R0 destroyed
```

```
; call->func = func
mov     [rax+18h], r14
; call->param0 = param
mov     [rax+20h], rbx
; call->queue = NULL
mov     qword ptr [rax+10h], 0
```

```
STR     R6, [R5,#0xC] ;call->func
MOVS    R1, #0        ; R1 = 0
MOVS    R0, #1        ; R0 = 1
STR     R4, [R5,#0x10] ;call->param
STR     R1, [R5,#8]    ;call->queue
; iOS has more fields than Lion..
STR     R1, [R5,#0x30]
STR     R0, [R5,#0x34]
STR     R0, [R5,#0x38]
```

```
return (call);
```

```
; rax already holds call
; restore regs in reverse order
pop     rbx
pop     r14
pop     rbp
retn
```

```
MOV     R0, R5 ; return (call);
POP            {R4-R7,PC}
```

FIGURE A-3: Comparison of thread_call_allocate code on both ARM and Intel

LISTING A-3: Switch_context on Intel x64, from osfmk/x86_64/cswitch.s

```
/*
 * thread_t Switch_context(
 *              thread_t old,                          // %rsi
 *              thread_continue_t continuation,        // %rdi
 *              thread_t new)                          // %rdx
 */
Entry(Switch_context)
        popq    %rax                            /* pop return PC */

        /* Test for a continuation and skip all state saving if so... */
        cmpq    $0, %rsi
        jne     5f
        movq    %gs:CPU_KERNEL_STACK,%rcx       /* get old kernel stack top */
        movq    %rbx,KSS_RBX(%rcx)              /* save registers */
        movq    %rbp,KSS_RBP(%rcx)
        movq    %r12,KSS_R12(%rcx)
        movq    %r13,KSS_R13(%rcx)
        movq    %r14,KSS_R14(%rcx)
        movq    %r15,KSS_R15(%rcx)
        movq    %rax,KSS_RIP(%rcx)              /* save return PC */
        movq    %rsp,KSS_RSP(%rcx)              /* save SP */
5:
        movq    %rdi,%rax                       /* return old thread */
        /* new thread in %rdx */
        movq    %rdx,%gs:CPU_ACTIVE_THREAD      /* new thread is active */
        movq    TH_KERNEL_STACK(%rdx),%rdx      /* get its kernel stack */
        lea     -IKS_SIZE(%rdx),%rcx
        add     EXT(kernel_stack_size)(%rip),%rcx /* point to stack top */

        movq    %rdx,%gs:CPU_ACTIVE_STACK       /* set current stack */
        movq    %rcx,%gs:CPU_KERNEL_STACK       /* set stack top */
        movq    KSS_RSP(%rcx),%rsp              /* switch stacks */
        movq    KSS_RBX(%rcx),%rbx              /* restore registers */
        movq    KSS_RBP(%rcx),%rbp
        movq    KSS_R12(%rcx),%r12
        movq    KSS_R13(%rcx),%r13
        movq    KSS_R14(%rcx),%r14
        movq    KSS_R15(%rcx),%r15
        jmp     *KSS_RIP(%rcx)                  /* return old thread */
```

The saved value of `RIP`, which is also the one restored, returns to `machine_switch_context()` which called this function. Because this is the very last line in `machine_switch_context`, however, control returns back to its caller, `thread_invoke()`, which either calls the continuation, or returns right after `thread_block()`.

iOS performs a context switch even more elegantly by using ARM's `STM` and `LDM` commands, which can store multiple registers with a single instruction, as shown in Listing A-4:

LISTING A-4: Context switching, ARM style

```
__Switch_context: ; (called in ARM from machine_switch_context)       _
0x8007B3A0     TEQ      R1, #0          ; is continuation specified?
```

```
0x8007D364     STRNE    R1, [R0,#0x44]  ; if yes, save to old+44
  ;;
  ;; If R1 == 0, there is no continuation - so we need to save state:
  ;;
0x8007D368     LDREQ    R3, [R0,#0x4B4] ; get TCB
0x8007D36C     ADDEQ    R3, R3, #0x10   ; get Register save area
0x8007D370     STMEQIA  R3!, {R4-LR}    ; save registers
  ;;
  ;; The following is done in any case (like the label "5" in the intel case)
  ;;
0x8007D374     LDR      R3, [R2,#0x4B4] ; get new thread TCB
0x8007D378     MCR      p15, 0, R2,c13,c0, 4
0x8007D37C     LDR      R6, [R2,#0x4C0]
0x8007D380     MRC      p15, 0, R5,c13,c0, 3
0x8007D384     AND      R5, R5, #3
0x8007D388     ORR      R6, R6, R5
0x8007D38C     MCR      p15, 0, R6,c13,c0, 3
0x8007D390     LDR      R6, [R2,#0x4C4]
0x8007D394     MCR      p15, 0, R6,c13,c0, 2
__load_context:  ; this is also called in iOS from machine_load_context
0x8007D398     ADD      R3, R3, #0x10   ; get Register save area
0x8007D39C     LDMIA    R3!, {R4-LR}    ; Load R4 through R14
0x8007D3A0     BX       LR              ; Return to loaded R14 (LR)
```

Note, that in both the OS X and iOS cases, a check is made for a continuation. If one is specified, the operation of saving the register state can be skipped altogether, allowing for a much faster thread context switch. Continuations are discussed in Chapter 11.

FLOW: OPCODES

Intel and ARM assembly are two different languages: They can be used to convey the same ideas, though with totally different syntax and words. The two assembly languages are also very rich, with hundreds of mnemonics. Just like human languages, however, which can be colloquially mastered with a subset of the full vocabulary, so can assembly be understood with relatively few mnemonics. These are listed in Table A-5.

TABLE A-5: Assembly Mnemonics

INSTRUCTION	INTEL MNEMONIC	ARM MNEMONIC
Move value to/from registers	`MOV`	`MOV` MVN: `move negative` `LDR/STR: Load/Store Register` `LDMIA/STMIA reg!, {register-list}` `Load/Store Multiple (Registers) and increment after`
Basic arithmetic	`ADD` `SUB` `MUL` `DIV`	`ADD` `SUB` `MUL/MULA` `SDIV/UDIV`

continues

TABLE A-5 *(continued)*

INSTRUCTION	INTEL MNEMONIC	ARM MNEMONIC
Logical test on value in a register	`TEST`	`TST` `MOVS`
No-operation	`NOP`	`MOV R0, R0`
Logical Operations	`AND` `OR` `XOR`	`AND` `ORR` `EOR` `BIC` (bitwise-complement)
Jump	`JMP/Jxx`	`B` (with standard conditionals, see "Conditional Execution" section below)
Call a function	`CALL address`	`BL address/register` `BLX address/register – change ARM/Thumb`
Return from a function	`RET`	`BX LR` (common) (Can also modify PC directly)
Stack operations	`PUSH register` `POP register`	`PUSH {register-list}` `POP {register-list}`
Simulated interrupt/system call	`INT`	`SWI/SVC`
Breakpoint	`INT $3`	`BKPT num`

A great "cheat sheet" for Intel Assembly can be found in a work by Ange Albertini[7], and ARM maintains a quick reference card as well[8].

ARM ASSEMBLY ENHANCEMENTS

ARM assembly is somewhat different from other assembly languages, in that it has specific features no other language has. Instructions may be suffixed with logical conditions, or specified with bit-shift operations. These features are discussed next.

Conditional Execution

ARM processors have a nifty feature: A conditional suffix may be appended to every instruction. This conditional tests the result of the last comparison or logical comparison operation, and only executes the instruction if it satisfies that result. Otherwise, the instruction in question effectively becomes a NOP command. This is more elegant and cache-friendly than simply jumping over a set of instructions. The suffixes are shown in Table A-6:

TABLE A-6: Instruction Suffixes on ARM for Conditional Execution

SUFFIX	MEANING
EQ/NE	Equal or Not-Equal
CS/CC HS/HL	Carry set or clear Unsigned Higher-same or lower
MI/PL	Minus (negative) or Zero-Positive
VS/VC	Overflow or not overflow
HI/LS	Signed higher or lower
GE/GT/LT/LE	>=/>/</<=
AL	Always (not specified, as it is default)

If you look back at Figure A-2, you will see how the suffix maps to the flags in the CPSR.

Built-in Bit Shifting

Another useful (though somewhat confusing) feature of ARM processors is the ability to specify bit-shifts in the instruction. The processor has a barrel shifter, which enables it to shift left (i.e. multiply by powers of 2) or right (divide by powers of 2). The right shifts, in particular, may be one of three types:

- **Logical:** A "0" is pushed into the most significant (leftmost) position, and pushes all the bits right. The least significant bit is lost.
- **Arithmetic:** The current bit value of the most significant bit is used to push it along with all other bits right. The least significant bit is lost.
- **Rotation:** As arithmetic, with the least significant bit used to push the most significant bit.

An example of the logical shift right could be seen in Listing A-2, which demonstrated getting the interrupt status. To isolate bit `#8` of the CPSR (the `I` bit, which holds the interrupt state), the command `BIC R0, R0, R2,LSR#7` is used to shift R2 (holding the value of CPSR) right 7 bits (making the eighth bit the first bit), then take a bitwise complement of it, and performs a bitwise AND with the value of 0x01 (which preserves the first bit) back into R0 (which is returned to the caller).

Thumb mode

ARM processors have more than one mode of operation. In the normal, 32-bit mode, they execute the default instruction set, known as ARM. They can, however, be instructed to dynamically change the instruction set to a more compact, 16-bit mode known as Thumb mode. This means that, when dumping an ARM binary, the assembly may be read in one of two ways, with only one of them being the "correct" mode. This dual mode often confused `otool(1)`, which is why it can be forced to dump ARM binaries in Thumb using the `-B` switch. Even powerful disassemblers, most notably IDA, sometimes get the mode wrong.

The processor itself "knows" which mode is required because its branch instruction, B can contain the X directive, specifying a mode switch. The encoding of the desired mode is in the address itself: The least-significant bit of the address encodes `1` for thumb mode, or `0` for ARM. This encoding is possible since bit is unused anyway: ARM instructions must be aligned on a four byte boundary, and thumb instructions must be aligned on a two byte boundary, leaving the bit unused in either case.

So long as you know how the processor got to a particular code section, telling the two modes apart is simple. But if you are dumping some random text, there is no way to disambiguate ARM mode from Thumb mode without trying both. Usually, trying the incorrect mode (ARM when it's actually Thumb, or vice versa) yields nonsensical or just plain illegal instructions.

GENERAL CONCEPTS

User mode programmers enjoy many benefits they often take for granted: multithreading, virtual memory, and synchronization objects, among others. The kernel, however, is the entity responsible for providing these, and falls back on the hardware whenever possible. This section discusses hardware support mechanisms the kernel utilizes for various tasks.

Multithreading

Both ARM and Intel processors support threading at the processor level. This is, in fact, why modern operating systems don't schedule processes anymore, but threads. The process as we know it, a vestige of UNIX terminology, remains only at the administrative level, used for accounting, and resource containment.

Intel

Intel-based operating systems use a segment register to hold the thread control block. OS X uses GS. This is shown in Listing A-4.

LISTING A-4: The current_task /current_thread machine-specific implementation in Lion

```
_current_task:
ffffff8000235f60    pushq   %rbp
ffffff8000235f61    movq    %rsp,%rbp
ffffff8000235f64    movq    %gs:0x00000008,%rax   ; get the current thread
ffffff8000235f6d    movq    0x00000348(%rax),%rax ; return thread->task (offset 0x348)
ffffff8000235f74    popq    %rbp
ffffff8000235f75    ret

_current_thread:
ffffff80002bc1c0    pushq   %rbp
ffffff80002bc1c1    movq    %rsp,%rbp
ffffff80002bc1c4    movq    %gs:0x00000008,%rax
ffffff80002bc1cd    popq    %rbp
ffffff80002bc1ce    ret
ffffff80002bc1cf    nop
```

ARM

On ARM (from an iOS 5.0.0 kernel), a call is made to `cr13`, the "thread and process ID register," as documented in the ARM architecture manuals. This is shown in Listing A-5:

LISTING A-5: The current_task and current_thread machine-specific implementation in iOS, from an iOS 5.0.0 iPod 4G (Apple A4, Arm Cortex A8)

```
_current_task:
80027a18    ee1d0f90    mrc     15, 0, r0, cr13, cr0, {4} ; Get the current thread
80027a1c    f8d004cc    ldr.w   r0, [r0, #1228]           ; 0x4CC (note different offset)
80027a20    4770        bx      lr                        ; return
_current_thread:
8007bc00    ee1d0f90    mrc     15, 0, r0, cr13, cr0, {4} ; Get the current thread
8007bc04    e12fff1e    bx      lr                        ; return
```

It is fairly common to find the ARM instruction sequences also inlined in various other thread and task functions. This is not necessarily for obfuscation, as much as it is a likely consequence of compiler optimizations.

Locking and Atomicity

A prerequisite of concurrency in modern operating systems is the ability to provide a safe locking mechanism, by means of which access to shared resources can be synchronized. This mechanism often relies on hardware support, and therefore is implemented differently in ARM and Intel architectures. Furthermore, often, even the same architecture may choose different implementations, based on UP or SMP availability.

A good example of this can be found in the implementation Mach's low level `hw_lock_lock()` function. From the kernel's perspective, this function always delivers the same functionality: a fast spinlock (as discussed in Chapter 10). The underlying implementation, however, uses different hardware features in Intel or in ARM.

Intel

Listing A-7 shows the various implementations of `_hw_lock_lock` on OS X 64-bit (Listing A-7) and iOS (Listing A-8 and Listing A-9). The i386 implementation is largely the same as the 64-bit one, and is left as an exercise for the reader.

LISTING A-7: hw_lock_lock from a 10.7.3 kernel, on an x86_64

```
_hw_lock_lock:
ffffff80002b3300    movq    %gs:0x00000008,%rcx
ffffff80002b3309    incl    %gs:0x00000010
  ;; Attempt lock here
ffffff80002b3311    movq    (%rdi),%rax
ffffff80002b3314    testq   %rax,%rax
ffffff80002b3317    jne     0xffffff80002b3326
```

continues

LISTING A-7 *(continued)*

```
   ;; lock is free - attempt to lock, but double check, since another thread can beat us
to it
ffffff80002b3319    lock/cmpxchgq   %rcx,(%rdi)
ffffff80002b331e    jne     0xffffff80002b3326  ;; double check failed - go spin
ffffff80002b3320    movl    $0x00000001,%eax    ;; Successful - return 1 to caller
ffffff80002b3325    ret                         ;; return
   ; Spinning - pause for a cycle, then jmp right back to the lock attempt
ffffff80002b3326    pause
ffffff80002b3328    jmp     0xffffff80002b3311
```

ARM

On a single core ARM processor (i.e. pre-A5 processors), `hw_lock_lock` doesn't need to spin. In fact, if it *did* spin a deadlock could result. The implementation is therefore straightforward:

LISTING A-8: hw_lock_lock from iOS 5.0, on an ARM single core (iPod touch 4G)

```
0x800757F0 _hw_lock_lock   MRC    p15, 0, R12,c13,c0, 4 ; Load current thread
0x800757F4                 LDR    R2, [R12,#0x4BC]      ; Load value from thread_t
0x800757F8                 ADD    R2, R2, #1            ; Increment value
0x800757FC                 STR    R2, [R12,#0x4BC]      ; Put value back into thread_t
0x80075800                 LDR    R3, [R0]              ; Load lock value into R3
0x80075804                 ORR    R1, R3, #1            ; Light lock bit
                           ;;  sanity check
0x80075808                 TST    R3, #1                ; Test if indeed 1
0x8007580C                 STREQ  R1, [R0]              ; Store back into lock, if 1
0x80075810                 BXEQ   LR                    ; And return, if 1
                           ;;  If we get here, panic!
0x80075814                 MOV    R1, R0                ; Move lock address to R1
0x80075818                 ADR    R0, "hw_lock_lock(): lock (0x%08X)\n"
0x8007581C                 LDR    PC, =(_panic+1)       ; Jump to panic, in Thumb mode
```

On the A5, which is a dual-core (hence, SMP) architecture, the code is more complex, with the LDR and STR replaced by their EX (exclusive) counterparts, and the addition of a slow path. Further, a Data Memory Barrier (DMB) instruction is executed prior to return:

LISTING A-9: hw_lock_lock from iOS 5.0, on an ARM dual core (iPhone 4S)

```
_hw_lock_lock:
0x80075630                 MRC              p15, 0, R12,c13,c0, 4
0x80075634LDR             R2, [R12,#0x4BC] ; Load value from thread_t
0x80075638                 ADD              R2, R2, #1       ; Increment
0x8007563C                 STR              R2, [R12,#0x4BC] ; Store it
0x80075640 _retry          LDREX            R3, [R0]
0x80075644                 TST              R3, #1
0x80075648                 ORREQ            R3, R3, #1
0x8007564C                 STREXEQ          R1, R3, [R0] ;  Store and exchange
0x80075650                 BNE              0x80075664   ; _slow_path
0x80075654                 CMP              R1, #0
```

```
0x80075658                  BNE            _retry
0x8007565C                  DMB            #0xB         ; Data Memory Barrier
0x80075660                  BX             LR
0x80075660 _slow_path    ; ...
```

A similar functionality closely related to locking is that of *atomic operations*. An atomic operation is an operation in which atomicity (i.e. non-interruptibility) is guaranteed. The `OSAddAtomic64(b,&a)` is an atomic operation of *a* = *a* + *b*, where *a* and *b* are signed Integer 64 types, and a is passed by reference. Atomic operations often serve as the underlying mechanism to enable locks (as locks must be accessed in a guaranteed atomic manner), and can often be used instead (when the object guarded is machine-word sized).

On OS X, either disassemble (`otool -tV`) the kernel image, or look at the XNU source code. If you choose to disassemble, make sure to select the i386 image by passing `-arch i386` to `otool(1)`, as shown in Listing A-10:

LISTING A-10: The implementation of _OSAddAtomic64 on Intel, 32-bit

```
_OSAddAtomic64:
        pushl           %edi
        pushl           %ebx

        movl            12+8(%esp), %edi   ; ptr
        movl            0(%edi), %eax      ; load low 32-bits of *ptr
        movl            4(%edi), %edx      ; load high 32-bits of *ptr
1:
        movl            %eax, %ebx
        movl            %edx, %ecx         ; ebx:ecx := *ptr
        addl            4+8(%esp), %ebx
        adcl            8+8(%esp), %ecx    ; ebx:ecx := *ptr + theAmount
        lock
        cmpxchg8b       0(%edi)            ; CAS (eax:edx, ebx:ecx implicit)
        jnz             1b                 ; - failure: eax:edx re-loaded, retry
                                           ; - success: old value in eax:edx
        popl            %ebx
        popl            %edi
        ret
```

On OS X in 64-bit mode, the atomic operation is natively supported by the architecture, making for even simpler code, as shown in Listing A-11:

LISTING A-11: The implementation of OSAddAtomic* on Intel, x86_64

```
_OSAddAtomic64:
ffffff800062916b        lock/xaddq      %rdi,(%rsi)
ffffff8000629170        movq    %rdi,%rax
ffffff8000629173        ret
_OSAddAtomic:
ffffff8000629174        lock/xaddl      %edi,(%rsi)
ffffff8000629178        movl    %edi,%eax
```

Kernel mode has no monopoly over atomic operations: Atomic functions are available in user mode, although with the name ordering reversed (q.v. `OSAtomicAdd32(3)` and friends). The implementation is the same as the kernel's, though through a stub (i.e. LibSystem's `OSAtomicAdd32`, for example, loads the address of `__atomic_add32` which has the i386 or x86_64 code). The actual code resides either in the commpage (in Snow Leopard, as discussed in Chapter 4), or is located by LibSystem's `find_platform_function`.

In iOS, you can disassemble (`otool -tV`) the kernel image, and look for the `_OSAddAtomic64` symbol which is still exported (using `more(1)/less(1)`, type **`"/^_OSAddAtomic64"`**). You should see something like Listing A-12:

LISTING A-12: The implementation of _OSAddAtomic on ARM (iOS 5.1)

```
_OSAddAtomic64:

;  ARM is a 32-bit processor, so to pass around 64-bits it groups registers
;  together. r0,r1,r2,r3 - usually used for four 32-bit arguments, can pass
; instead up to two 64-bit ones. Thus:
; @param: r0-r1: amount, as 64-bit value spanning both registers
; @param: r2:    address of 64-bit value in memory

80077f30  e92d4330     push    {r4, r5, r8, r9, lr} ; save non volatile
80077f34  e1b24f9f     ldrexd  r4, [r2]             ; atomic load: *r2 to r4-r5
80077f38  e0948000     adds    r8, r4, r0           ; add-signed low bits
80077f3c  e0a59001     adc     r9, r5, r1           ; add-carry high bits
80077f40  e1a23f98     strexd  r3, r8, [r2]         ; atomic store r8-r9 -> *r2
80077f44  e3530000     cmp     r3, #0  @ 0x0        ; test if failed..
80077f48  1affff9      bne     0x80077f34           ; if indeed failed, retry
80077f4c  e1a00004     mov     r0, r4               ; else return: low  in r0
80077f50  e1a01005     mov     r1, r5               ;           .. high in r1
80077f54  e8bd8330     pop     {r4, r5, r8, r9, pc} ; restore regs, return
```

Note that "atomic" does not necessarily mean "single cycle." It just means that the CPU guarantees uninterrupted access. There are many more examples of this. If you want, take a peek at `task_reference()` (which is defined over `task_reference_internal` (`osfmk/kern/task.h`), itself a macro over `hw_atomic_add`). The Intel and ARM implementations closely resemble the preceding example.

Barriers

Modern CPUs can execute instructions out of order to optimize utilization of their internal components (such as the ALU, FPU, and load/store units). The CPU has liberty in deciding the actual order, and usually this goes unnoticed by both the developer and the compiler generating the code. In some cases, however, out-of-order execution may introduce bugs into the program. In these cases, *barrier* instructions can be used to ensure all access completes by a certain point in the program's execution.

Intel provides Load (LFENCE), Store (SFENCE), and both (MFENCE) barrier instructions. ARM provides three types of barrier instructions: Data synchronization (DSB), Data Memory (DMB), and Instruction Synchronization (ISB).

Virtual Memory

Both Intel and ARM chips support virtual memory at the processor level, with the low-level functionality of virtual to physical translation performed by a dedicated Memory Management Unit (MMU). This allows the CPU to switch into virtual memory mode fairly early during the operating system boot, and from thereon use virtual addresses instead of physical ones.

Intel

Intel architectures enable protected mode and paging through CR0 (bits 0 and 31, respectively). From that moment on, the CPU shifts to virtual addresses, with CR3 used as the master page table.

The page table is actually a multi-level table: Depending on architecture (32-bit, PAE, or 64-bit), the page table is of varying depth (2, 3, or 4, respectively). The kernel sets up the page tables in a format that the MMU can understand, and virtual address resolution is conducted by the MMU. In case of a page fault, the MMU reports back to the CPU the page fault address in CR2.

In Intel 32-bit architectures each level is on a physical page with 1024 entries 3 (32-bit pointer) = 4k. Physical Address Extensions (PAE) extend this to work with 64-bit pointers, reducing the number of entries to 512 (to preserve 512 entries 3 (64-bit pointer) = 4k), resulting in the addition of the third level (a small 2-bit table, with only four entries). This scheme is further extended in 64-bit to four levels, each with a 9-bit index, allowing for a maximum addressable space of 48 bits. PAE and 64-bit can also opt to use the penultimate table for pages, which allows for 2 MB ("super") pages.

Using a multi-level table makes the table more space-efficient (at the cost of multiple lookups) and facilitates sharing, particularly of kernel memory. In the original 32-bit OS X, the kernel used its own virtual memory space (and hence, its own value of CR3). As of OS X 64-bit this is, by default, no longer true, with the kernel mapping its memory into the high region of every address space, unless explicitly instructed to not do so with the `–no_shared_cr3 boot` argument.

ARM

ARM supports a two level page table. Unlike Intel, in 32-bit mode the first level divides the address space into 1 MB sections (as opposed to Intel's 4 MB), with 4096 page table entries, allowing for 256 entries of 4 K pages, or 1 entry of a 1 MB superpage. (This is, of course a greatly simplified nutshell view: ARM processors also allow fine and course page granularity for smaller or larger page sizes).

Virtual memory is controlled on ARM (like just about everything else) through coprocessor 15, as the example in Listing A-13 shows. The MMU control bits can be used to enable/disable the MMU (least significant bit), data and instruction caches, and various other settings. Most important of those are memory *domains* and *access permissions*

LISTING A-13: Controlling the MMU

```
; Near textbook example of reading from cp15. In this case, read MMU value
; (q.v. ARM manual, 3-46)
_get_mmu_control:
_0x8007BDF0      MRC   p15, 0, R0,c1,c0, 0 ; Read CP15, c1,c0, opcode 0 into R0
_0x8007BDF4      BX    LR                  ; Returns R0
_set_mmu_control:
_0x8007BDF8      MCR  p15, 0, R0,c1,c0, 0  ; Write CP15, c1, c0, opcode 0 from R0
_0x8007BDFC      ISB  SY                   ; Instruction barrier
_0x8007BE00      BX   LR                   ; Returns R0
```

The c2 register holds the Translation Table Base (TTB), which is akin to CR3. ARM also supports a Translation Lookaside Buffer (TLB) for faster lookups, which is controlled through c8 (usually with c7). The TLB lines can be locked, which permits them to persist when the TLB is flushed (as a result of a context switch). This is accomplished by modifying p15's c10 register.

REFERENCES

1. Intel Architecture manuals, `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`
2. AMD64 manuals, `http://developer.amd.com/documentation/guides/Pages/default.aspx`
3. ARM Architecture Manuals, `http://infocenter.arm.com/help/topic/com.arm.doc.set.architecture/index.html`
4. Sloss, Symes and Wright, *ARM System Developer's Guide*. Morgan Kaufmann; 2004
5. "Mac OS X ABI Function Call Guide," `http://developer.apple.com/library/mac/#documentation/DeveloperTools/Conceptual/LowLevelABI/`
6. "Iphone OS ABI Reference," `http://developer.apple.com/library/ios/#documentation/Xcode/Conceptual/iPhoneOSABIReference/`
7. x86/x64 Opcodes infographics, `https://code.google.com/p/corkami/`
8. ARM and Thumb-2 Instruction quick reference card, `http://infocenter.arm.com/help/topic/com.arm.doc.qrc0001l/QRC0001_UAL.pdf`

INDEX

Symbols

\ (backslash), NVRAM variables, 199
?? (question mark-double), dyld, 114

A

-a, 126
aapl, 193
ABIs, 779–780
abnormal_exit_notify, 438
aborts, 270
ABRT(), 534
Absinthe, 173
ABT, 268
Accelerate, 35
accept(), 240
Access Control Lists (ACLs), 578, 608
accessory_device_arbitrator, 244
Accounts, 35
accountsd*, 244
ACLs. *See* Access Control Lists
-acm, 578
acpi_*, 329
act_set_astbsd(), 325
act_set_bsdast(), 536
addDisk, 576
AdditionalEssentials.pkg, 216
AdditionalSystemVoices.pkg, 216
Address Family (AF), 650
Address Space Layout Randomization (ASLR), 12, 122, 131–132, 173, 548–549
AddressBook, 35
ADD_TO_ZONE, 469
Advanced PIC (APIC), 270
Adv-cmds, 14
AEDebug*, 73
AEServer, 73
AF. *See* Address Family
AF_, 650
afc, 246
affinity, CPU threads, 415
AFFINITY_POLICY, 421
affinity_tag, 421
AF_INET, 677
AFP. *See* Apple Filing Protocol
AGL, 35
Air Drop, 8
alarm_expire_timer, 380
alarm_lock, 380
allmemory(1), 160, 161
alloca(), 138–139
AllocatePages, 189
AllocatePool, 189
allocation file, B-Tree, 642
alloc_size, 469
_AllowedClients, 257
al_port, 380
alternate data streams, 611
AMFI. *See* AppleMobileFileIntegrity
amfi_*, 563
Amfid, 244
Animation, 201
APIC. *See* Advanced PIC
APM. *See* Apple Partitioning Scheme
App Store, 11, 25–26
AppKit, 35
AppKitScripting, 35
apple argument, 130
Apple Filing Protocol (AFP), 582, 651, 652
Apple Partitioning Scheme (APM), 570–572
Apple policy modules, BSD, 560–563
Apple Protect pager, 491–493
Apple TV, 11–12
AppleACPIPlatform.Kext, 329
APPLE_BOOT_GUID, 193
AppleEvents, 72–79
AppleFSCompressionTypeZlib.kext, 612
AppleIDAuthAgent(), 242
AppleMobileFileIntegrity (AMFI), 89–90, 331, 562–563
AppleOnBoardSerialBSDClient, 656
AppleProfile*, 155
appleprofilepolicyd, 242
apple_protect_pager_setup(), 493
AppleScript*, 35, 72–79
AppleShareClientCore, 35
AppleTalk, 35
APPLE_VENDOR_NVRAM_GUID, 195
/Applicants, 23
/Application, 25

applications
  bundles, 24
  containers, Lion, 84–97
  debugging
    crashes, 170–176
    hangs, 173–174
    sampling, 173–174
  defaults, 30–32
  Java, 44–45
  NeXTSTEP, 4, 24
  OS X, 24–32
Application Frameworks layer, 15
Application Services, 18
Applications, 26
/Applications, 23
ApplicationServices, 35
Application.System, 257
Application.User, 257
<app>.pkg, 216
Apsd, 244
ApTicket, 213
Aqua, 17–18
arbiter, kernel, 262
arch(1), 100, 101–102
architecture
  ARM, 519
  Intel, Mach physical memory management, 465–467
  kernel, 261–287
    XNU, 302–305
  Mach
    Intel, physical memory management, 465–467
    VM, 447–462
  modular, 712–713
  OS X, 518
  PPC, 518–519
arg_ptr, 304
arg_string, 304
arguments, XNU boots, 329–331
argv[], 326–327
ARM, 12, 14
  architecture, 519
  ASLR, 549
  assemblies, 784–786
  atomicity, 788–790
  Darwin, 5
  EFLAGS, 296
  exception vector, 268
  Intel trap handlers, 275–278
  interrupts, 296
  iOS, 5, 261
  kernel, 267–268
  locking, 788–790
  machine_init, 316
  multithreading, 787
  registers, 776–779
  SWI, 280
  VM, 447, 791
  voluntary user/kernel transition, 280–282
__arm__, 12
ARM_ARCH, 12
arm_init, 311
ARM_THREAD_STATE, 109
arm_vm_init(), 311
array, 255
AS, 302
asctl(1), 84
A_SETCOND, 557
AsianLanguagesSupport.pkg, 216
ASL, 70–71
-asl_in 1, 70
aslmanager, 242
ASLR. *See* Address Space Layout Randomization
assemblies
  ARM, 784–786
  mnemonics, 783–784
assert_wait, 406, 414
assert_wait_deadline, 430
Assetsd, 244
AssetsLibrary, 35
AST. *See* asynchronous system trap
AST_*, 406, 424, 426, 430
ASTs. *See* Asynchronous Software Traps
ast_taken(), 425–426
asynchronous interrupts, 431
asynchronous kernel, 268
Asynchronous Software Traps (ASTs), 275, 423–427
asynchronous system trap (AST), 325
Atc, 244
atd, 231
atomicity
  ARM, 788–790
  Intel, 787–788
atrun, 231
Attribute B-Tree, 640–641
Audio*, 35
audit(), 61
audit_*, 60–61, 352
AUDIT_ARG, 557
auditctl(), 61
auditing, OS X, 59–62, 556–558
auditon(), 61, 557
AUDIT_SYSCALL_*, 557
authentication, 80
auto-boot, 212
autofsd, 587
Automator, 36
automount, 587
autorun, 237
AVFoundation, 35

B

-b, XNU boot argument, 330
Background Color, 199
BACKGROUND_APPLICATION, 423
BACKGROUND_POLICY, 421
backing store, 452, 497

BackupAgent, 173
Backupd, 242
bad_info, 395
Baker, 11
barriers, 256, 790–791
BaseSystemBinaries.pkg, 216
BaseSystem.dmg, 214, 215
BaseSystemResources.pkg, 216
Basic Input Output System (BIOS), 183–185
Basic Security Module (BSM), 59
BBTicket, 213
BeepGen, 191
Berkeley Packet Filter (BPF), 701–705
BigBear, 11
/bin, 22
binaries
  BSD process creation, 516–522
  EFI, 187
  ELF, 15–16
  Mach-O, 522–525
  portability, 46, 502
  __stubs, 115
  universal
    executables, 98
    file(1), 99
    kernel, 100
    Mach-O, 102–105
    OS X, 99
    processes, 99–111
    Snow Leopard, 99
    Tiger, 6
  widgets, 47
/bin/csh, 21
binding, Mach, 415
binfmt, 516
/bin/ksh, 21
/bin/sh, 20
/bin/tcsh, 21
binutils, 102
/bin/zsh, 21
/bin/zsh -i, 241
BIOC*, 702, 705
BIOS. *See* Basic Input Output System
bit shifting, 785
Blazakis, Dionysus, 561
bless(1), 204–206, 215
bless(8), 204–206
block fragmentation, 624
blockCount, 639
BLOCK_IO_PROTOCOL, 190
/bn/bash, 20
Bom, 217, 613
bond, 678
Bonjour, 6
bool, 254
boot, 183–225
  disk image files, 590–591
  EFI, 185–210
  iBoot, 210–214
  installation images, 214–225
  Mach zones, 470–471
  traditional, 183–185
  XNU
    arguments, 329–331
    kernel, 299–340
Boot Camp, 204
boot loader, 184
Boot Logo*, 199
Boot Services, 188–191
boot-args, 193
  launchd, 228
  nvram, 329
boot_args
  dTrace, 202–203
  Lion, 201–202
  Revision, 202
  Version, 202
boot-command, 212
boot.efi, 187, 195, 204
  BootServices, 201
  EFI GUIDs, 192–193
  OS X, 194–210
boothowto, 326
boot-image*, 193
BootMGR, 184
bootsArgs, 304
BootServices, 201
BOOT_SERVICES_TABLE, 188
boot-signature, 193
bootstrap server, 234–235
bootstrap_cmds, 300
bootstrap_server, 235
Bourne Again shell, 20
Bourne shell, 20
BPF. *See* Berkeley Packet Filter
bpfattach(), 702
BPF_WORDALIGN, 705
bplist, 26
bridge, 678
BSD, 22, 45, 501–536
  advanced aspects, 519–563
  Apple policy modules, 560–563
  ASLR, 548–549
  cache, 545
  disk image files, 589
  EFI, 203
  heirlooms, 55–65
  implementing, 503
  initialization, 318
  I/O Kit, 737, 769–771
  kqueues, 555–556
  ledgers, 398
  MAC, 318, 558–560
  Mach, 343, 501, 510–512
    tasks, 395
  malloc(), 541–544
  _MALLOC, 479
  mcache, 545

memory
management, 539–549
pressure, 545
mincore(2), 456
msync(2), 454, 458
network stack, 649
OS X, 501
packet filtering, 693, 697
POSIX, 501, 503
system calls, 284–287
processes, 504–508
control and tracing, 525–529
creating, 512–525
groups, 507–508
lists, 507–508
software, 535
structs, 504–507
suspension and resumption, 529
signals, 529–536
handling by victims, 536
hardware, 534
slab allocators, 545
sysctl(8), 552–555
system calls, 47–48
threads, 508–512
objects, 508–510
UNIX, 501–502
work queues, 550–552
XNU, 49–50, 501, 504
zones, 541–544
bsd, 307
BSD(4), 167
bsd_ast(), 536
bsd_info, 510
bsd_init(), 318, 320–325, 326, 544, 673
bsdinit_task(), 227, 325–328, 530
bsd/kern_descrip.c, 601–602
bsd/kern/kern_descrip.c, 603–604
bsd/kern/mach_loader.c, 522–523
bsd/kern/makesyscalls.master, 285–286
bsd/kern/uipc_domain.c, 673
bsd/net/if_var.h, 680–681
bsd/net/kpi_protocol.h, 677
-bsd_out, 70, 71
BSD.pkg, 216
bsd/sys/file_internal, 602–603
bsd/sys/mount.h, 591–592
bsd/sys/mount_internal.h, 592–593
bsd/sys/protosw.h, 672–673
bsd/sys/sysent, 285
bsd/sys/user.h, 508–510
bsd/sys/vnode_if.h, 597
bsd_utasbootstrap(), 325
bsd/uxkern/ux_exception.c, 529–533
BSM. *See* Basic Security Module
bsm/security, 307
bstree, 253
BTNodeDescriptor, 625
B-Tree
allocation file, 642
Attribute, 640–641
catalog, 633–640
deletions, 636–637
forks, 639–640
hard links, 639
insertions, 636
lookups, 634–636
permissions, 637–639
soft links, 639
components, 630–645
definition of, 625
extent overflow, 640
header node, 627–629
HFS+, 624–645
journaling, 642–643
volume header, 631–632
hot files, 641–642
insertions, 624
catalog, 636
nodes, 625–627
random access, 624
search, 624, 629–630
updates, 624
buffer overflow, 131
bundle, 248
bundles
applications, 24
Finder, 25
frameworks, 32–34
Info.plist, 26
NeXTSTEP, 4, 24
OS X, 24
Quicklook, 18
byteordering, 100

C

-c, dtruss, 151
C++, I/O Kit, 737, 740–741
C++, 302
cache, 23, 121, 545
shared library, 121
Unified Buffer Cache, 484, 488, 596
Calaccessd, 244
CalendarStore, 36
CALL, 279
call psignal(), 535
call_continuation(), 420
callnum, 156–157
calloutart, 508
canblock, 469
cansignal(), 535
can_update_priority(), 430
Carbon, 34, 122
Carbon, 36
CARenderServerSBUserNotificationUIKit.statusbarserverbulletinboard.*.chatkit, 245

Cascading Style Sheet (CSS), 45
widgets, 47
case sensitivity, 21, 619
cat(1), 88–89
catalog, B-Tree, 633–640
deletions, 636–637
forks, 639–640
hard links, 639
insertions, 636
lookups, 634–636
permissions, 637–639
soft links, 639
Catalog Node ID (CNID), 633–634, 635
catch_mach_exception_raise, 533
CC, 302
CCALL, 274
CD-Audio File System (CDDAFS), 581
CD-ROM File System (CDFS/ISO-9660), 582
CFBundle*, 27, 248, 257, 718
C++filt, 300
CG(11), 167
CGXServer. *See* Core Graphics X Server
CheckEvent, 189
CheckHibernate, 198
checksum, 644
Cheetah, 5–6
chfn(1), 67
child_thread, 514
chmod(1), 578
chmod +x, 98
choose_processor(), 429
choose_thread(), 429
chown(2), 48
C/H/S. *See* Cylinder/Head/Sector
chsh(1), 67
CHUD. *See* Computer Hardware Understanding and Development
chud, 307
chud.chum, 242
c_init, 379
Clock, 352
clock, 378–380
clock_alarm*, 378, 380
clock_get_*, 378
clock_init(), 379
clock_oldinit(), 379
clock_priv, 352
clock_reply, 352
clock_service_create(), 379
cloneproc(), 325, 516
close(), 127, 512
CloseEvent, 189
CloseProtocol, 189
CNID. *See* Catalog Node ID
Cocoa, 34, 122, 145, 254
Cocoa, 36
code injection, 131
code signing, 80–81, 712–713
Code Signing in Depth, 110
CodeDirectory, 717
CodeRequirements, 717
CodeResources, 29–32, 717
codesign(1), 80, 86–87, 110
CodeSignatures, 717
Collaboration, 36
com.apple.audited.plist, 59
com.apple.blued.plist, 237
com.apple.Boot.plist, 199
com.apple.decmpfs, 612, 613
com.apple.dock.extra, 247
com.apple.iokit.matching, 237
com.apple.kpi*, 714
com.apple.syslogd.plist, 233–234
com.apple.WindowServer.plist, 235–236
Comex, 11, 12
commpage, 318
compartmentalization. *See* sandboxing
compression, 7, 612–617
compute_averages, 411
compute_priority(), 429
Computer Hardware Understanding and Development (CHUD), 154–155, 373
conditional execution, 784–785
conf, 303, 307
config, 307
CONFIG_AUDIT, 305
CONFIG_CODE_DECRYPTION, 493
configd, 242, 411
-configd(8), 67
CONFIG_DEBUG, 308
CONFIG_DTRACE, 305
CONFIG_EMBEDDED, 305, 421, 548
CONFIG_FREEZE, 494, 547
CONFIG_MACF, 305, 318
CONFIG_NO_KPRINTF_STRINGS, 305
CONFIG_NO_PRINTF_STRINGS, 305
CONFIG_SCHED_*, 305, 428
CONFIG_SOCKETS, 649
CONFIG_ZLEAKS, 468
connect(2), 682
connection, 254
consider_buffer_cache_collect(), 497
consider_machine_collect(), 497
consider_zone_gc(), 471–473, 497
console, 307
console protocols, 189–190
CONT, 94
Contents/, 26
Contents/Frameworks, 33
context switching, 780–783
continuations, 416–418
Control Registers (CRs), 266–267, 775–776, 778–779
CONTROL_APPLICATION, 423
cooperative multitasking, 420
coprocessors, 778
Core*, 36, 494
Core Animation, 7
Core Audio, 7
Core Data, 7
core dumps, 170–171

Core Frameworks layer, 15
Core Graphics X Server (CGXServer), 18
Core Image, 7
Core Storage, 8, 200, 204
Core Utilities, 14
Core Video, 7
/Cores, 23
CoreServices, 36, 75–76, 247
coreservicesd, 75
CoreStorage, 191, 575–577
CORESTORAGE(10), 167
CoreTelephony, 36
CoreText, 36
CoreVideo, 36
CoreWifi, 36
coreWLAN, 36
correctness, 265
corservices.appleid.authentication.coreservices.appleid.passwordcheck, 242
cprotect, 609
CPSR. *See* Current Program Status Register
CPU
  affinity, threads, 415
  multithreading, 93
  processes, 92–93
  threads, 408
  yielding, 415
cpuid, 195, 200
cpu_mode_init(), 279
cpus, 330
cpusubtype, 100
cputype, 100
crash_mover, 244
CrashReporter, 171–173, 442
CrashReportSupport, 336
CreateEvent*, 189
CreateRemoteThread(), 407
CRO, 266–267
crond, 231
CRs. *See* Control Registers
crypto, 307
cs_debug, 562
cs_enforcement_disable, 562
C-shell, 21
csops, 110
csreq(1), 80
CSS. *See* Cascading Style Sheet
CTFCONVERT, 302
CTFMerge, 300
CTL_*, 552–553
CTLIOCGINF ioctl(2), 682
Current Program Status Register (CPSR), 267–268, 777–778
cut(1), 409
cvmsServ, 242
cvmsServer, 242
Cxxfilt, 300
Cylinder/Head/Sector (C/H/S), 568

D

.d, DTrace, 149
-d, dtruss, 151
D language, 147–150
DAAP. *See* Digital Audio Access Protocol
DADissenterCreate, 589
-daemon, 18
daemons. *See also specific daemons*
  launchd, 229
  Spotlight, 20
  system configuration, 67
Darwin
  architecture, 15–17
  Cheetah, 6
  GDB, 181
  iOS, 12
  Jaguar, 6
  LibC, 139
  Mountain Lion, 9
  notifications, 78
  Panther, 6
  Snow Leopard, 8
  Tiger, 7
  UNIX, 5, 20–22
data, 254
Data?, 85
_DATA(), 107
__DATA, 125, 134
Data Execution Prevention (DEP), 522, 549
data forks, 611
Data Link Interface Layer (DLIL), 680
DATA_HUB_PROTOCOL, 191
data_list, 454
data_request, 493, 494
data_return, 482, 494
date, 254
DB_*, 332–333
DB_ARP, 333
DBG_APPS(33), 167
DBG_MACH_SCHED, 430
DBG_MIG(255), 168
DBG_PERF(37), 168
DbgPrintKernel, 332
ddb, 307
deadfs, 586–587
deadline timers, 432–433
DEAD_NAME, 350
DEBUG, 308
debug, 56, 331, 332
Debug Registers (DRs), 775
debugging, 147–182
  applications
    crashes, 170–176
    hangs, 173–174
    sampling, 173–174
  DTrace, 147–154
  exception ports, 439
  GDB, 181–182

hfsleuth, 577
init_kdp(), 318
kernel, 332–340
launchd, 228
LLDB, 182
Mach zones, 473
memory leaks, 176–178
UNIX, 178–180
VMWare, 333
DebugPrintFilter, 332
debugserver, 87–88
decmpfs, 608
decmpfs_file_is_compressed, 613
decompression, 613–616
decryptVolume, 576
default:, 333
default directories, 25
Default Freezer, 494
DEFAULT_APPLICATION, 423
default_freezer, 529, 547
default_pager, 307, 448, 499
default_pager_*, 487
defaults(1), 173
#defines, 305, 318, 463, 650
defragmentation, 622–623
.defs, 353
<defunct>, 93
DEP. *See* Data Execution Prevention
DEPRESSPRI, 412
dev, 307
/dev, 22
/dev/auditpipe, 60
/Developer, 23, 24
DeveloperDiskImage.dmg, 24
devfs*, 584
device drivers
I/O Kit, 738
user mode, 749–750
NeXTSTEP, 4
Device Firmware Update (DFU), 211, 213
device tree, 196–198
iOS, 224–225
device_pager, 448
DFLAGS(2), 169
DFU. *See* Device Firmware Update
diag, 331
diagCall(), 292–295
diag.h, 487
diagnose, 86
diagnostic system calls, 292–295
dictionary, 255
didReceiveMemoryWarning, 139, 545
Differentiated Services (DiffSrv), 706
Digital Audio Access Protocol (DAAP), 652
di_load_controller, 592
DIOCADDRULE, 698
DIOCGETRULE, 698
direct_dispatch_to_idle_processors, 430
directories
GUID, 25
UNIX, 22–24
iOS, 23–24
OS X, 23
DirectoryServices, 37
-disable_aslr, 330
DiscRecording*, 37
disk image files, 589–591
DiskArbitration, 37
diskarbitrationd, 587–589
DiskImageMounter.app, 589
DISK_IO_PROTOCOL, 190
diskutil(8), 575
dispatch_get_global_queue(), 145
dispatch_queue_create(), 146, 257
DISPATCH_QUEUE_PRIORITY_*, 145, 550
--display, 86
distnoted(8), 78
disym(void *handle, char *sym), 122
ditto(1), 613
dladdr(), 122
dlerror(), 122
DLIL. *See* Data Link Interface Layer
DLIL(8), 167
dlil_output(), 693
dlopen(), 122
dlopen_preflight(), 122
.dmg, 589–591
dmgextractor, 589
DNS
mDNS, 652
reverse, 18–19, 30
Dock.app, 247
document type definition (DTD), 26
Documents, 25
do_init_slave(), 313–314
domains
initialization, 673–675
protosws, 669–673
sockets, UNIX, 651
XNU, 675
domaininit(), 673
Don't Steal Mac OS X (DSMOS), 491, 716–717
do_priority_computation, 411, 412
double, 254
double fault, 270
downgrade attacks, 213–214
do_write, 599–600
dp_backing_store.c, 487
dp_memory_object.c, 487
Draves, Richard, 418
DrawBootGraphics, 200
DrawSprocket, 37
drivers. *See also* device drivers
I/O Kit
kernel, 755–769
matching, 755–757
model, 761–763
NDIS, 739

DriverKit, 4
DRIVERS(6), 167
DRs. *See* Debug Registers
dscl(8), 65–66
DSMOS. *See* Don't Steal Mac OS X
DTD. *See* document type definition
DTrace, 147–154
  Leopard, 7
Dtrace, 300
dTrace, 202–203
dtrace, 152
dtruss, 150–151
dummynet(4), 705–707
DumpPanic(), 242
dup2(), 240
Durango, 12
DVComponentGlue, 36
DVDPlayback, 37
dyld
  environment variables, 128–130
  function interposing, 125–128
  kernel, 111
  load commands, 114
  shared library cache, 121
  two-level namespace, 125
DYLD_*, 125, 126, 128, 129–130, 493
.dyld(1), 44
DYLD(31), 167
dyldinfo(1), 114
dyld_stub_linker, 119
dyld_stub_puts, 120
.dylib, 42
dynamic defragmentation, 622–623
dynamic libraries, 111–130
dynamic resizing, 620
dynamic_pager(8), 142–143, 488, 498–499
DYNAMIC_PAGER_PORT, 499

E

-e, 59, 409
  ACLs, 578
  dtruss, 151
EAPOL, 653
EAX, 278
EDR, SUN-RPC, 353
EEPROM. *See* Electronically-Erasable Programmable Read Only Memory
EFI. *See* Extensible Firmware Interface
efi-boot-*, 193, 205
efi_init(), 203
efi_set_tables_[32|64], 203
EFI_STATUS, 187
EFI_SUCCESS, 187
EFI_SYSTEM_TABLE, 187–188
EFLAGS, 295, 296, 774–775
EFLAGS(1), 169
Electronically-Erasable Programmable Read Only Memory (EEPROM), 184
ELF. *See* Executable and Library Format
EMMI. *See* External Memory Manager Interface
EMT, 534
en, 679
ENABLE(3), 169
Enable Transactions, 236
enable_preemption(), 426
encryptVolume, 576
ENDIAN_MAGIC(), 644
EndOfAllTime, 435–436
endpoint, 254
enterlctx(), 515
entitlements
  iOS, 87–89, 97
  OS X, 97
  sandboxing, 83–89
entry points, 130
environment variables, 128–130
EPPC. *See* Event Process-to-Process Communication
errno_t, 681
error, 255
Essentials.pkg, 216
etap_trace_thread, 405
/etc/syslog.conf, 70
/etc/ttys, 18, 22
etimer_intr, 434
etimer_resync_deadlines(), 435
EULA, 10
Event Process-to-Process Communication (EPPC), 652
EventKit*, 37
every, 409
EVFILT_*, 57–58
exc, 352
EXC_*, 438, 534
exceptions
  Intel trap handlers, 269–270
  involuntary user/kernel transition, 269–270
  Mach scheduling, 436–445
  ports, 436
    debugging, 439
  UNIX, 529–534
  vector, ARM, 268
EXCEPTION_DEFAULT, 439
exception_deliver(), 439
ExceptionHandling, 37
EXCEPTION_STATE*, 439
exception_triage(), 438, 439
ExceptionVectorsBase, 275–276
EXC_SOFTWARE, 534
exec(), 240
exec_activate_image(), 521
exec_archhandler, 519
execargs_alloc, 521
exec_save_path, 521
execsw, 516, 518
executables
  entry points, 130

libraries, 111
Mach-O, segments and sections, 108
PEs, 187
processes, 98
UNIX, 98
Executable and Library Format (ELF), 102, 502
binaries, 15–16
execution
conditional, 784–785
DEP, 522, 549
policies, 527–528
threads, 408
execve(), 130, 327, 520–521
exit(), 117
exit(2), 92, 93, 143
ExitBootServices(), 188
explicit preemption, 418–420
Exposé, 6
extended attributes, 577, 608–611
EXTENDED_POLICY, 421
Extensible Firmware Interface (EFI), 10, 185–210
architecture, 186
ASLR, 549
binaries, 187
BIOS, 184
Boot Camp, 204
Boot Services, 188–191
BSD, 203
console protocols, 189–190
GUIDs, boot.efi, 192–193
kernel, 203
Mach, 203
media access protocols, 190
Platform Expert, 303
protocols, 188–191
runtime services, 191–192
variables, APPLE_BOOT_GUID, 193
extents, 577
overflow, 640
external data representation (XDR), 351
External Memory Manager Interface (EMMI), 480
ExternalAccessory, 37
extract_heap, 706

F

-f, dtruss, 150
-F, dynamic-pager(8), 143
FaceTime, 11
facility, 70
FairplaydUnfreed, 244
fairplay.d.XXX, 244
fairshare_dequeue(), 430
fairshare_enqueue(), 430
fairshare_init(), 430
fairshare_runq_count(), 430
fairshare_runq_stats_count_sum(), 430
false, 254
FAT. *See* File Allocation Table
faults, 270
fbt, 152
fd, 255
fdcopy(), 515
FDE. *See* full disk encryption
fd_ofiles, 601
f_flob, 602
fg_data, 603
fg_type, 603
FIFOfs, 584–586
file(1), 99, 212–213
File Allocation Table (FAT), 580, 625
file systems
CDDAFS, 581
CDFS/ISO-9660, 582
generic concepts, 577–579
HFS, 4, 579
HFS+, 21–22, 579–580, 607–648
ACLs, 608
B-Tree, 624–645
case sensitivity, 619
compression, 612–617
decompression, 613–616
design concepts, 624
disk image files, 589
dynamic defragmentation, 622–623
dynamic resizing, 620
extended attributes, 608–611
finderInfo, 205–106
forks, 611–612
hfsleuth, 577
hot files, 621–622
journaling, 619–620
metadata zone, 620–621
OS X Finder, 617–618
panic(), 333
permissions, 577, 639
sandboxing, 84
status notifications, 647
timestamps, 607–608
Unicode, 617
VFS, 591
links, 578–579
native, 579–580
networks, 582–583
NFS, 582–583
NTFS, 578, 581, 591, 624
OS X, 587–589
pseudo, 583–587
shortcuts, 578–579
timestamps, 578
VFS, 22, 577, 591–600
fsctl(2), 645–646
FUSE, 597–605
kernel, 645–648
mount entry, 592–595
struct vnode, 595–597
sysctl(2), 646–647

vnode, 595–597
File systems in USEr space (FUSE), 597–605
File Transfer Protocol (FTP), 583, 598
fileglob, 602–603, 605
FILE_PROTOCOL, 190
filesize, 107
FileVault, 6, 8
filterfn, 508
FinalizeBootStruct, 201
Finder
bundles, 25
GUI, 247–248
OS X, 247–248
HFS+, 617–618
Quicklook, 18–19
Spotlight, 19–20
UI, 250–253
UNIX directories, 22
FinderInfo, 608
finderInfo, 205–206
FIQ, 268
firmware, 184
DFU, 211, 213
EFI, 10, 185–210
architecture, 186
ASLR, 549
binaries, 187
BIOS, 184
Boot Camp, 204
Boot Services, 188–191
BSD, 203
console protocols, 189–190
GUIDs, boot.efi, 192–193
kernel, 203
Mach, 203
media access protocols, 190
Platform Expert, 303
protocols, 188–191
runtime services, 191–192
variables, APPLE_BOOT_GUID, 193
UEFI, 185–186, 191
FixedPriorityString(), 427
FixedPriorityWithPsetRunqueueString(), 427
flags, 107
flashing, 184
flavor, 156
fleh_irq, 426
fleh_swi, 280, 438
floating point registers, 774, 777
fmm-hostname, 193
folderCount, 633
f_ops, 605
Force Quit, 174
ForceFeedback, 37
fo_read, 604
FOREGROUND_APPLICATION, 422
fork(), 512, 514, 515
forks, 611–612, 639–640
forkproc(), 514, 515–516
Foundation, 37
fp_data, 667
FPE, 534
fpextovrfit, 438
fp_lookup, 601–602
free(), 127
memory leaks, 176
vm_allocate, 453
FreeBSD, 55
FreePages, 189
FreePool, 189
FREE_ZONE, 544
friends, 44
fsboot(), 212
fsck(1), 217
fsck_cs(8), 576
fsctl(2), 645–646
FSE_CHOWN, 5
FSE_CONTENT_MODIFIED, 5
FSE_CREATE_DIR, 5
FSE_CREATE_FILE, 5
FSE_DELETE, 5
FSE_EVENTS_DROPPED, 75
FSE_FINDER_INFO_CHANGED, 5
FSE_RENAME, 5
FSE_STAT_CHANGED, 5
FSEvents, 7, 74–78, 237, 242
fseventsd, 75, 242
FSEventStreamCreate, 75
-fstack-protector, 130
fstat1(), 603–604
fs_usage(1), 76, 165
FSYSTEM(3), 166
FTP. *See* File Transfer Protocol
full disk encryption (FDE), 204, 575
function interposing, 125–128
FUSE. *See* File systems in USEr space
fuse_main(), 598
fuse_operations, 599
fuser(1), 156, 180
fw, 679
FWAUserLib, 37
fwkpfv(1), 333
FXR, 305

G

-g, dtruss, 151
GameKit, 37
garbage collection
Mach zones, 471–473
Objective-C, 545
vm_pageout(), 497
GateKeeper, 84
GCD. *See* Grand Central Dispatch
GDB. *See* GNU Debugger
gdb, 118, 119, 337

gen, 303
general protection fault, 267
Generic Security Services (GSS), 37
getaudit(), 61
getaudit_addr(), 61
get_bsdtask_info(task_t), 511
get_bsdthread_info(thread_t), 511
GETBUF(5), 169
get_dp_control_port, 376
GetMemoryMap, 189
GetNextVariableName, 192
GETREG(9), 169
getrlimit(2), 398, 512
get/set inferior-auto-start-dyld, 181
get/set inferior-bind-exdception-port, 181
get/set inferior-ptrace[-on-attach], 181
get_special_port(), 400
get-task-allow, 444
GetTime, 192
GetVariable, 192
GetWakeupTime, 192
GID. *See* group identifier
gif, 678
gif(4), 655
gif_clone_create(), 685
GLKit, 37
Globally Unique Identifier Partition Table (GPT), 572–574, 576–577
GLUT, 37
GNU Debugger (GDB), 181–182, 458
GNUStep port, 4
GPT. *See* Globally Unique Identifier Partition Table
GPU, 7
GrabFS, 598
Grand and Unified Bootloader (GRUB), 184
Grand Central Dispatch (GCD), 7, 79, 145–146, 253, 550
Graphical User Interface (GUI), 15
  Aqua, 17–18
  dtruss, 151
  Finder, 247–248
  Force Quit, 174
  Leopard, 7
  Lion, 8
  Mac OS Classic, 3–4
  NeXTSTEP, 4
  OS X, 215
  shells, launchd, 246–253
  SpringBoard, 13, 248–253
  Tiger, 6
graphics, Quartz Extreme, 6
GRAPHICS_OUTPUT_PROTOCOL, 190
GRAPHICS_SERVER, 423
grep, 306
groups
  lock, 361
  processes, 91
    BSD, 507–508
group identifier (GID), 97
GRRRString(), 427
GRUB. *See* Grand and Unified Bootloader
GSEvent, 253
GSS. *See* Generic Security Services
GuardMalloc, 125
GUI. *See* Graphical User Interface
GUIDs
  directories, 25
  EFI, boot.efi, 192–193
  protocols, UEFI, 191
GUID/tmp, 25

H

-h, 105
-H, dynamic-pager(8), 143
HandleProtocol, 189
handoffs, 415–416
hard links, 578–579, 639
hardening, 13
HardResourceLimits, 236
hardware
  BSD signals, 534
  CHUD, 154–155, 373
  EFI, 189
  interrupts, 431
  non-Apple, 10
  pop, timer interrupts, 435–436
hardware extraction, XNU kernel, 295–297
hdiutil, 213, 568–569, 589
header records, 627
heap(1), 177
heaps, 139–140
heap spray, 103
Heavenly, 11
hertz_tick(), 431
HFS. *See* Hierarchical File System
hfs, 307
HFS+. *See* Hierarchical File System Plus
HFS_BULKACCESS_FSCTL, 646
HFSCatalogFileRecord, 637
HFSCatalogFolderRecord, 637
HFS_CHANGE_NEXT_ALLOCATION, 646
HFS_CLRBACKINGSTOREINFO, 646
--hfsCompression, 613
HFS_DISABLE_JOURNALING, 647
HFS_DISABLE_METAZONE, 646
HFS_ENABLE_JOURNALING, 647
HFS_ENABLE_RESIZE_DEBUG, 647
HFS_ENCODINGBIAS, 647
HFS_ENCODINGHINT, 647
HFS_EXTEND_FS, 647
  sysctl(2), 620
HFS_FSCTL_GET_JOURNAL_INFO, 646
HFS_FSCTL_SET_DESIRED_DISK, 646
HFS_FSCTL_SET_LOW_DISK, 646
HFS_FSCTL_SET_VERY_LOW_DISK, 646
HFS_GET_BOOT_INFO, 646

HFS_GET_JOURNAL_INFO, 647
HFS_GETPATH, 646
<hfs/hfs_format.h>, 625
hfsleuth, 577, 613, 628, 635
HFS_MARK_BOOT_CORRUPT, 646
HFS_NEXT_LINK, 646
HFSPlusCatalogKey, 633
HFSPlusCatalogThread, 633
HFSPlusForkData, 639
HFS_PREV_LINK, 646
hfs_readwrite.c, 622–623
hfs_relocate(), 622–623
HFS_RESIZE_PROGRESS, 646
HFS_RESIZE_VOLUME, 646
  ioctl(2), 620
HFS_SET_ALWAYS_ZEROFILL, 646
HFS_SETBACKINGSTOREINFO, 646
HFS_SET_BOOT_INFO, 646
HFS_SET_PKG_EXTENSIONS, 647
HFS_SET_XATTREXTENTS_STATE, 646
HFS_VOLUME_STATUS, 646
HFSX, 619
hibernate_newruntime_map(), 203
hidden, 248
Hierarchical File System (HFS), 4, 579
Hierarchical File System Plus (HFS+), 21–22, 579–580, 607–648
  ACLs, 608
  B-Tree, 624–645
  case sensitivity, 619
  compression, 612–617
  decompression, 613–616
  design concepts, 624
  disk image files, 589
  dynamic defragmentation, 622–623
  dynamic resizing, 620
  extended attributes, 608–611
  file systems, status notifications, 647
  finderInfo, 205–106
  forks, 611–612
  hfsleuth, 577
  hot files, 621–622
  journaling, 619–620
  journaling, B-Tree, 642–645
  metadata zone, 620–621
  OS X Finder, 617–618
  panic(), 333
  permissions, 577, 639
  sandboxing, 84
  timestamps, 607–608
  Unicode, 617
  VFS, 591
  volume header, B-Tree, 631–632
himemory_mode, 331
HNDL_ALLINTRS, 274
HNDL_ALLTRAPS, 274
hndl_alltraps, 273–274
Hoodoo, 12
host, 367–371
HOST_AMFID_PORT(18), 373
HOST_AUDIT_CONTROL(9), 372
HOST_AUTOMOUNTD_PORT(11), 373
HOST_CHUD_PORT(16), 373
host_default_memory_manager, 375
HOST_DYNAMIC_PAGER_PORT(8), 372
host_get_boot_info, 374
host_get_clock_control, 375
host_get_clock_service, 368, 379
host_get_host_priv_port(), 374
host_get_special_port, 371–374, 375
host_get_UNDServer, 376
HOST_GSSD_PORT(19), 373
host_info, 355, 368
hostinfo(1), 369–371
HOST_KEXTD_PORT(15), 373
host_load_symbol_table, 376
HOST_LOCKD_PORT(12), 373
host_lockgroup_info, 369
host_notify_reply, 352
host_priv, 352
host_priv_statistics, 374
host_processor_info, 368
host_processors, 375
host_processor_sets, 376
host_reboot, 374
HOST_SEATBELT_PORT, 561
HOST_SEATBELT_PORT(14), 373
host_security, 352
host_set_exception_ports, 376
  bsdinit_task(), 530
host_set_special port, 376
host_set_UNDServer, 376
HostSpecialPort, 235
host_statistics, 369
HOST_UNFREED_PORT(17), 373
HOST_USER_NOTIFICATION_PORT(10), 372
host_virtual_physical_table(), 456
host_virtual_physical_table_info, 369
hot files, 621–622, 641–642
HTML, 241
  widgets, 45, 47
hw, 56
hw_lock_t, 364
hybrid kernel, 265–267
hyperthreading, 408, 415

I

i386, 303
[i386|arm]_init, 311–313
i386_astintr(), 425
i386_exception(), 274, 438
i386_init(), 311, 395
i386_init_slave(), 311, 313
i386/machine/ppc, 307
i386/ppc/x86_64, 307
i386_THREAD_STATE, 109

i386/trap.c, 274
iAD, 37
iBoot, 210–214
ICADevices, 37
iCloud, 8, 12
.icns, 29
icons, 29
IDL. *See* Interface Definition Language
idle_queue, 384
idle_thread, 384
IDT. *See* Interrupt Descriptor Table
IDT_ENTRY_WRAPPER, 272
ifconfig(8), 679
#ifdef, 12
#ifdef'ed, 314
ifnet, 680–682
ifnet_allocate(), 682
ifnet_attach(), 682
ifnet_attach_proto_param(), 682
ifnet_reference(), 682
ifnet_release(), 682
if_output, 693
ILL, 534
imageboot_needed(), 590
imageboot_setup(), 590
ImageCaptureCore, 37
ImageIO, 37
image_params, 518
IMCore, 37
IMG3, 221–222
implicit preemption, 420–423
__IMPORT, 134
IMServicePlugin, 37
#include, 355
INET, 234
inetd, 232–234, 238
info mach-port <task> <port>, 181
info mach-ports <task>, 181
info mach-region <address>, 181
info mach-regions, 181
info mach-task <task>, 181
info mach-tasks, 181
info mach-thread <thread>, 181
info mach-threads <task>, 181
Info.plist, 26–28, 717, 721
  I/O Kit, 741
init(), 93, 230, 428
inital_thread_sched_mode(), 429
InitBootStruct, 200
initialization
  BSD, 318
  domains, 673–675
  launchd, 230–231
initializeConsole, 195
initial_quantum_size(), 429
init_kdp(), 318
InitMemoryConfig, 198
initprot, 107
init_proto(), 674
InitSupportedCPUTypes, 198
inode, 608
inotify, 74
InputMethodKit, 37
insertions, B-Tree, 624, 636
installation
  images, boot process, 214–225
  OS X, 214–219
InstallESD.dmg, 214, 215
InstallProtocolInterface, 189
install_real_mode_bootstrap(), 316, 329
instances, processes, 91
InstantMessage, 38
Int 13h, 183
int64, 254
IntallerPlugins, 38
Integrated Services (IntSrv), 706
Intel
  architecture, Mach physical memory management, 465–467
  atomicity, 787–788
  IDT, 268
  kernel, 266–267
  locking, 787–788
  multithreading, 786
  OS X, 261
  registers, 773–776
  32-bit, process address space, 132
  trap handlers, 268–278
    ARM, 275–278
    XNU, 272–275
  VM, 791
  x86, 6
interfaces. *See also* Graphical User Interface
  EMMI, 480
  filters, packet filtering, 701
  ifconfig(8), 679
  iOS, 678–680
  KPI functions, 682
  layer III, 678–686
  NDIS, 739
  OS X, 678–680
  protocols, 677–678
  utun, 682–686
Interface Definition Language (IDL), 351
internationalization, 29
__interpose, 125
interpreters, 98
INTerrupt, 278
Interrupt(), 275
interrupts
  ARM, 296
  asynchronous, 431
  hardware, 431
  involuntary user/kernel transition, 270–271
  I/O Kit, 765–768
  PIC, 270
  PPC, 296–297
  SWI, 275, 280

  synchronous, 278
  timer, 431–436
Interrupt Descriptor Table (IDT), 268, 438
Interrupt Handler, 270
Interrupt Request (IRQ), 270–271
Interrupt Service Routine (ISR), 268, 270
IntSrv. *See* Integrated Services
involuntary user/kernel transition
  exceptions, 269–270
  interrupts, 270–271
io, 331
I/O. *See also* Basic Input Output System
  launchd, 236
  policies, 527–528
  processes, 93, 600–605
I/O Kit, 737–771
  BSD, 769–771
  C++, 737, 740–741
  device drivers, 738
  diagnostics, 753–755
  diskarbitrationd, 587
  driver matching, 755–757
  driver model, 761–763
  families, 757–761
  Info.plist, 741
  interrupts, 765–768
  I/O registry, 740, 743–746
  IOMalloc, 479
  IOMemoryDescriptor, 485
  IOPlatformExpert, 304–305
  kernel drivers, 755–769
  kernel_bootstrap_thread, 318
  launchd, 237–238
  libkern, 742–743
  loops, 740
  memory management, 769
  name mangling, 740
  namespaces, 740
  NDIS, 739
  OSObject, 739, 741
  Platform Expert, 303
  power management, 751–753
  subsystems, 753
  user mode, 740, 746–755
    device drivers, 749–750
    I/O registry, 747–749
    plug and play, 750–751
  XNU, 50
I/O registry, 740, 743–749
IOACPIPlane, 744
ioalloccount(8), 753
IOAllowPowerChange(), 752
IOBluetooth, 38
IOBluetoothUI, 38
ioclasscount(8), 754
IOCommandate, 764
IOCopyAssertionsByProcess(), 753
ioctl(), 566–567, 672
ioctl(2), 75, 620
IODeviceTree, 196–198, 744
IOFilterInterruptEventSource, 764
IOFilterInterruptSource, 764
IOGENERALMEMORYDESCRIPTOR::doMap, 488
IOHDIXController, 592
IOHDIXController.kext, 589
IOHibernateIO.cpp, 193
IOHibernatePrivate.h, 193
IOKernelConfigTables, 755
IOKit, 38, 79, 196
iokit, 307
IOKIT(5), 167
iokit/bsddev/DINetBookHook.cpp, 590
IOMalloc, 479
IOMemoryDescriptor, 485
IONetworkController::outputPacket(), 693
IONotificationPortCreate, 750
IOPlatformDevice, 755
IOPlatformExpert, 304–305
IOPMAssertionCreateWithName(), 752
IOPMSchedulePowerEvent(), 752
IOPMSleepSystem(), 752
iopolicysys(), 527–528
IOPower, 744, 751
ioreg, 194, 224
ioreg(8), 196
IORegisterForSystemPower(), 752
IORegistry, 717
IORegistryEntry, 746
iOS
  Apple TV, 11–12
  architecture, 15–51
  ARM, 5, 261
  ASLR, 173
  BackupAgent, 173
  CHUD, 155
  default directories, 25
  Default Freezer, 494
  device tree, 224–225
  downgrade attacks, 213–214
  DTrace, 148
  entitlements, 87–89, 97
  fleh_irq, 426
  frameworks, 32–43
  GDB, 182
  hiding applications, 250
  hostinfo(1), 369
  iBoot, 210–214
  interfaces, 678–680
  iPad, 11
  iPad 2, 11–12
  iPhone, 11
  iPhone 4, 11–12
  .ipsw, 219–225
  IPv6, 654
  jailbreaking, 210
  Jetsam, 236–237, 546–548
  kernel, 23
    ExceptionVectorsBase, 275–276
    jailbreaking, 457
    system calls, 286–287

versions, 14–15
kernelcache, 719
LaunchDaemons, 241–253
libraries, 42–44
lockdownd, 234
Mach, 343
scheduling exceptions, 445
Mountain Lion, 9
network stack, 649
OS X, 12–15
merger, 16
PF_NDRV, 652
pid_shutdown_sockets, 94
process hibernation, 547–548
replay attacks, 213–214
sandboxing, 81–82
security, 79–90
Setup.App, 249
shared cache, 121
sleep, 329
SpringBoard, 248–253
start(), 310–311
32-bit, process address space, 133–134
UNIX directories, 23–24
versions, 10–12
XNU, 310
XPC, 253–257
IOService, 744, 746
IOServiceAddInterestNotification(), 750
IOSurface, 38
iothread, 496
IOUSB, 744
IOUserEthernetController, 656
IOWorkLoop, 764–765
IP filters, 698–701
ip6config, 655
iPad, 11
iPad 2, 11–12
ipc, 307
IPC services, 234, 357–360
ipc_kmsg_send(), 359
ipc_kobject_server(), 359
ip_clock_enable(), 379
ipc_mqueue_*, 359–360
ipc_port_t, 357
ipf_filter, 699
ipfw(8), 696–697
iPhone, 11
iPhone 4, 11–12
ip_output_list(), 692
IPSec Key Management sockets, 654
.ipsw, 219–225
iptap_init(), 656
IPv4, 651–652
IPv6, 654–655
IRQ. *See* Interrupt Request
IRQ, 268
isolated virtual memory, 130
ISR. *See* Interrupt Service Routine
iTunesArtwork, 25
iTunesMetaData.plist, 25
Itunesstored(), 244
iTunesStore.daemon.*itunesstored.*, 244

J

Jaguar, 6
jailbreaking, 13
ASLR, 549
CrashReporter, 173
ioreg, 224
iOS, 210
kernel, 457
LC_CODE_SIGNATURE, 110
lockdownd, 245
logging, 71–72
sandboxing, 81–82
SSH, 21
unionfs, 587
versions, 213
Jasper, 11
Java, 44–45
java, 44
JavaApplicationLauncher.framework, 45
javac, 44
JavaEmbedding, 38
JavaEmbedding.framework, 44
JavaFrameEmbedding, 38
JavaFrameEmbedding.framework, 44
JavaLaunching.framework, 45
JavaScript, 45
widgets, 47
JavaScript Object Notation (JSON), 26
JavaScriptCore, 38
JavaTools.pkg, 216
JavaVM, 38
Jetsam, 236–237, 546–548
jetsam_flags_procs(), 546
jetsam_kill_hiwat_proc(), 546
jetsam_kill_top_proc(), 546
jetsam_snapshot_procs(), 546
jetsam_task_page_count(), 546
JIT. *See* Just-In-Time
JOURNAL_HEADER_MAGIC, 644
journalInfoBlock, 643
journaling, HFS+, 21, 619–620, 642–645
JSON. *See* JavaScript Object Notation
jtool, 721
Just-In-Time (JIT), 457

K

-k, 149–150
kalloc(), 470, 477–479
kas_info(), 549
KAUTH, 578
kBTHeaderNode(1), 627
kdebug, 79, 165–170, 434

kdebug_trace, 169
KDGETENTROPY(16), 170
KDP. *See* Kernel Debugger Protocol
kdp, 307
kdp_match_name, 332
kdp_register_send_received(), 332
Kerberos, 38
kern, 56, 307
KERN_BAD_ACCESS, 534
kern.coredump, 171
kern.corfile, 171
kern_ctl_reg, 656
kernel, 12–13
  arbiter, 262
  architecture, 261–287
  ARM, 267–268
  asynchronous *versus* synchronous, 268
  cache, 23
  clients, 261
  control protocol, 655–657
  debugging, 332–340
  drivers, I/O Kit, 755–769
  DTrace, 152
  dyld, 111
  EFI, 203
  event protocol, 657–658
  extensions, 711–735
    code signing, 712–713
    modular architecture, 712–713
    pre-linking, 713
  FUSE, 598
  hybrid, 265–267
  Intel, 266–267
  iOS, 23
    ExceptionVectorsBase, 275–276
    jailbreaking, 457
    system calls, 286–287
    versions, 14–15
  kdebug, 166
  kext, 725–735
  kprintf(), 313
  launchd, 227
  Linux, 262–264, 303
  logging, 70
  MAC, 63
  Mach, 303
    scheduling, 406–407
  memory, 198
    Mach, 473–480
  microkernels, 264–265
  monolithic, 262–264
  Mountain Lion, 9
  NeXTSTEP, 4
  OS X versions, 14–15
  panic(), 333–340
  permissions, 262
  Platform Expert, 296, 303
  printf(), 313
  runtime services, 191
  scheduling, 262
  security services, 262
  serial, 313
  64-bit, 264
  system calls, 261, 268, 283–295
  32-bit, 266
  tick-less, 432
  Tiger, 6
  trap handlers, 334
  universal binaries, 100
  user mode/kernel mode, 266–282
    involuntary transition, 269–271
    transition, 268–282
    voluntary transition, 278–282
  VFS, 22, 645–648
  virtualization, 262
  XNU, 50
    architecture, 302–305
    boot, 299–340
    hardware extraction, 295–297
Kernel, 38, 199
kernel Architecture, 199
Kernel Cache, 199
Kernel Debug Kit, 337
Kernel Debugger Protocol (KDP), 331, 332
Kernel Flags, 199
kernel mode
  BSD process creation, 513–516
  involuntary transition
    exceptions, 269–270
    interrupts, 270–271
  sockets, 667–668
  voluntary transition, 278–282
kernel_bootstrap(), 314–316, 379
kernel_bootstrap_thread(), 318–320, 379, 495
kernelcache, 201, 211, 214, 719–723
kernel_create_thread(), 318
KERNEL_DEBUG_CONSTANT, 169
kernel_memory_allocate(), 469, 473–476
kernel_task, 395, 402
-kernel_text_ps-4k, 330
kernel_thread_create, 416–417
kernel_thread_start_priority, 416
kern_hibernation_wakeup, 547
kern_invalid(), 290
KERN_INVALID_ARGMENT, 290
KERN_KD*, 169–170
KERN_KDENABLE, 169
KERN_NOT_SUPPORTED, 345
kern_os_malloc(), 479
KERN_PANICINFO_TEST, 336
KERN_PROCARGS, 156
kern_return_t, 353
KERN_SUCCESS, 436
kern.sugid_coredump, 171
KEV_APPLESHARE_CLASS(4), 657
kevent(2), 556
kevent64(2), 556
KEV_FIREWALL_CLASS(5), 657
KEV_IEEE80211_CLASS(6), 657
KEV_IOKIT_CLASS(2), 657

KEV_NETWORK_CLASS(1), 657
KEV_SYSTEM_CLASS(3), 657
kext, 713–735
  kernel, 725–735
  loading, 732–733
  MIG, 734–735
  otool(1), 340
  plist, 718
  programmer's view of, 724–725
  security, 718
  structure, 717–718
kextd, 733–734
kextd(8), 728
kextlog, 331
kext_request, 377, 733–734
kextstat(8), 714, 727
Kext-tools, 300
keyed records, 627
kHFS*, 633
Khronos, 7
KILL(), 534
kill -9, 236, 247, 253
kill -15, 236
kill -CONT, 247
kill -STOP, 247
killall(1), 248
killpg1(), 508, 535
killpg1_callback(), 535
kIODTNVRAMPanicInfoKey, 336
KirkWood, 11
kJIJournalInFsMask(), 644
kJIJournalOnOtherDeviceMask(), 644
-klog_in 1, 70
kmem, 331
kmem_alloc(), 477
kmem_alloc_contig(), 477
kmem_alloc_pageable(), 477
kmem_alloc_pages(), 477
kmeminit(), 544
kmod_get_info(), 368, 726
kmod_info_t, 725
kmzones[], 542
Kodiak, 5
Korn shell, 21
KPI functions
  interfaces, 682
  protocols, 677
kpi_socket, 667
kprintf(), 313, 333
kqueues, 57–59, 237, 555–556
kSBXProfileNoNetwork, 83

L

-l, 330, 409
-L, dynamic-pager(8), 143
Label, 238
labels, MAC, 62
<language>.pkg, 216
LAPIC. *See* Local Advanced Programmable Interrupt Controller
Large Block Address (LBA), 568
lastMountedVersion, 642–645
latency(1), 165
LatentSemanticMapping, 38
Launch Daemon, 79, 229
launchctl(), 228, 240–241
  bstree, 253
launchctl(1), 240–246
launchd, 93, 227–257, 326
  agents, 229
  atd, 231
  autorun, 237
  bsdinit_task, 227
  crond, 231
  daemons, 229
  GUI shells, 246–253
  inetd, 232–234
  init, 230
  initialization, 230–231
  I/O Kit, 237–238
  kernel, 227
  LaunchDaemons, 241–246
  load_init_program(), 227, 326
  lockdownd, 245–246
  mach_init(), 234–236, 351
  MachServices, 373
  parameters, 240
  PID, 228
  resource limits, 236–237
  socket descriptors, 240
  starting, 227–241
  syslogd, 72
  system-wide *versus* per-user, 228
  throttling, 236–237
  transactions, 236
  wrappers, 240–241
  xinetd, 232–234
  XPC, 253–257
launchd(8), 59
LAUNCHD(34), 168
.launchd_log_debug, 228
.launchd_log_shutdown, 228
.launchd_use_gmalloc, 228
<launch.h>, 235, 240
launch_msg(), 240
LaunchPad, 13
layer III interfaces, 678–686
layer III network protocols, 676–678
layer IV transport protocols, 668–669
layer V sockets, 660–668
LBA. *See* Large Block Address
LC_CODE_SIGNATURE, 106, 110
LC_DYLINKER, 111
LC_DYSMTAB, 114
LC_ENCRYPTION_INFO, 106
LC_FUNCTION_STARTS, 114
LC_ID_DYLIB, 114
LCK_ATTR_DEBUG, 361

lck_grp_t, 361
lck_mtx_destroy, 364
lck_mtx_free, 363, 364
lck_mtx_init, 364
lck_mtx_lock, 364
lck_mtx_t, 364
lck_mtx_try_lock, 364
lck_mtx_unlock, 364
lck_rw_destroy, 363
lck_rw_init, 363
lck_rw_lock, 363
lck_rw_t, 363
lck_rw_unlock, 363
lck_spin_t, 364
LC_LOAD_DYLIB, 114, 115
LC_LOAD_DYLINKER, 106
LC_MAIN, 110
LC_REEXPORT_DYLIB, 114, 115
LC_SEGMENT, 106, 107–109
LC_SEGMENT(64), 130
LC_SEGMENT_64, 106, 107–109
LC_SOURCE_VERSION, 114
LC_SYMTAB, 114, 115
LC_THREAD, 106, 109
LC_UNIXTHREAD, 106, 109, 110, 311
  otool, 308
LC_UUID, 106
LC_VERSION_MIN_IPHONEOS, 114
LC_VERSION_MIN_MACOSX, 114
LDAP, 38
ldd, 105, 114
LDFILELIST, 302
leaks(1), 177–178
ledger, 352
ledgers, Mach scheduling, 398–399
ledger_entry_info(), 399
ledger_info(), 399
ledger_template_info(), 399
Legacy PICs (XT-PICs), 270
Lemon, Jonathan, 555
Leopard, 7, 131
LibC, 139
LibC, OS X, memory, 174–175
libdispatch, 545
libgmalloc, 175–176
libKern, 728–732
libkern, 50, 307, 742–743
libraries
  ASLR, 122
  dynamic, 111–130
  ELF, 102, 502
    binaries, 15–16
  executables, 111
  iOS, 42–44
  launch-time loading, 111–121
  NeXTSTEP, 4
  OS X, 42–44
  runtime loading, 122–124
  shared cache, 121
/Library, 23
Library/, 25
/Library/Frameworks, 33
/Library/LaunchAgents, 229
~/Library/LaunchAgents, 229
/Library/LaunchDaemons, 229
libSystem, 115
libxpc.dylib, 254, 256
libXSLT, 44
libZ, 44
LICENSE, 717
Lightweight Volume Manager (LwVM), 574–575
lightweight_update_priority(), 430
links, 578–579, 639
  pre-linking, 713
_LINKEDIT(), 107
__LINKEDIT, 134
Linus Cross Reference (LXR), 305
Linux
  binfmt, 516
  kernel, 262–264, 303
  NetFilter, 698
  OOM, 139
Lion, 8
  application containers, 84–97
  asctl(1), 84
  ASLR, 131–132
  boot_args, 201–202
  boot.efi, 195, 204
  booting from disk image, 590–591
  compression, 612
  Core Storage, 200, 204
  diagnose, 86
  dscl(8), 67
  I/O Kit, 237–238
  kextstat(8), 714
  kSBXProfileNoNetwork, 83
  LaunchPad, 13
  ledgers, 398
  MAC, 64
  malloc_entropy, 130
  64-bit, 200
  stack_guard, 130
  XPC, 253–257
  zones, 542
lipo, 99
lipo(1), 101–102
listImages(), 122
listq, 452
lists, BSD processes, 507–508
LLB. *See* Low Level Bootloader
LLDB, debugging, 182
lo, 678
LoadCoreStorageConfiguration(), 200
LoadDrivers, 200
LOAD_FAILURE, 523
LOAD_FILE_PROTOCOL, 190

load_init_program(), 227, 326
LoadKernelCache, 200
load_machfile(), *522–525*
LoadRamDisk, 200–201
load_result(), *524–525*
load_segment(), 107, 492
Local Advanced Programmable Interrupt Controller (LAPIC), 316
/Local/Default, 65
LocateHandle, 189
locationd, 242
lock groups, 361
lock objects, Mach
  groups, 361–362
  lock sets, 366–367
  mutex, 362–363
  read-write, 363
  semaphores, 364–366
  spinlock, 364
lock sets, 366–367
lock_acquire, 366
Lockbot, 244
lockbundle, 248
Lockdownd, 245
lockdownd, 234, 245–246
lockdown.host_watcher, 245
lock_handoff, 367
lock_handoff_accept, 367
locking
  ARM, 788–790
  Intel, 787–788
lock_make_stable, 367
lock_release, 366
lock_set, 352
lock_set_create, 366
lock_set_destroy, 366
lock_set_t, 366
lock_try, 367
LOG_ALERT, 70
LOG_ERR, 70
logging
  jailbreaking, 71–72
  OS X, 69–72
LoginWindow, 18
LOG_KERN, 70
lookups, B-Tree catalog, 634–636
loops, I/O Kit, 740
loopattach(), 677
LoopbackFS, 598
Low Level Bootloader (LLB), 210, 211
LowPriorityIO, 236
.lproj, 29
ls(1), 578
lseek(), 512
lsof(1), 156, 180
lsregister, 31–32
LwVM. *See* Lightweight Volume Manager
LXR. *See* Linus Cross Reference

M

-m, Mach, 441
MAC. *See* Mandatory Access Control
Mac OS Classic, 3–5
MAC_CHECK, 560
mac_execve, 82
__mac_execve(), 520
MACF. *See* Mandatory Access Control Framework
Mach, 4, 45
  APIs, 79
  binding, 415
  BSD, 343, 501, 510–512
  design goals, 345–346
  design philosophy, 344
  EFI, 203
  eradication of, 15
  I/O Kit, 737
  iOS, 343
  IPC services, 234
  kernel, 303
    memory allocators, 473–480
  lock groups, 361
  lock objects
    groups, 361–362
    lock sets, 366–367
    mutex, 362–363
    read-write, 363
    semaphores, 364–366
    spinlock, 364
  -m, 441
  messages, 346–357
    complex, 347–348
    MIG, 351–357
    passing, 344
    ports, 349–351
    sending, 348–349
  microkernels, 264, 501
    XNU, 343
  OS X, 343
  osfmk/console, 334
  pagers, 447, 480–499
    policy management, 494–499
  physical memory management, 462–467
    Intel architecture, 465–467
  PID, 511–512
  ports, 251–253, 357–358
  POSIX, 343
  primitives, 343–388
    clock, 378–380
    IPC, 357–360
    machine primitives, 367–387
    privileged ports, 374–377
    processor, 380–384
    processor_set, 384–387
    scheduling, 389–408
    synchronization, 360–367
  read-write lock objects, 363

scheduling, 389–446
algorithms, 427–430
ASTs, 423–427
continuations, 416–418
dispatch table, 428–430
exceptions, 436–445
explicit preemption, 418–420
handoffs, 415–416
implicit preemption, 420–423
kernel, 406–407
ledgers, 398–399
preemption modes, 418–423
tasks, 395–398, 422–423
task APIs, 399–404
threads, 390–395
thread APIs, 404–408
thread creation, 407–408
timer interrupts, 431–436
subsystems, 352–353
system calls, 46–48
throttling, 412
trailers, 347
trap handlers, 287–291
UNIX, 534
UN*X, 389
UPL, 484–486
VM, 447–500
architecture, 447–462
XNU, 49
zones, 467–473
boot, 470–471
debugging, 473
garbage collection, 471–473
OS X, 470–471
mach, 307
MACH(1), 166
Mach Interface Generator (MIG), 236, 256, 343, 351–357, 734–735
mach_call_munger, 287–289
mach_call_munger_xx, 438
machdep, 56
machdep_call_table, 292
mach_exc, 352
<mach/exception_types.h>, 437
mach_header, 105
mach_host, 352
mach_host.h, 355
mach_host_self(), 374, 496
machine, 307
machine primitives, 367–387
machine_init, 316–317
machine_startup, 314
mach_init(), 234–236, 351
<mach/mach_host.h>, 355
mach_make_memory_entry(), 456
<mach/message.h>, 346
mach_msg(), 236, 349, 353, 442
mach_msg_context_trailer_t, 347
mach_msg_mac_trailer_t, 347
MACH_MSG_OOL_DESCRIPTOR, 347–348
MACH_MSG_OOL_PORTS_DESCRIPTOR, 347–348
MACH_MSG_OOL_VOLATILE_DESCRIPTOR, 347–348
mach_msg_overwrite, 348
mach_msg_overwrite_trap(), 359
MACH_MSG_PORT_DESCRIPTOR, 347–348
mach_msg_receive(), 359–360
mach_msg_receive_results(), 360
mach_msg_security_trailer_t, 347
mach_msg_seqno_trailer_t, 347
mach_msg_trailer_t, 347
mach_msg_trailer_type_t, 346
mach_msg_trap(), 349
Mach-O
ASLR, 131–132
binaries, 522–525
dynamic libraries, 111–130
executables, 98
segments and sections, 108
file types, 103
header flags, 104
heaps, 139–140
LC_CODE_SIGNATURE, 110
load commands, 106–111
loader, 44
memory, 138–143
NeXTSTEP, 102
otool(1), 105
process address space, 130–138
universal binaries, 102–105
VM, 140–143
<mach-o/arch.h>, 100
mach_port, 352
mach_port_name_t(), 61
mach_port_t, 357, 452
MACH_RCV_INTERRUPT, 348
MACH_RCV_LARGE, 348
MACH_RCV_MSG, 348
mach_msg(), 353
MACH_RCV_NOTIFY, 348
MACH_RCV_OVERWRITE, 348
MACH_RCV_TIMEOUT, 348
MACH_RCV_TOO_LARGE, 348
mach_reply_port, 48
MACH_SEND_ALWAYS, 349
MACH_SEND_CANCEL, 349
MACH_SEND_INTERRUPT, 349
MACH_SEND_MSG, 349
mach_msg(), 353
MACH_SEND_NOTIFY, 349
MACH_SEND_TIMEOUT, 349
MACH_SEND_TRAILER, 349
MachServices, 373
mach_sg_send(), 359
mach_task_self(), 400, 407, 453
mach_trap, 152
mach_trap_table, 290–291
mach_types, 352
mach_vm, 352

mach_vm_allocate(), 453
mach_vm_behavior_set(), 454
mach_vm_deallocate(), 453
mach_vm_inherit(), 454
mach_vm_machine_attribute(), 455
mach_vm_map(), 455
mach_vm_map_page_query(), 456
mach_vm_msync(), 454
mach_vm_page_info(), 456
mach_vm_page_query(), 456
mach_vm_protect(), 453
mach_vm_purgable_control(), 456
mach_vm_read(), 454
mach_vm_read_overwrite(), memcpy, 454
mach_vm_region(), 453
mach_vm_region_recurse(), 453
mach_vm_region_recurse, 458–462
mach_vm_remap(), 455
[mach]_vm_wire, 375
mach_vm_wire, 458
mach_vm_write(), 454
mach_zone_info(), 467
MacOS, 717
mac_policy_conf, 559
mac_policy_initmach(), 318
mac_policy_ops, 559
MAC_POLICY_SET, 559
mac_policy_unregister, 559
mac_vnode_check_signature, 560
macx_swapoff(), 499
macx_swapon(), 499
macx_triggers(), 499
madvise(), 454
magazine allocator, 139
main(), 18, 92, 93, 187
_main, 120
maintenance_continuation(), 428
malloc(), 125, 127–128, 453, 467, 541–544
_MALLOC, 479
malloc(3), 174–175
MallocCheckHeapEach, 174
MallocCheckHeapSleep/Abort, 174
MallocCheckHeapStart, 174
MallocCorruptionAbort, 175
MallocDoNotProtectPostlude, 175
MallocDoNotProtectPrelude, 175
malloc_entropy, 130
MallocErrorAbort, 175
MallocGuardEdges, 175
malloc_history(1), 178
__MALLOC_LARGE, 134
MallocLogFile, 174
malloc_printf, 128
MALLOC_PROTECT_BEFORE, 176
MallocScribble, 175
__MALLOC_SMALL, 134
MallocStackLogging, 175
MallocStackLoggingDirectory, 175
MallocStackLoggingNoCompact, 175
__MALLOC_TINY, 134
man, 307
Management Information Base (MIB), 56, 64
Mandatory Access Control (MAC), 55, 62–65, 558–560
  mac_policy_initmach(), 318
  sandboxing, 89
Mandatory Access Control Framework (MACF), 527
Map Record, 628
MapKit, 38
Master Boot Record (MBR), 568–570
maxmem, 331
MAXPRI_THROTTLE(4), 412
maxprot, 107
MBR. *See* Master Boot Record
mbuf, 661–667
mcache, 545
mdcheckschema, 20
mddiagnose, 20
mdfind, 20
mdimport, 20
MDL. *See* Memory Descriptor List
mdls, 20
mDNS. *See* multicast DNS
mDNSResponder, 243
mds, 75
mdutil, 20
media access protocols, 190
MediaFiles.pkg, 216
MediaPlayer, 38
MediaToolbox, 38
memcpy, 26, 454
memory
  EEPROM, 184
  EFI, 189
  EMMI, 480
  kernel, 198
    Mach, 473–480
  leaks, debugging, 176–178
  Mac OS Classic, 3–4
  Mach-O, 138–143
  management, 13
    BSD, 539–549
    I/O Kit, 769
  OOM, 139
  OS X LibC, 174–175
  physical
    Mach, 462–467
    VM, 448–449
  PROM, 184
  ROM, 184
  VM
    ARM, 447, 791
    arm_vm_init(), 311
    Intel, 791
    isolated, 130
    Mach, 447–500
    Mach-O, 140–143
    PE, 304
    physical memory plane, 448–449

POSIX, 458, 540–541
processes, 107–109
threads, 144
Memory Descriptor List (MDL), 485
memory objects, 447
memory pressure, 545
memory_object, 452
memory_object_control.defs, 481
memory_object_data_initialize(), 482
memory_object_data_reclaim(), 482
memory_object_data_request(), 482
memory_object_data_return(), 482, 496
memory_object_data_unlock(), 482
memory_object_deallocate(), 481
memory_object_default.defs, 481
memory_object.defs, 481
memory_object_init(), 481
memory_object_last_unmap(), 482
memory_object_map(), 482
memory_object_name.defs, 481
memory_object_reference(), 481
memory_object_synchronize(), 482
memory_object_t, 483, 596
memory_object_terminate(), 481
Memorystatus, 546–548
memq, 452
Message, 38
messages
facility, 70
Mach, 346–357
complex, 347–348
MIG, 351–357
ports, 349–351
sending, 348–349
passing, 264, 344
severity, 70
MessageUI, 38
metadata, 20
metadata, 609
MetaData Importer, 20
metadata zone, 620–621
MH_ALLOW_STACK_EXECUTION, 103
MH_BINDS_TO_WEAK, 103
MH_BUNDLE(8), 103
MH_CORE(4), 103
MH_DSYM(10), 103
MH_DYLIB(6), 103
MH_DYLINKER(7), 103
MH_EXECUTABLE(2), 103
MH_FORCEFLAT, 103
MH_FORCE_FLAT, 125
MH_KEXT_BUNDLE(11), 103
MH_NO_HEAP_EXECUTION, 103
MH_NOUNDEFS, 103
MH_OBJECT(1), 103
MH_PIE, 103
MH_SPLITSEGS, 103
MH_TWOLEVEL, 103
MH_WEAK_DEFINES, 103
MIB. *See* Management Information Base
microkernels, 264–265, 343, 501
MIG. *See* Mach Interface Generator
MIG, 302
mig(1), 353
MINCORE_*, 456
mincore(2), 456
MISC(20), 167
MJ_DIRECTORY_CONTROL, 74
mkext, 723–724
Mkext Cache, 199
ml_enable_initmach(), 318
ml_enable_interrupts(), 318
ml_functions, 296–297
ml_get_interrupts_enabled, 295
mlock(2), 458
ml_thrm_init(), 314
Mobileassetd, 245
MobileCoreServices, 38
MobileFileIntegrity, 244
mobile.installd, 245
mobil.installd.mount_helper, 245
model specific registers (MSRs), 279
modular architecture, 712–713
Mohave, 11
monolithic kernel, 262–264
mount entry, 592–595
Mountain Lion, 9, 84, 215, 398
ASLR, 548–549
CoreStorage, 575–576
mpo_proc_check_get_task, 562
mpo_proc_check_run_cs_invalid, 562
mpo_vnode_check_exec, 562
mpo_vnode_check_signature, 562
msgbuf, 331
msgh_remote_por, 359
msgh_size, 346
MSRs. *See* model specific registers
msync(2), 454, 458
multicast DNS (mDNS), 652
multitasking, 4, 11, 420–423
multithreading, 93, 786, 787
mutual exclusion (mutex), 360, 362–363
mvn, 48

N

-n, 59
name mangling, 740
named pipes, 584
namespaces
I/O Kit, 740
two-level, 125
native file systems, 579–580
Natural Language Processing, 8
ncmds, 104
NDIS. *See* Network Driver Interface Specification
NDR. *See* Network Data Representation

NDRV sockets, 653
net, 56
net_add_domain(), 674
net_del_domain(), 674
NetFilter, 698
NetFS, 38
net*/netinet*, 307
netsrc_init(), 655
/Network, 23
NETWORK(2), 166
Network Data Representation (NDR), 353
Network Driver Interface Specification (NDIS), 739
network driver sockets, 652–654
Network File System (NFS), 582–583
network protocols, layer III, 676–678
network stack, 649–708
  layer II interfaces, 678–686
  layer III network protocols, 676–678
  layer IV transport protocols, 668–669
  layer V sockets, 660–668
  packet filtering, 693–705
  QoS, 705–707
  receiving data, 686–690
  sending data, 690–693
  socket statistics, 658–660
  traffic shaping, 705–707
  user mode, 650–658
networking
  file systems, 582–583
  IPv4, 651–652
  IPv6, 654–655
/Network/Library/Networks, 33
NewsStand, 12
Newsstandkit, 38
nextCatalogID, 633
NeXTSTEP, 4, 24, 34, 102, 737
NFS. *See* Network File System
nfs, 307
.nib, 28–29
nm(1), 105
-no64exec, 330
nodeSize, 627
NONUI_APPLICATION, 423
NORMAL, 528
NorthStar, 11
-no_shared_cr3, 330
notifications, OS X, 78–79
notify(), 78–79, 352
notifyd(/usr/sbin), 243
notifyd(8), 79
<notify.h>, 79
notifyutil(1), 79
not_terminated, 59
novfscache, 331
NRQBM, 413
nsects, 107
NSGlobalDomain, 30
nstat_control_register(), 656
NSTAT_PROVIDER_CP, 659
NSTAT_PROVIDER_ROUTE, 659
NSTAT_PROVIDER_UDP, 659
NSZones, 139
NT File System (NTFS), 578, 581, 591, 624
NULL, 134
null, 254
numbers, system calls, 46
NVRAM, 191, 192–194, 336
nvram, 329
NXFindBestFatArch(), 100
NXGetLocalArchInfo(), 100

O

-o, 151, 612
-o union, 587
Objective-C, 45
  Cocoa, 34
  CoreServices, 75–76
  garbage collection, 545
  Info.plist, 26
  Java, 44
  Leopard, 7
  NeXTSTEP, 4
  XPC, 256
offset, 456
ofileflags, 601
OOM. *See* Out-Of-Memory
open(), 127
OpenAL, 38
OpenCL, 7, 215
OpenCL, 39
OpenDarwin, 10
OpenDirecotry, 39
OpenGL, 7
OpenGL, 39
OpenGLES, 39
OpenProtocol, 189
OpenSL, 7
OpenSSH, 13, 44
OpenSSL, 44
OS X
  ACLs, 578
  applications, 24–32
  architecture, 15–51, 518
  auditing, 59–62, 556–558
  boot.efi, 194–210
  BSD, 501
  bundles, 24
  CHUD, 155
  Code Signing in Depth, LC_CODE_SIGNATURE, 110
  defaults(1), 173
  device tree, 196–198
  disk image files, 589
  DSMOS, 491, 716–717
  DTrace, 148
  dynamic_pager(8), 498–499
  EFI, 185

entitlements, 97
evolution, 3–16
execsw, 516
file systems, 587–589
Finder, 247–248
HFS+, 617–618
forks, 611
frameworks, 32–43
FUSE, 598
future, 15–16
GUI, 215
hdiutil, 568–569
hostinfo(1), 369
installation, 214–219
Intel, 261
interfaces, 678–680
iOS, 12–15
merger, 16
IPv6, 654–655
-k, 149–150
kernel, versions, 14–15
kernelcache, 719
kextd(8), 728
-l, 409
LaunchDaemons, 241–253
LibC, memory, 174–175
libgmalloc, 175–176
libraries, 42–44
logging, 69–72
Mac OS Classic, 4–5
Mach, 343
scheduling exceptions, 444–445
zones, 470–471
machine_init, 316
Mountain Lion, 9
network stack, 649
non-Apple hardware, 10
notifications, 78–79
otool -L, 114
PF_NDRV, 652
POSIX, 45, 46
preemptive multitasking, 420–423
process information, 156–159
Rosetta installer, 102
security, 79–90
shared cache, 121
sleep, 328–329
snapshots, 159–170
system configuration, 67–69
universal binaries, 99
UNIX, 502
directories, 23
user and group management, 65–67
utun, 682
versions, 5–10
vstart, 310
XNU, 266
OSAKit, 38
OSArray, 742
osascript(1), 72
OSBoolean, 742
OSBundleAllowUserLoad, 718
OSBundleCompatibleVersion, 718
OSBundleLibraries, 718
OSBundleRequired, 718
OSCollection, 742
OSCollectionIterator, 742
OSData, 742
OSDeclareDefaultStructors, 741
OSDefineMetaClassAndStructures, 741
OSDictionary, 742
osfmk, 303, 307
osfmk/console, 334
osfmk/kern/ast.h, 423
osfmk/kern/ledger.c, 398
osfmk/kern/sched.h, 409, 412–413
osfmk/kern/task.h, 395–397
osfmk/kern/timer_call_entry.h, 433
osfmk/kern/wait_queue.h, 414
osfmk/kern/zalloc.h, 469
osfmk/mach/host_priv.h, 457
osfmk/man, 345
osfmk/memory_object.types.h, 483
osfmk/thread/thread.c, 395
osfmk/vm/vm_user.c, 458
OSInstaller, 216–217
OSInstall.mkpg, 216
OSInstall.pkg, 216
OSIterator, 742
OSKext*, 728, 742
OSMalloc, 479–480
OSMetaClass, 741, 742
OSNumber, 742
OSObject, 739, 741, 742
OSORderedSet, 742
OSSet, 742
OSString, 742
OSSymbol, 742
otool, 308
otool(1), 105, 340
otool -L, 114
otool -l, 117
Out-Of-Memory (OOM), 139

P

-p, 467
p_acflag, 514
PackageInfo, 217
PackageKit, 217
packet filtering
BPF, 701–705
interface filters, 701
IP filters, 698–701
ipfw(8), 696–697
network stack, 693–705
PF, 697–698

socket filters, 694–696
page management system calls, 540–541
page table entries (PTEs), 449
#defines, 463
pageout, 495–497
pageout daemon, 448
pagers
Apple protect, 491–493
Mach, 447, 480–499
policy management, 494–499
swap files, 488
XNU, 486
pagestuff(1), 126–127
__PAGEZERO(), 107, 133–134
panic(), 333–340
_panicd_corename, 332
panic_dialog.c, 334
_panicd_ip, 332
_panicd_port, 332
panic_image.c, 334
panic-info, 193
panic_ui/genimage.c, 334
panic_ui/qtif2raw.c, 334
panic_ui/setupdialog.c, 334
Panther, 6
Parent Process Identifier (PPID), 91–92
parent processes, 91–92
parentID, 633, 635
parent_proc, 516
parse_machfile, 523–524
Partition Boot Record, 568
partitions, 565–577
CoreStorage, 575–577
disks, APM, 570–572
GPT, 572–574
LwVM, 574–575
MBR, 568–570
PASSIVE, 528
passwords, 67
Pastboard(), 245
Payload, 217
PCSC, 39
pdp_ip, 679
PE. *See* Platform Expert
PE_i_can_has_debugger, 562
PE_init_platform, 304
PE_parse_boot_argn, 314, 331, 562
permissions, 262, 577–578, 637–639
PersistentURLTranslator.Gatekeeper, 244
personality, 755
PEs. *See* Portable Executables
PESavePanicInfo(), 336
PE_State, 202
PE_state, 304
_PE_state, 303
PE_Video, 304
pexpert, 303, 307
PF. *See* Protocol Family
PFDL. *See* process file descriptor lock
PF_INET, 650, 677
PF_INET6, 650
PF_KEY, 650
PF_LAT, 650
PF_LOCAL, 650
pflog, 678
pflog_clone_create(), 685
PF_NDRV, 650, 651, 652
spoofing packets, 653–654
PF_PACKET, 653
PF_PPP, 651
PF_ROUTE, 650, 652
PF_SYSTEM, 79, 650, 651, 682
system sockets, 655–666
PF_SYSTEM/SYSPROTO_EVENT, 657–658
PFZ. *See* Preemption Free Zone
pg_members, 508
pgrp_iterate(), 508
physical memory
Mach, 462–467
VM, 448–449
PIC. *See* Programmable Interrupt Controller
PID. *See* Process ID
PIDEX(14), 170
pid_resume(), 94, 494
pid_shutdown_sockets, 94
pid_suspend(), 94, 494
PIDTR(11), 169
pinsertchild(), 516
PIPE(), 534
pipeops, 605
.pkg, 217
PL. *See* process lock
<platform>, 313
Platform Expert (PE), 296, 303, 304
plist, 162, 718
.plist, 229
p_listflag, 516
P_LIST_INCREATE, 516
plumbing, 677
plutil(), 28
pmap, 448–449, 463, 464–465
pmap_create(), 464
pmap_destroy(), 464
pmap_disconnect(), 465
PMAP_ENTER(), 498
pmap_enter(), 464
pmap_enter[_options](), 464
pmap_page_protect(), 464
pmap_reference(), 464
pmap_remove(), 465
pmap_switch(), 465
pmap_t, 463, 465–467
pmap_zero_page(), 464
pmc/profiling, 307
pmCPUGetDeadline(), 433
pmset(1), 68
PNG, 204
Point-to-Point Protocol (PPP), 651

policies
  Apple policy modules, 560–563
  execution, 527–528
  I/O, 527–528
  MAC, 559–560
  Mach pagers, 494–499
policy_check, 331
poll(2), 144
Portable Executables (PEs), 187
portmapper, RPC, UNIX, 234–235
ports, 234
  exceptions, 436, 439
  Mach, 251–253, 357–358
    messages, 349–351
  tasks, 402
PORT_SET, 350
POSIX
  BSD, 501, 503
    system calls, 284–287
  FUSE, 598
  Leopard, 7
  Mach, 343
  network stack, 649
  OS X, 45, 46
  page management system calls, 540–541
  semaphores, 364
  system calls, 46, 283
  threads, 144–145
  VFS, 591
  VM, 458, 540–541
posix_spawn(), 91, 132, 513, 514, 515
PostScript, 4
power management, 751–753
Power On Self Test, 184
PowerPC, 183
PPC, 296–297, 518–519
PPID. *See* Parent Process Identifier
PPP. *See* Point-to-Point Protocol
ppp, 679
praudit(1), 60, 556
pr_ctlinput(), 671
pr_ctloutput(), 671
pr_drain(), 671
PRECEDENCE_POLICY, 421
Preemption Free Zone (PFZ), 275, 426–427
preemption modes, Mach scheduling, 418–423
  explicit, 418–420
  implicit, 420–423
preemptive multitasking, OS X, 420–423
prefabt, 426
PreferencePanes, 39
_PrelinkBundlePath, 722
_PrelinkExecutable*, 722
PRELINK_INFO, 109
__PRELINK_INFO, 721–722
pre-linking, 713
_PrelinkInterfaceUUID, 722
pr_fasttimo(), 671
pr_init(), 671, 674
pr_input(), 671
printf(), 117, 128, 131, 313
private frameworks, 33
privileged ports, 374–377
pr_lock(), 672
probes, 147
proc, 152
PROC_ALLPROCLIST, 508
PROC_CREATE_FORK, 514, 516
PROC_CREATE_SPAWN, 514, 516
PROC_CREATE_VFORK, 514
Procedure, 353
proc_enforce, 64
processes, 91–146
  BSD, 504–508
    control and tracing, 525–529
    creating, 512–525
    lists, 507–508
    software, 535
    structs, 504–507
    suspension and resumption, 529
  CPU, 92–93
  executables, 98
  groups, 91
    BSD, 507–508
  hibernation, iOS, 547–548
  information, OS X, 156–159
  instances, 91
  I/O, 93, 600–605
  lifecycle, 92–95
    pid_resume, 94
    pid_suspend, 94
    zombie state, 93–94
  security, 97
  threads, 91–92
  universal binaries, 99–111
  UNIX, 91
    signals, 95–97
  VM, 107–109
process address space, Mach-O, 130–138
process file descriptor lock (PFDL), 507
Process ID (PID), 91, 93, 228, 326, 515
  bsdinit_task(), 325
  dtruss, 150
  killpg1_callback(), 535
  Mach, 511–512
process lock (PL), 507
process spin lock (PSL), 507
ProcessOptions, 198–199
processor, 352, 380–384
processor_assign, 381
processor_control, 381
processor_csw_check(), 429
processor_enqueue(), 429
processor_exit, 381
processor_get_assignment, 381
processor_info, 381
processor_init(), 428
PROCESSOR_NULL, 415

processor_queue_empty(), 429
processor_queue_has_priority(), 429
processor_queue_remove(), 429
processor_queue_shutdown(), 429
processor_queue_urgent(), 429
processor_runq(), 430
processor_runq_stats_count_sum(), 430
processor_set, 352, 384–387, 408
processor_set_destroy, 385
processor_set_info, 386
processor_set_max_priority, 385
processor_set_policy_control, 386
processor_set_policy_enable, 385
processor_set_stack_usage, 386
processor_set_statistics, 385
processor_set_tasks, 385
processor_set_threads, 385
processor_start, 381
processor_ts, 382–384
process_policy(), 528
Procfs, 598
proc_info, 156–159, 527, 552
proc_iterate(), 508
proc_listallpids, 159
proc_listchildpids, 159
proc_listpgrppids, 159
PROC_PIDWORKQUEUEINFO, 552
PROC_POLICY_APP_LIFECYCLE, 528
PROC_POLICY_APPTYPE, 528
PROC_POLICY_BACKGROUND, 528
PROC_POLICY_HARDWARE_ACCESS, 528
PROC_POLICY_RESOURCE_STARVATION, 528
PROC_POLICY_RESOURCE_USAGE, 528
proc_t, 326, 515, 600
PROC_ZOMPROCLIST, 508
profile, 152
Program, 238
ProgramArguments, 238
Programmable Interrupt Controller (PIC), 270
Programmable Read Only Memory (PROM), 184
protocols. *See also specific protocols*
  EFI, 188–191
  GUIDs, UEFI, 191
  interfaces, 677–678
  KPI functions, 677
  transport, layer IV, 668–669
Protocol Family (PF), 650
  packet filtering, 697–698
proto_plumb(), 677
ProtoString(), 427
protosws, 669–673
prototypes, 46
pr_output(), 671
pr_slowtimo(), 671
pr_sysctl(), 672
pr_unlock(), 672
pru_sosend, 691
pr_usrreq(), 672–673
ps(1), 179, 409–411
pset_init(), 428
pset_name_self, 384
psets, 408
pseudo file systems, 583–587
PSL. *See* process spin lock
PTEs. *See* page table entries
Pthread, 49
pthread, 144–145
pthread_create(), 407, 510
pthread_exit(), 408
pthread_mutex_lock(), 134
ptrace(2), 148, 525–527
PubSub, 39
Puma, 6
PureDarwin, 10
purgeable zones, 139
PurpleSystemEventPort, 253
PUSH_FUNCTION, 272
p_uthlist, 515
puts, 117
Pystar, 10
Python, 7
Python, 39

Q

.qlgenerator, 18
qlmanage(), 19
QoS. *See* Quality of Service
QT(32), 167
QTKit, 39
Quality of Service (QoS), 705–707
quantum_expire(), 430
quarantine, 609
Quartz, 39
Quartz Extreme, 6
QuartzCore, 39
QueueDirectories, 237
queue_head.t, 398
queue-iterate, 398
QuickLook, 18–19
QuickLook, 39
QuickLookGeneratorPluginFactory, 18
QuickTime, 39

R

Racoon, 243
RaiseTPL, 189
RAM Disk, 199
RAMDisk, 200–201
random access, 624
RAX, 278
RB_SINGLE, 326
read(), 418
read(2), 143
readelf, 105

Read-Only Memory (ROM), 184
READTR(10), 169
read-write lock objects, 363
ready_heap, 706
real GID, 97
real UID, 97
realtime_setrun, 407
RECEIVE, 349
recovery mode, iBoot, 212–213
ref_count, 456
rEFIT, 194
registers
  ARM, 776–779
  CPSR, 267–268, 777–778
  CRs, 266–267, 775–776, 778–779
  DRs, 775
  floating point, 774, 777
  Intel, 773–776
  MSRs, 279
RegisterProtocolNotify, 189
regular expressions, 306
ReinstallProtocolInterface, 189
relpath, 300
Remote Procedure Call (RPC), 351
  portmapper, UNIX, 234–235
REMOVE(7), 169
removeDisk, 576
Rendezvous, 6
RENICED, 422
replay attacks, 213–214
_reply_sync, 256
ReportCrash, 243
reservation specification (RSpec), 706
ResetSystem, 192
ResizeDisk, 576
resizeStack, 576
ResizeVolume, 576
resource forks, 611–612
Resources, 28
RestoreTPL, 189
Return-Oriented Programming (ROP), 132
reverse DNS, 18–19, 30
Revision, 202
RFLAGS, 774–775
Rhapsody, 5
rings, 266–267
RLIMIT_CORE, 170
robustness, 265
ROM. *See* Read-Only Memory
Root UUID, 199
ROP. *See* Return-Oriented Programming
Rosetta installer, 102
route(8), 652
_router_ip, 332
Routine, 353
routing sockets, 652
RPC. *See* Remote Procedure Call
rpcgen, 351
RSpec. *See* reservation specification
rtclock, 431
rtclock_timer.deadline, 432–433
rtclock_timer_t, 432
rtc_timer, 435
Ruby, 7
Ruby, 39
RubyCocoa, 39
run queues, 412–413
RunLoopType(), 257
runtime services, 191–192
RunTimeServices, initializeConsole, 195

S

-S, 143
-s, 151, 228, 326, 330
Safari, 6
Saffron, 12
sample(1), 174
Sandboxd, 243
sandboxd, 243
SandBoxedFetch, 257
sandboxing, 65, 81–90
  controlling, 82–83
  enforcing, 89–90
  entitlements, 83–89
  iOS, 81–82
  jailbreaking, 81–82
  voluntary imprisonment, 82
sandbox_init(3), 82
Sandbox.kext, 561
_SandboxProfile, 257
SandboxProfileData, 86
SandboxProfileDataValidation
    EntitlementsKey, 86
Saved Application State, 85
sbappend(), 690
sbappendaddr(), 690
sbappendrecord(), 690
sbappendstream(), 690
SBAppTags, 248
/sbin, 22
/sbin/launchd, 227
scalable allocator, 139
SCDyamicStore, 69
SceneKit, 39
sched, 152
sched_decay_shifts, 411–412
sched_dispatch_table, 428
sched_pri, 413
sched_prim.h, 428
sched_pri_shift, 411
scheduling
  kernel, 262, 406–407
  Mach, 389–446
      algorithms, 427–430
      ASTs, 423–427
      continuations, 416–418

dispatch table, 428–430
exceptions, 436–445
explicit preemption, 418–420
handoffs, 415–416
implicit preemption, 420–423
kernel, 406–407
ledgers, 398–399
preemption modes, 418–423
primitives, 389–408
tasks, 395–398, 422–423
task APIs, 399–404
threads, 390–395
thread APIs, 404–408
thread creation, 407–408
timer interrupts, 431–436
SCNetworkReachability, 69
SCNetworkReachabilityConfigd, 242
ScreenSaver, 39
Scripting, 39
ScriptingBridge, 39
Scripts, 217
sc_usage(1), 165
scutil(8), 67–68, 69
search, B-Tree, 624, 629–630
SECURE_KERNEL, 305
security
iOS, 79–90
kernel, 262
kext, 718
Lion, 8
OS X, 79–90
processes, 97
Security, 39
security, 307, 352
_security(), 553
security(1), 80
SECURITY(9), 167
Securityd, 243
securityd, 243
SecurityFoundation, 39
SecurityInterface, 39
SecurityServer (SL), 243
-segcreate, 109
segedit(1), 105, 721
segname, 107
select(), 418
select(2), 144
self-contained*_init(), 320
semaphores, 364–366
Mach lock objects, 364–366
POSIX, 364
semaphore_create, 365
semaphore_destroy, 365
semaphore_signal, 365
semaphore_signal_all, 365
semaphore_wait, 365
SEND, 349
SEND_ONCE, 350
serial, 313, 318, 331, 332
SERIAL_KDP, 318
ServerNotification, 39
serverperfmode, 331
servicebundle, 248
ServiceManagement, 40
ServiceType, 257
set_alarm, 380
setaudit(), 61
setaudit_addr(), 61
SETBUF(4), 169
SetConsoleMode, 200
set_dp_control_port, 376
setfsgid, 97
setfsuid, 97
setpgrp(2), 91
setPop(), 435
SETREG(8), 169
setrlimit(2), 170, 398, 515
SETRTCDEC(15), 170
SetTime, 192
SetTimer, 189
SETUP(6), 169
Setup.App, 249
setup_wqthread, 551
SetVariable, 192
SetWakeupTime, 192
severity, 70
sflt_detach(), 695
SFLT_GLOBAL, 696
sflt_register(), 694
sflt_unregister(), 694
sftl_attach(), 695
SG_PROTECTED_VERSION, 492
shared library cache, 121
shells, 246–253
shmem, 255
should_current_thread_rechoose_processor(), 430
show regions, 458
SHSH, 213–214
SIDL, 92
signals
BSD, 529–536
UNIX, processes, 95–97
SignalEvent, 189
Simple Network Management Protocol (SNMP), 56
SIMPLE_FILE_SYSTEM_PROTOCOL, 190
SIMPLE_POINTER_PROTOCOL, 190
Simpleprocedure, 353
Simpleroutine, 353
SIMPLE_TEXT_INPUT_PROTOCOL, 190
SIMPLE_TEXT_OUTPUT_PROTOCOL, 190
single UNIX specification (SUS), 502
Siri, 12
SIUResources.pkg, 216
64-bit
BIOS, 184
kernel, 264
Lion, 8, 200

memory leaks, 176
process address space, 132–133
Snow Leopard, 7
XNU, system calls, 283–284
size, 346
size(1), 105, 109
sizeof(void *), 286
sizeofncmds, 104
slab allocators, 545
slave_pstart(), 313, 316, 329
sleep, 328–329
sleep, 418
sleep_kernel(), 329
sleh_abort, 438
sleh_undef, 438
SMP, 316, 319, 360, 415
smp_init, 316–317
snapshots, 159–170
SNMP. *See* Simple Network Management Protocol
Snow Leopard, 7–8, 99, 130, 139, 561
.so, 42
sockaddr, 691
sockets
descriptors
launchd, 240
layer V sockets, 660–661
domains, UNIX, 651
filters
packet filtering, 694–696
XNU, 695–696
kernel mode, 667–668
layer V, 660–668
NDRV, 653
network driver, 652–654
routing, 652
statistics, 658–660
system, 556, 655–658
Sockets, 238
socket_t, 696
sock_inject_*, 695
sockkets, IPSec Key Management, 654
SOCK_RAW, 653
soft links, 578–579, 639
SoftResourceLimits, 236
SoftWare Interrupt (SWI), 275, 280
Solaris, 149
so_proto, 667
source-level compatibility, 502
specfs, 586
Spin Control, 174
spindump, 174
spinlock, Mach lock objects, 364
spllo(), 318
spoofing packets, 653–654
Spotlight, 6, 19–20, 75
SpotlightFS, 598
SpringBoard, 13, 248–253, 411
Springboard(), 245
SRUN, 93
SSH, 13–14, 21, 598
ssh.plist, 232–233
SSLEEP, 94
stack protector, 130
stack_collect(), 497
stack_guard, 130
stackshot(1), 160–162
stack_snapshot, 162–165
STANDARD_POLICY, 421
starblock, 639
start(), 310–311
start-stf, 655
start_time.stop_time, 59
stderr, 232, 238, 241
stdin, 232, 238, 241
<stdlib.h>, 503, 724
stdout, 232, 238, 241
std_types, 352
steal_thread(), 429
stf, 678
stf(4), 655
stfattach(), 685
STOP, 94
StopAnimation, 201
StoreKit, 40
strace, 150
string, 254
<string.h>, 503
strings(1), 105
stroff, 115
struct, 201, 463
structs, BSD processes, 504–507
struct fuse_operations, 598
struct ifnet, 680–681
struct mbuf, 661
struct mount, 592–593
struct proc, 504–507
struct proclist, 507–508
struct sockbuf, 661
struct uthread, 508–510
struct vnode, 595–597
stub_helper, 118
__stubs, 115
subsystems
I/O Kit, 753
Mach, 352–353
sunrpc, 235
SUN-RPC, 351, 353
superblock, 592
SuperVisor Call (SVC), 275
supports_timeshare(), 429
SUS. *See* single UNIX specification
SVC. *See* SuperVisor Call
SVC, 267
swap files, 488
swapfile_pager_data_request(), 488–491
SWI. *See* SoftWare Interrupt

switch(), 272, 333
symoff, 115
synchronous interrupt, 278
synchronous kernel, 268
SyncServices, 40
SYS, 267
SYS(), 534
sys, 307
SYSCALL, 279–282
syscall, 152, 169
sysctl(), 56–57, 156
SYSCTL_*, 553, 554
sysctl(2), 169, 620, 646–647
sysctl(8), 110, 142, 171, 552–555
sysdiagnose(1), 159–160
<sys/disk.h>, 566–567
sysent, 285–287
SYSENTER, 279–282
sysenter, 280
sys/kern_control.h, 656
syslog, 70
syslogd, 71, 72, 243
sys/malloc.h, 542
<sys/proc.h>, 92
SYSPROTO_EVENT, 657
<sys/signal.h>, 95
<sys/socket.h>, 650
<sys/syscall.h>, 94
System, 40
/System, 23
system calls
  BSD, 47–48
    POSIX, 284–287
  diagnostic, 292–295
  kernel, 261, 268, 283–295
    iOS, 286–287
  MAC, 63–64
  Mach, 46–48
  numbers, 46
  POSIX, 46, 283
    BSD, 284–287
  prototypes, 46
  UNIX, 292
  XNU 64-bit, 283–284
system sockets, 556, 655–658
system sockets, 79
SystemAudioVolume, 193
SystemConfiguration, 40
SystemConfiguration.framework, 68
/System/Library/CoreServices, 247
/System/Library/Frameworks, 33
/System/Library/LaunchAgents, 229
/System/Library/LaunchDaemons, 229
/System/Library/Sandbox/Profiles, 83
system.logger, 243
system.notification_center, 243
system_profiler(8), 159
system.Security, 609
SystemUIServer, 247

T

tar(1), 217
target_task, 455
task, 352
tasks
  Mach scheduling, 395–398, 422–423
    APIs, 399–404
  multitasking, 4, 11, 420–423
  ports, 402
  threads, 397
task_access, 353
task_create(), 400
task_for_allow, 444
task_for_pid(), 462, 511
task_get_exception_ports(), 401
task_get_state(), 401
task_importance(), 401
task_info(), 400
task_policy_get(), 401
task_policy_set(), 401
task_priority(), 397–398, 401
task_resume(), 400, 529
task_sample(), 401
task_set_emulation(), 345
task_set_exception_ports(), 401
task_set_info(), 400
task_suspend(), 400, 529
task_terminate(), 400
task_threads(), 400, 405
task_zone_info(), 467
Tcl, 40
TC-shell, 21
Telluride, 12
Terminal, 20
Terminal.app, 231
_TEXT(), 107
__TEXT, 134
TextEdit, 84–87
32-bit
  Intel, process address space, 132
  iOS, process address space, 133–134
  kernel, 266
  memory leaks, 176
threads, 143–146
  BSD, 508–512
  CPU, 408
    affinity, 415
  execution, 408
  hyperthreading, 408, 415
  Mach scheduling, 390–395
    APIs, 404–408
    creation, 407–408
  multithreading, 93, 786, 787
  objects, BSD, 508–510

POSIX, 144–145
priorities, 409–412
processes, 91–92
run queues, 412–413
tasks, 397
UNIX, 143
VM, 144
vm_pageout(), 495
wait queues, 414
XNU, 512
thread_abort[_safely](), 404
thread_act, 353
[thread/act]_[get/set]_state, 404
THREAD_AFFINITY_POLICY, 422
thread_assign(), 405
thread_assign_default(), 405
thread_ast_set(), 423
THREAD_BACKGROUND_POLICY, 422
THREAD_BASIC_INFO, 405
thread_bind, 406
thread_block(), 416
thread_block_parameter(), 406, 419
thread_block_reason(), 406, 418–419
thread_bootstrap(), 395
thread_bootstrap_return(), 417
thread_call_daemon, 469
thread_count, 397
thread_create(), 395, 407
thread_create_running(), 407
thread_depress_abort(), 404
thread_exception_return(), 417
THREAD_EXTENDED_POLICY, 422
thread_get_assignment(), 405
thread_get_exception_ports(), 405
thread_[get/set]_special port(), 405
thread_go, 407, 414
thread_info(), 405
thread_invoke(), 406, 419
thread_policy, 405
thread_policy_[get/set](), 405
thread_policy_set_internal(), 421
THREAD_PRECEDENCE_POLICY, 422
thread_resume(), 325–326, 404
thread_run, 406
thread_sample, 405
thread_set_exception_ports, 405, 436
thread_set_policy, 405
thread_setrun, 407, 414
thread_set_state, 408
THREAD_STANDARD_POLICY, 422
thread_suspend(), 404
thread_swap_exception_ports, 405
thread_switch(), 415–416
thread_t, 419
thread_t mach_thread(), 404
thread_template, 395
thread_terminate(), 404
thread_terminate, 408
THREAD_TIME_CONSTRAINT_POLICY, 422
thread_unblock, 414
thread_wakeup_prim, 406
THRMAP(12), 169
THROTTLE, 528
THROTTLE_APPLICATION, 423
throttling
launchd, 236–237
Mach, 412
thumb mode, 785–786
tick-less kernel, 432
Tiger, 6–7
TIME_ABSOLUTE, 378
timebase_init(), 428
TIME_CONSTRAINT_POLICY, 421
TimeOut, 59
timer interrupts, 431–436
TIMER_CALL_CRITICAL, 433
timer_call_enter, 433
TIME_RELATIVE, 378
timer_queue_expire, 434
timestamps, 578, 607–608
TinySCHEME, 82
TinyUmbrella, 214
Tk, 40
TLB. *See* Translation Lookaside Buffer
Tmp, 25
/tmp, 22, 25
top(1), 179–180
TOSTOP, 93
totalNodes, 628
tr(1), 409
TRACE(7), 167
Trace Server, 162
tracers, 147
TraditionalString(), 427
TraditionalWithPsetRun
QueueString(), 427
traffic shaping, 705–707
transactions
HFS+ journaling, 644–645
launchd, 236
Translation Lookaside Buffer (TLB), 144, 449
transport protocols, layer IV, 668–669
TRAP, 272, 274, 534
trap handlers
Intel, 268–278
ARM, 275–278
kernel, 334
Mach, 287–291
treeDepth, 627
true, 254
truss, 150
Trusted BSD, 62
TSTOP, 93
tunneling, 682–686
TWAIN, 40
Twitter, 40
twitter.authenticate, 245
Twitterd, 245
twittered.server, 245
two-level namespace, 125

U

-u, 441
-u mobile, 246
ubc_info, 596
ubc_info_init(), 488
UDF. *See* Universal Disk Format
UDIF. *See* Universal Disk Image Format
-udp_in 1, 70
udp_output(), 691
udp_send(), 691
UEFI. *See* Universal Extensible Firmware Interface
UGA_DRAW_PROTOCOL, 190
UID. *See* user identifier
UIKit, 40
UIKit.pasteboardd, 245
uint64, 254
ulimit(1), 512, 515
ulimit -c, 170–171
uname(1), 9, 14
UND, 268
undef, 426
Unicode, 617
Unified Buffer Cache, 484, 488, 596
Uniform Type Identifier (UTI), 18
UninstallProtocolInterface, 189
unionfs, 587
<unistd.h>, 46, 503, 724
universal binaries
  executables, 98
  file(1), 99
  kernel, 100
  Mach-O, 102–105
  OS X, 99
  processes, 99–111
  Snow Leopard, 99
  Tiger, 6
Universal Disk Format (UDF), 582, 591
Universal Disk Image Format (UDIF), 589
Universal Extensible Firmware Interface (UEFI), 185–186, 191
Universal Page List (UPL), 484–486
Universal Plug and Play (uPNP), 6
UNIX. *See also* X is Not UNIX
  BSD, 501–502
  Darwin, 5, 20–22
  debugging, 178–180
  directories, 22–24
    iOS, 23–24
    OS X, 23
  domain sockets, 651
  exceptions, 529–534
  executables, 98
  fork(), 512
  FUSE, 598
  INET, 234
  inetd, 238
  inode, 608
  Leopard, 7
  load_init_program(), 326
  Mach, 534
  OS X, 502
  permissions, 577, 639
  processes, 91
  RPC portmapper, 234–235
  signals, processes, 95–97
  system calls, 292
  threads, 143
  -u, 441
unix_syscall, 284–285
unpackers, cache, 121
unprotect_segment(), 492, 493
UNSPECIFIED4, 422
UN*X
  atd, 231
  crond, 231
  inetd, 232–234
  launchd, 229
  ldd, 114
  Mach, 389
  SUN-RPC, 351
  xinetd, 232–234
update_priority(), 411, 430
UPL. *See* Universal Page List
upl_abort[range](), 486
upl_clear_dirty(), 486
upl_create(), 485
upl_deallocate(), 486
uPNP. *See* Universal Plug and Play
user, 56
User Data Record, 628
User Experience layer, 15, 17–20
user identifier (UID), 97
user mode
  BSD process creation, 512–513
  involuntary transition
    exceptions, 269–270
    interrupts, 270–271
  I/O Kit, 740, 746–755
    device drivers, 749–750
    I/O registry, 747–749
    plug and play, 750–751
  network stack, 650–658
  traffic shaping, 707
  voluntary transition, 278–282
UserNotification, 307
/Users, 23
USER_TRAP, 272
user_trap(), 274, 438
user_trap_returns, 425
USR, 267
/usr, 22
/usr/share/sandbox, 83
utaskbootstrap(), 326
uthread, 510
UTI. *See* Uniform Type Identifier
utun, 679, 682–686
utun_control_register(), 655
utun_ctl_connect(), 684–685
uuid, 255

ux_handler(), 529–532
ux_handler_init(), 326, 529–530

V

-v, 313
/var, 22
/var/audit, 60
/var/log/asl, 70
/var/log/install.log, 214
/var/run/lockdown.sock, 234
/var/tmp/launchd-shutdown.log, 228
Vassetd, 245
vecLib, 40
--verify, 86
Version, 202
version.plist, 717
vfork(), 514, 515
VFS. *See* Virtual FileSystem Switch
vfs, 56, 307
VFS_CTL_QUERY, 647
vfs_fentry, 591–592, 593
vfs_fsadd(), 593
vfs_mountroot(), 592
VideoDecodeAcceleration, 40
VideoToolKit, 40
Virtual FileSystem Switch (VFS), 22, 577, 591–600
  fsctl(2), 645–646
  FUSE, 597–605
  kernel, 645–648
  mount entry, 592–595
  struct vnode, 595–597
  sysctl(2), 646–647
  vnode, 595–597
virtual memory (VM)
  ARM, 447, 791
  arm_vm_init(), 311
  Intel, 791
  isolated, 130
  Mach, 447–500
    architecture, 447–462
  Mach-O, 140–143
  PE, 304
  physical memory plane, 448–449
  POSIX, 458, 540–541
  processes, 107–109
  threads, 144
virtualization, 10, 262, 267
vlan, 679
VM. *See* virtual memory
vm, 56, 307
vmaddr, 107
vm_allocate, 453
vm_allocate_cpm, 375
VM_BASIC_INFO_64, 453
VM_CHECK_MEMORYSTATUS, 548
vm_check_memorystatus, 548
vm_fault(), 498
VM_FLAGS_ANWHERE, 453
vminfo, 152
VM_INHERIT_COPY, 455
VM_INHERIT_SHARE, 455
vmmap(), 135–138
vm_map(), 353, 448, 450–451, 456, 493
VM_MAP_ANWHERE, 455
vm_map_apple_protected(), 493
vm_map_behavior_set, 454
vm_map_copyin(), 454
vm_map_copyout(), 454
vm_map_copy_overwrite, 454
vm_map_enter(), 453, 457
vm_map_entry(), 448, 451–452
vm_map_inherit(), 454
vm_map_lookup_entry(), 453
vm_map_machine_attribute(), 455
vm_map_msync, 454
vm_map_object, 452
VM_MAP_OVERWRITE, 455
vm_map_page_query_internal(), 456
vm_map_protect(), 453, 457
vm_map_remap(), 455
vm_map_t, 452
VM_MEM_SUPERPAGE, 465
VM_NOT_CACHEABLE, 465
vm_object(), 448
vm_object_t, 452
vm_page(), 448, 452
vm_page_info(), 456
VM_PAGE_INFO_BASIC, 456
vm_pageout(), 319, 495, 496, 497
vm_pageout_garbage_collect, 471–473
VM_PAGE_QUERY_PAGE_*, 456
vm_page_queue_active, 495
vm_page_queue_free, 495
vm_page_queue_inactive, 495
vm_page_queue_speculative, 495
VM_PRESSURE_MINIMUM_RSIZE, 545
vm_pressure_monitor(), 545
VM_PROT_EXECUTE, 455
VM_PROT_READ, 455
VM_PROT_WRITE, 455
vm_rdwr, 521
vm_read_overwrite, 454
VM_REGION_BASIC_INFO, 458–462
vmsize, 107
vm_stat(1), 141–142, 495–496
vm_statistics, 495–497
VMWare, 10, 333
vnmap(1), 458–462
vnode, 488, 584–587, 595–597
vnode_enforce, 64
vnode_pager, 448
VNOP_LOOKUP, 597
void, 361, 464–465
/Volume, 23, 24
volume header, HFS+, 631–632
voluntary user/kernel transition, 278–282

<vproc.h>, 236
vpro_transaction, 236
vstart(), 279, 306, 310

W

wait(), 93
wait(2), 93
wait queues, 414
wait3(2), 93
wait4(2), 93
WaitForEvent, 189
waitpid(2), 93
wait_queue_assert_wait[64[_locked]], 414
wait_queue_t, 365
wait_result_t, 363, 364
WatchPaths, 237
weakly defined symbols, 124
Web Distributed Authoring and Versioning (WebDAV), 583
WebKit, 40
wfq_ready_heap, 706
widgets, 6, 45, 47
WildCat, 11
WindowServer, 17
work queues, 550–552
wpkernel, 331
WQOPS_QUEUE_ADD, 550
WQOPS_THREAD_RETURN, 551
WQOPS_THREAD_SETCONC, 550
wq_runitem, 551
wrappers, 122, 149, 240–241
write(2), 144
WriteProcessMemoryEx(), 407

X

-x, 330
X is Not UNIX (XNU), 5
  boot
    arguments, 329–331
    kernel, 299–340
  BSD, 49–50, 501, 504
  build actions, 302
  Cheetah, 6
  CHUD, 155
  compiling, 300–302
  CONFIG_CODE_DECRYPTION, 493
  CONFIG_DEBUG, 308
  configuration, 305
  CONFIG_ZLEAKS, 468
  DEBUG, 308
  domains, 675
  EFI, 184
  hardware extraction, kernel, 295–297
  hybrid kernel, 265
  Intel trap handlers, 272–275
  I/O Kit, 50, 737
  iOS, 12, 310
  Jaguar, 6
  kdebug, 165–170
  kernel, 50
    architecture, 302–305
  kpi_socket, 667
  kqueues, 555
  ledgers, 398
  Lion, 8
  MAC, 560
  Mach, 49
    microkernels, 343
  Memorystatus, 546
  microkernels, 264, 343
  ml_functions, 296–297
  Mountain Lion, 9
  OS X, 266
  osfmk/man, 345
  packet filtering, 693, 697
  pagers, 486
  Panther, 6
  Puma, 6
  regular expressions, 306
  runtime services, 191
  sandboxing, 89
  64-bit, system calls, 283–284
  Snow Leopard, 8
  socket filters, 695–696
  sources, 299–308
  source tree, 305–308
  stack_snapshot, 162–165
  struct proclist, 507–508
  system sockets, 556
  threads, 512
  Tiger, 7
  timer interrupts, 431–436
X Kernel, 12
xar(1), 217
xattr(1), 608, 609
XBD, 503
XCode, 20, 148, 173, 174
xcodebuild(1), 723
XCU, 503
XDR. *See* external data representation
XgridFoundation, 40
.xib, 28
xinetd, 232–234
X11User.pkg, 216
XNU. *See* X is Not UNIX
XPC, 79
  Cocoa, 254
  GCD, 253
  iOS, 253–257
  kill -9, 253
  launchd, 253–257
  Lion, 253–257
  messages, 255–256
  MIG, 256
  object types, 254–255

Objective-C, 256
property lists, 257
SandBoxedFetch, 257
services, 256–257
<xpc/connection.h>, 255–256
xpc_connection_send_barrier, 255
xpc_connection_send_message, 255
xpc_connection_send_message_with_reply, 255
xpc_connection_set_target_queue, 257
xpc_dictionary_create_replay, 257
XPCKit, 254
xpc_main, 256
xpc_object_t, 256
XPCServices, 257
XSH, 503
XT-PICs. *See* Legacy PICs

Y

yielding, 415

Z

zalloc(), 470, 544
*zalloc(), 469
*zalloc_canblock(), 469
zalloc_noblock(), 544
-zc, 330
Z_CALLERACCT, 469, 544
ZeroConf, 6
Z_EXHAUSTIBLE, 469
Z_EXPAND, 469
Z_FOREIGN, 469
zfree(), 469
ZFS, 16
-zinfop, 330
zinit(), 469, 544
zlog, 330
Z_NOENCRYPT, 469
zombie state, 93–94
zones
BSD, 541–544
Lion, 542
Mach, 467–473
boot, 470–471
debugging, 473
garbage collection, 471–473
OS X, 470–471
zone_bootstrap(), 470
zone_change(), 469
zone_init(), 470
-zp, 330
zprint(1), 467
zrecs, 330
Z-shell, 21