
Figures

FIGURE 1–1 Microsoft Windows NT 3.1

FIGURE 1–2 Copland architecture

FIGURE 1–3 BeOS

FIGURE 1–4 NEXTSTEP

FIGURE 1–5 The timeline of NeXT’s operating systems

FIGURE 1–6 OPENSTEP

FIGURE 1–7 Mac OS 8

FIGURE 1–8 Mac OS 9

FIGURE 1–9 Rhapsody

FIGURE 1–10 Yellow Box running on Microsoft Windows XP

FIGURE 1–11 An approximation of the Mac OS X timeline

FIGURE 1–12 Mac OS X Public Beta

FIGURE 2–1 The high-level architecture of Mac OS X

FIGURE 2–2 Sequence of GDB commands for generating a crash report

FIGURE 2–3 The structure of the Mach-O header (32-bit version)

FIGURE 2–4 A trivial C program to be compiled to an “empty” executable

FIGURE 2–5 Displaying the load commands in an executable’s Mach-O header

FIGURE 2–6 Creating fat binaries

FIGURE 2–7 A Universal Binary containing PowerPC and x86 Mach-O executables

FIGURE 2–8 Compiling dynamic and static libraries

FIGURE 2–9 Using a custom initialization routine in a dynamic shared library

FIGURE 2–10 Using weak symbols on Mac OS X 10.2 and newer

FIGURE 2–11 Interposing a library function through `dylld`

FIGURE 2–12 Hierarchical structure of the iTunes application bundle

FIGURE 2–13 Bundle structure of a Mac OS X framework

FIGURE 2–14 Standard frameworks on Mac OS X

FIGURE 2–15 Standard umbrella frameworks on Mac OS X

FIGURE 2–16 Listing the load commands in a Mach-O file

FIGURE 2–17 Printing libraries loaded in a program

FIGURE 2–18 Determining whether a Mach-O file is prebound

FIGURE 2–19 The Mac OS X graphics and multimedia architecture

FIGURE 2–20 The key constituents of Quartz

FIGURE 2–21 An overview of Quartz Extreme

FIGURE 2–22 An overview of Quartz Extreme with Accelerated 2D

FIGURE 2–23 Interfaces to OpenGL in Mac OS X

FIGURE 2–24 Image processing with Core Image

FIGURE 2–25 Core Image and Core Video in the QuickTime video-rendering pipeline

FIGURE 2–26 Using `nibtool` to view the contents of a nib file

FIGURE 2–27 A Core Data stack

FIGURE 2–28 Using precompiled headers

FIGURE 2–29 A trivial AppleScript program

FIGURE 2–30 Command-line interaction with the Disc Recording framework

FIGURE 2–31 Command-line interaction with the Disk Images framework

FIGURE 2–32 Using the `sips` command to resample an image and convert its format

FIGURE 2–33 Using the `mdfind` command to find files matching a given query

FIGURE 2–34 The Mac OS X security architecture

FIGURE 2–35 Using Authorization Services

FIGURE 2–36 Examining keychains using the `security` command

FIGURE 2–37 Interacting with Directory Services by using command-line tools

FIGURE 2–38 Using the `scutil` command to access the System Configuration dynamic store

FIGURE 2–39 An audit control file

FIGURE 2–40 Xgrid architecture

FIGURE 2–41 Xsan architecture

FIGURE 3–1 Architecture of a dual-processor Power Mac G5 system

FIGURE 3–2 HyperTransport I/O link

FIGURE 3–3 The PowerPC 9xx family and the POWER4+

FIGURE 3–4 Caches and buffers in the 970FX

FIGURE 3–5 Retrieving processor cache information using the `sysctl` command

FIGURE 3–6 Address translation in the 970FX MMU

FIGURE 3–7 Data prefetching in AltiVec

FIGURE 3–8 PowerPC UISA and VEA registers

FIGURE 3–9 Retrieving and displaying the Timebase Register

FIGURE 3–10 Code template for inline assembly in the GNU assembler

FIGURE 3–11 PowerPC OEA registers

FIGURE 3–12 Precision of the floating-point-estimate instruction on the G4 and the G5

FIGURE 3–13 The core of the 970FX

FIGURE 3–14 The 970FX instruction pipeline

FIGURE 3–15 The FPU and FXU/LSU issue queues in the 970FX

FIGURE 3–16 A trivial AltiVec program

FIGURE 3–17 Displaying the contents of the VRSAVE

FIGURE 3–18 A simple C function that calls another function

FIGURE 3–19 Assembly code depicting an indirect function call

FIGURE 3–20 Assembly code depicting a direct function call

FIGURE 3–21 Darwin 32-bit ABI runtime stack

FIGURE 3–22 Assembly listing for a C function with no arguments and an empty body

FIGURE 3–23 Examples of stack usage in functions

FIGURE 3–24 Examples of stack usage in functions (continued from Figure 3–23)

FIGURE 3–25 Printing a function call stack trace

FIGURE 3–26 A recursive function to compute factorials

FIGURE 3–27 Annotated assembly listing for the function shown in Figure 3–26

FIGURE 3–28 A hardware-based compare-and-store function for the 970FX

FIGURE 3–29 Overview of function rerouting by instruction patching

FIGURE 3–30 Unconditional branch instruction on the PowerPC

FIGURE 3–31 Implementation of function rerouting by instruction patching

FIGURE 3–32 Function rerouting in action

FIGURE 3–33 Tracing an “empty” C program using `amber`

FIGURE 3–34 A C program with instructions that are illegal in user space

FIGURE 3–35 Tracing program execution with `amber`

FIGURE 3–36 Accounting for instructions traced by `amber`

- FIGURE 3–37 `simg5` output
- FIGURE 4–1 Defining a Forth word: syntactic requirements and conventions
- FIGURE 4–2 Physical memory properties in the device tree of a PowerBook G4
- FIGURE 4–3 Physical memory properties in the device tree of a Power Mac G5
- FIGURE 4–4 Dumping NVRAM contents
- FIGURE 4–5 The Towers of Hanoi: layout and relative dimensions of on-screen objects
- FIGURE 4–6 The Towers of Hanoi: simulating recursion using a stack
- FIGURE 4–7 The Towers of Hanoi: Forth code for animation
- FIGURE 4–8 The Towers of Hanoi: Forth code for the program’s core logic
- FIGURE 4–9 Actual photo of the Towers of Hanoi program in Open Firmware
- FIGURE 4–10 Fabricating and using a mouse pointer in Open Firmware
- FIGURE 4–11 AND and XOR masks for an X-shaped pointer
- FIGURE 4–12 Pixel-addressable printing in Open Firmware made possible by stealing a font
- FIGURE 4–13 A clock implemented in the Open Firmware environment
- FIGURE 4–14 PPM image data for a 4×4-pixel image
- FIGURE 4–15 A window created using Open Firmware primitives
- FIGURE 4–16 A bootinfo file
- FIGURE 4–17 Properties of the chosen device node as seen from Mac OS X
- FIGURE 4–18 Retrieving a BSD device node’s Open Firmware path
- FIGURE 4–19 An AppleRAID software RAID configuration
- FIGURE 4–20 The EFI architecture
- FIGURE 4–21 Booting an operating system through EFI
- FIGURE 4–22 Using the EFI shell
- FIGURE 4–23 A GPT-partitioned disk
- FIGURE 4–24 Listing the partitions on a GPT-partitioned disk
- FIGURE 5–1 A high-level view of Mac OS X system startup
- FIGURE 5–2 Low-level processor initialization
- FIGURE 5–3 The kernel’s per-processor data table
- FIGURE 5–4 The entry for the PowerPC 970FX in the processor-type table
- FIGURE 5–5 Data structure and related definitions of a patch-table entry
- FIGURE 5–6 The kernel’s patch table

FIGURE 5–7 Flushing the L2 cache on the PowerPC 970FX

FIGURE 5–8 High-level processor initialization

FIGURE 5–9 Calculating the PowerPC PTEG hash table size used by the kernel

FIGURE 5–10 Initialization of the `kprintf()` function

FIGURE 5–11 Bootstrapping kernel subsystems

FIGURE 5–12 High-level virtual memory subsystem initialization

FIGURE 5–13 Initializing page frames during VM subsystem initialization

FIGURE 5–14 I/O Kit initialization

FIGURE 5–15 Loading a replacement panic user interface image into the kernel

FIGURE 5–16 Testing the panic user interface

FIGURE 5–17 BSD initialization

FIGURE 5–18 Domain and protocol initialization

FIGURE 5–19 Mounting the root file system

FIGURE 5–20 Finding the root device with help from the I/O Kit

FIGURE 5–21 Doing the core work of finding the root device

FIGURE 5–22 Slave processor initialization

FIGURE 5–23 A `launchd` configuration file

FIGURE 5–24 Creating a periodic `launchd` job

FIGURE 5–25 Guidelines and caveats for creating `launchd`-compliant daemons

FIGURE 5–26 A trivial echo server called `dummyd`

FIGURE 5–27 The contents of the `com.osxbook.dummyd.plist` configuration file

FIGURE 5–28 A high-level depiction of `launchd`'s operation

FIGURE 5–29 The sequence of operations performed by `/etc/rc`

FIGURE 5–30 The sequence of operations performed by `/etc/rc` (continued from Figure 5–29)

FIGURE 5–31 Overview of session launching by `launchd`

FIGURE 5–32 Implementation of session creation and launching by `launchd`

FIGURE 5–33 Important steps performed by the `loginwindow` application

FIGURE 5–34 The handling of user logout, system restart, and system shutdown by `loginwindow`

FIGURE 5–35 Single-user bootstrap through `launchd`

FIGURE 6–1 Retrieving basic host information using Mach calls

FIGURE 6–2 Retrieving clock attributes and time values in Mach

FIGURE 6–3 Setting an alarm using Mach calls

FIGURE 6–4 Using Mach calls to retrieve scheduling and virtual memory statistics

FIGURE 6–5 The Mach-O segment containing the exception vectors in the kernel executable

FIGURE 6–6 The kernel’s exception vectors

FIGURE 6–7 Trap vectors in Third Edition UNIX

FIGURE 6–8 Structure for a thread’s machine-dependent state

FIGURE 6–9 A high-level view of exception processing in the xnu kernel

FIGURE 6–10 Common code for exception processing

FIGURE 6–11 Processing of traps, interrupts, and system calls

FIGURE 6–12 Details of system call processing in Mac OS X

FIGURE 6–13 Mapping an incoming system call number to an index in the first-level system call dispatch table

FIGURE 6–14 Passing a long long parameter in 32-bit and 64-bit ABIs

FIGURE 6–15 An example of system call argument munging

FIGURE 6–16 Details of the final dispatching of BSD system calls

FIGURE 6–17 System call data structures in Third Edition UNIX

FIGURE 6–18 Creating a user-space system call stub

FIGURE 6–19 Directly invoking a BSD system call

FIGURE 6–20 Multiple ways of retrieving a Mach task’s self port

FIGURE 6–21 Mach trap table data structures and definitions

FIGURE 6–22 Mach trap table initialization

FIGURE 6–23 Setting up the `pid_for_task()` Mach trap

FIGURE 6–24 Testing the `pid_for_task()` Mach trap

FIGURE 6–25 Testing the Thread Info UFT

FIGURE 6–26 Testing the Facility Status UFT

FIGURE 6–27 Enabling the kernel’s Blue Box support

FIGURE 6–28 Displaying the contents of the comm area

FIGURE 6–29 Initialization of the `kprintf()` function

FIGURE 6–30 Invocation of CHUD system-wide hooks for traps and ASTs

FIGURE 6–31 Invoking a disabled PowerPC-only system call

FIGURE 6–32 Implementing a kernel extension to register a PowerPC-only system call

FIGURE 6–33 Using `kgmon` and `gprof` for kernel profiling

FIGURE 6–34 Implementation of the `profil()` system call

FIGURE 6–35 The Mac OS X auditing system

FIGURE 6–36 Audit macros in the kernel and how they are used

FIGURE 6–37 The composition of a debug code in the `kdebug` facility

FIGURE 6–38 kdebug tracing in the BSD system call handler

FIGURE 6–39 Using the kdebug facility in a program

FIGURE 6–40 Common header file for using the diagnostics system call interface

FIGURE 6–41 Retrieving boot-screen information using a diagnostics system call

FIGURE 6–42 Retrieving the physical address (if any) for a virtual address in the caller’s address space

FIGURE 6–43 Retrieving physical memory using a diagnostics system call

FIGURE 6–44 The low-memory global data area

FIGURE 6–45 Data structures for holding per-processor information

FIGURE 6–46 Causing an exception and retrieving the corresponding counter from the kernel

FIGURE 6–47 Maintenance of hardware exception counters in the kernel

FIGURE 6–48 Processing of low-tracing-related boot-time arguments during system startup

FIGURE 6–49 An example of low-trace record generation by kernel code

FIGURE 6–50 Control data structures for the VMM facility

FIGURE 6–51 A program to run machine code within a VM using the VMM facility

FIGURE 6–52 Result of running a sequence of machine instructions in a VM using the vmachmon32 program

FIGURE 6–53 Result of running a recursive factorial function within a VM using the vmachmon32 program

FIGURE 6–54 Software dependencies in the kernel build process

FIGURE 7–1 An overview of the Mac OS X process subsystem

FIGURE 7–2 The processor set structure in the xnu kernel

FIGURE 7–3 The processor structure in the xnu kernel

FIGURE 7–4 A processor set containing two processors

FIGURE 7–5 A processor set with two processors on its active queue

FIGURE 7–6 Retrieving information about processors on the host

FIGURE 7–7 Out-of-line data received by a process from the kernel as a result of a Mach call

FIGURE 7–8 Starting and stopping a processor through the Mach processor interface

FIGURE 7–9 The task structure in the xnu kernel

FIGURE 7–10 The thread structure in the xnu kernel

FIGURE 7–11 Functions for creating kernel threads

FIGURE 7–12 The shuttle and thread data structures in Mac OS X 10.0

FIGURE 7–13 The shuttle and the thread within a single structure in Mac OS X 10.3

FIGURE 7–14 Retrieving the current thread (shuttle) and the current activation on Mac OS X 10.3 and 10.4

FIGURE 7–15 Blocking with and without continuations

FIGURE 7–16 Continuation-related aspects of the BSD `uthread` structure

FIGURE 7–17 The use of continuations in the implementation of the `select()` system call

FIGURE 7–18 The skeleton of the `nfsiod` program

FIGURE 7–19 Creating a Mach task

FIGURE 7–20 Creating a Mach thread

FIGURE 7–21 Retrieving detailed task and thread information from the kernel

FIGURE 7–22 Counting the number of system calls made by a process

FIGURE 7–23 The BSD `proc` structure

FIGURE 7–24 The BSD `uthread` structure

FIGURE 7–25 User-space processing of the `fork()` system call

FIGURE 7–26 Registering handlers to be run before and after a `fork()` system call invocation

FIGURE 7–27 Kernel-space processing of the `fork()` system call

FIGURE 7–28 Verifying the values returned by the kernel in the case of a `raw fork()` system call

FIGURE 7–29 Kernel-space processing of the `vfork()` system call

FIGURE 7–30 Verifying that the child borrows the parent’s resources during a `vfork()` system call

FIGURE 7–31 Pthread creation in the system library

FIGURE 7–32 Creating and running Java threads

FIGURE 7–33 Using the `NSTask` Cocoa class

FIGURE 7–34 Using the `NSThread` Cocoa class

FIGURE 7–35 Launching an application through the Carbon Process Manager

FIGURE 7–36 Using Carbon Multiprocessing Services

FIGURE 7–37 Creating Carbon Thread Manager threads

FIGURE 7–38 Scheduling-related initializations during system startup

FIGURE 7–39 Real-time clock interrupt processing

FIGURE 7–40 System clock configuration

FIGURE 7–41 Converting between absolute- and clock-time intervals

FIGURE 7–42 Scheduler startup

FIGURE 7–43 Sampling the value of the scheduler tick

FIGURE 7–44 Timer call processing

FIGURE 7–45 The scheduler’s bookkeeping function

FIGURE 7–46 A nonexhaustive call graph of functions involved in thread execution and scheduling

FIGURE 7–47 The run queue structure

FIGURE 7–48 Important scheduling-related fields of the task and thread structures

FIGURE 7–49 Computation of the timesharing priority of a thread

FIGURE 7–50 User-space computation of `sched_pri_shift` and `sched_tick_interval`

FIGURE 7–51 Computation of the usage-to-priority conversion factor for timeshared priorities

FIGURE 7–52 Approximating multiplication by 5/8 as implemented in the scheduler

FIGURE 7–53 Experimenting with the `THREAD_PRECEDENCE_POLICY` scheduling policy

FIGURE 7–54 Experimenting with the `THREAD_TIME_CONSTRAINT_POLICY` scheduling policy

FIGURE 7–55 Recomputing a thread’s priority on a scheduling-policy change

FIGURE 7–56 Structure for holding executable image parameters during the `execve()` system call

FIGURE 7–57 The operation of the `execve()` system call

FIGURE 7–58 Creation of the user stack during the `execve()` system call

FIGURE 7–59 A Mach-O executable with a custom stack

FIGURE 7–60 User stack arranged by the `execve()` system call

FIGURE 7–61 Dumping the Launch Services registration database

FIGURE 7–62 Example of a UTI declaration

FIGURE 8–1 An overview of the Mac OS X memory subsystem

FIGURE 8–2 A shell script for reading kernel virtual memory

FIGURE 8–3 Using the `kvm(3)` interface to read kernel memory

FIGURE 8–4 Determining the size of physical memory on a system

FIGURE 8–5 The Mac OS X implementation of the Mach VM architecture

FIGURE 8–6 Details of the Mac OS X Mach VM architecture

FIGURE 8–7 The Mach pager interface in Mac OS X

FIGURE 8–8 Symmetric copy-on-write using shadow objects

FIGURE 8–9 Asymmetric copy-on-write using copy objects

FIGURE 8–10 The structure of a resident page

FIGURE 8–11 Grabbing a page from the free list

FIGURE 8–12 Reserving physical memory

FIGURE 8–13 Initialization of the hardware-independent part of the Mach VM subsystem

FIGURE 8–14 Controlling memory inheritance

FIGURE 8–15 Protecting memory

FIGURE 8–16 Accessing another task’s memory

FIGURE 8–17 Common header file for the shared memory client-server example

FIGURE 8–18 Source for the shared memory client

FIGURE 8–19 Source for the shared memory server

FIGURE 8–20 Examples of exported UBC routines

FIGURE 8–21 System-wide shared memory setup during bootstrapping

FIGURE 8–22 dyld’s operation while loading a non-split-segment file

FIGURE 8–23 dyld’s operation while loading a split-segment file

FIGURE 8–24 Using `shared_region_map_file_np()`

FIGURE 8–25 Loading a split-segment library in the global shared region

FIGURE 8–26 Structure of the `LC_PREBOUND_DYLIB` load command

FIGURE 8–27 Reading the contents of the TWS subsystem’s global user profile cache

FIGURE 8–28 TWS-related processing during the `execve()` system call

FIGURE 8–29 The malloc zones API

FIGURE 8–30 An overview of the Mac OS X malloc implementation

FIGURE 8–31 Scalable zone data structures

FIGURE 8–32 Internal bookkeeping of tiny allocations (32-bit)

FIGURE 8–33 A program that performs tiny allocations

FIGURE 8–34 Internal bookkeeping of small malloc allocations (32-bit)

FIGURE 8–35 Processing of the `malloc()` function in the system library

FIGURE 8–36 Determining the size of the largest single `malloc()` allocation

FIGURE 8–37 Allocating 2 petabytes of virtual memory

FIGURE 8–38 Implementing a version of the `printf()` function without using `malloc()`

FIGURE 8–39 Enumerating all `malloc()`-allocated pointers in a program

FIGURE 8–40 Displaying scalable-zone statistics

FIGURE 8–41 Logging malloc operations

FIGURE 8–42 Intercepting the malloc layer

FIGURE 8–43 An overview of memory allocation in the Mac OS X kernel

FIGURE 8–44 An overview of memory deallocation in the Mac OS X kernel

FIGURE 8–45 Zone allocator functions

FIGURE 8–46 Printing sizes of kalloc zones supported in the kernel

FIGURE 8–47 Array of memory types supported by the BSD memory allocator

FIGURE 8–48 Setting up the vnode pager during the `open()` system call

FIGURE 8–49 Setting up of the vnode pager for a newly created vnode

FIGURE 8–50 Kernel processing of the `mmap()` system call

FIGURE 8–51 An overview of a page-in operation

FIGURE 8–52 Paging in from a vnode

FIGURE 8–53 An overview of a page-out operation

FIGURE 8–54 Paging out to a vnode

FIGURE 8–55 Using a 64-bit-only instruction

FIGURE 8–56 An overview of 64-bit support in Mac OS X

FIGURE 9–1 System library wrappers around Mach messaging traps

FIGURE 9–2 The layout of a complex Mach message

FIGURE 9–3 Descriptors for sending ports and out-of-line memory in Mach IPC messages

FIGURE 9–4 Requesting the kernel to include the sender’s security token in the message trailer

FIGURE 9–5 An overview of Mach IPC implementation in Mac OS X

FIGURE 9–6 The data structure for a task’s IPC space

FIGURE 9–7 A view of the internal structure of a Mach port

FIGURE 9–8 IPC-related data structures associated with a Mach task

FIGURE 9–9 IPC-related data structures associated with a Mach thread

FIGURE 9–10 The allocation of a port right

FIGURE 9–11 Listing the Mach ports and their attributes in a given process

FIGURE 9–12 An overview of the kernel processing for sending a Mach IPC message

FIGURE 9–13 An overview of the kernel processing for receiving a Mach IPC message

FIGURE 9–14 Initialization of the IPC subsystem

FIGURE 9–15 Displaying information about all known services in a bootstrap context

FIGURE 9–16 A crash-resistant server

FIGURE 9–17 `launchd` debug messages corresponding to a Mach server’s initialization

FIGURE 9–18 `launchd` debug messages corresponding to a Mach server’s relaunch

FIGURE 9–19 Common header file for the simple IPC client-server example

FIGURE 9–20 Source for the simple IPC server

- FIGURE 9–21 Source for the simple IPC client
- FIGURE 9–22 Using a port set to receive request messages destined for multiple services
- FIGURE 9–23 Inserting port rights into an IPC space
- FIGURE 9–24 Sending out-of-line memory in an IPC message
- FIGURE 9–25 Sending port rights in an IPC message
- FIGURE 9–26 Common header file for the Miscellaneous Server and its client
- FIGURE 9–27 MIG specification file for the Miscellaneous Server
- FIGURE 9–28 Programmer-provided source for the Miscellaneous Server
- FIGURE 9–29 A client for accessing the services provided by the Miscellaneous Server
- FIGURE 9–30 Creating a MIG-based client and server system
- FIGURE 9–31 Invocation of Miscellaneous Server routines by a client
- FIGURE 9–32 Initialization of MIG subsystems during kernel bootstrap
- FIGURE 9–33 Exception ports at the host, task, and thread levels
- FIGURE 9–34 Initialization of exception handling during kernel bootstrap
- FIGURE 9–35 Kernel functions involved in Mach exception delivery
- FIGURE 9–36 Delivery of Mach exceptions
- FIGURE 9–37 Processing of unresolved kernel traps
- FIGURE 9–38 An exception handler for “fixing” illegal instructions
- FIGURE 9–39 Signal notification of asynchronous I/O completion
- FIGURE 9–40 Implementation of the signal mechanism in the kernel
- FIGURE 9–41 Program-visible data structures involved in file descriptor passing
- FIGURE 9–42 Common header file for the descriptor-passing client-server implementation
- FIGURE 9–43 Implementation of the descriptor-passing server
- FIGURE 9–44 Implementation of the descriptor-passing client
- FIGURE 9–45 Working with POSIX semaphores
- FIGURE 9–46 Working with POSIX shared memory
- FIGURE 9–47 Communication in a Distributed Objects client-server system
- FIGURE 9–48 Common header file for the Distributed Objects client-server example
- FIGURE 9–49 Server source for the Distributed Objects client-server example
- FIGURE 9–50 Client source for the Distributed Objects client-server example
- FIGURE 9–51 Exception-handling macros in the Foundation framework

FIGURE 9–52 AppleScript program to speak the system version
FIGURE 9–53 AppleScript program for opening and tiling Terminal application windows
FIGURE 9–54 Sending Apple Events to the Finder from a C program
FIGURE 9–55 A program for posting distributed notifications (NSNotification)
FIGURE 9–56 A program for observing distributed notifications (NSNotification)
FIGURE 9–57 Common header file for defining notification names
FIGURE 9–58 Program for posting `notify(3)` notifications
FIGURE 9–59 Receiving notifications through multiple mechanisms
FIGURE 9–60 Monitoring a file through `notify(3)`
FIGURE 9–61 Using the `kqueue()` and `kevent()` system calls to watch for file events
FIGURE 9–62 A program for posting Core Foundation distributed notifications
FIGURE 9–63 A program for observing Core Foundation distributed notifications
FIGURE 9–64 Creating and adding run-loop input sources
FIGURE 9–65 A `CFMessagePort` server
FIGURE 9–66 A `CFMessagePort` client
FIGURE 9–67 A `CFSocket` client
FIGURE 9–68 Using a `CFRunLoopTimer`
FIGURE 9–69 An overview of Mac OS X synchronization mechanisms
FIGURE 9–70 Using locks in the kernel
FIGURE 9–71 The Mach lock set interface
FIGURE 9–72 Using Mach semaphores
FIGURE 9–73 Internal structure of a Mach semaphore
FIGURE 9–74 The structure of a Mac OS X funnel
FIGURE 9–75 Automatic funnel use in a thread-unsafe file system
FIGURE 9–76 Interfaces for advisory-mode file locking
FIGURE 10–1 An overview of the I/O Kit
FIGURE 10–2 The I/O Kit class hierarchy
FIGURE 10–3 The main function of the `IOWorkLoop` class
FIGURE 10–4 Using `IOWorkLoop` in a driver
FIGURE 10–5 The `kmod_info` structure
FIGURE 10–6 The contents of a simple kernel extension bundle
FIGURE 10–7 Excerpt from the build output of a Universal kernel extension
FIGURE 10–8 Implementation of `libkmodc++.a`

FIGURE 10–9 Declaration of the `kmod_info` structure for an I/O Kit driver kernel module

FIGURE 10–10 Implementation of `libkmod.a`

FIGURE 10–11 Declaration of the `kmod_info` structure for a generic kernel module

FIGURE 10–12 Kext management

FIGURE 10–13 Source for the `DummySysctl` kernel extension

FIGURE 10–14 The kernel's `sysctl` hierarchy

FIGURE 10–15 Implementing `sysctl` nodes

FIGURE 10–16 An I/O Kit driver's personality and dependencies

FIGURE 10–17 The driver personality of the `BootCache` kernel extension

FIGURE 10–18 An excerpt from the iPod driver's property list file

FIGURE 10–19 Header file for the `DummyDriver` I/O Kit driver

FIGURE 10–20 Implementation of the `DummyDriver` I/O Kit driver's class

FIGURE 10–21 The sequence of I/O Kit driver methods called in a driver's lifecycle

FIGURE 10–22 Processing involved in user-program-induced framebuffer rotation

FIGURE 10–23 Programmatically rotating a framebuffer

FIGURE 10–24 Accessing framebuffer memory

FIGURE 10–25 Retrieving the list of firmware variables from the I/O Registry

FIGURE 10–26 Retrieving information about loaded kernel extensions

FIGURE 10–27 Relevant contents of the `IOExternalMethod` structure corresponding to `getOrientationUC()`

FIGURE 10–28 Opening a connection to the motion sensor service object

FIGURE 10–29 Invoking a user client method given an `IOService` connection handle

FIGURE 10–30 Listing PCI devices in the system

FIGURE 10–31 Retrieving the computer's serial number and model information

FIGURE 10–32 Retrieving temperature sensor readings

FIGURE 10–33 Retrieving the MAC addresses of Ethernet interfaces in the system

FIGURE 10–34 Personality and dependency list for the `SimpleCryptoDisk` I/O Kit driver

FIGURE 10–35 Header file for the `SimpleCryptoDisk` I/O Kit driver

FIGURE 10–36 Source for the `SimpleCryptoDisk` I/O Kit driver

FIGURE 10–37 Reading the contents of cleartext storage directly from the

storage medium

- FIGURE 10–38 Using encrypted storage with the SimpleCryptoDisk filter scheme driver
- FIGURE 10–39 Dissection of a kernel panic dump
- FIGURE 10–40 Logging in Mac OS X
- FIGURE 10–41 Periodic delivery of log messages to readers of the log device
- FIGURE 10–42 Objects that implement target-side KDP
- FIGURE 10–43 A KDB setup
- FIGURE 10–44 A simple C function
- FIGURE 10–45 Stabs-encoded debugging information in an assembly file
- FIGURE 11–1 Using ioctl operations to display information about a disk device
- FIGURE 11–2 Listing a disk's partitions
- FIGURE 11–3 A disk partitioned using the Apple partitioning scheme
- FIGURE 11–4 An overview of disk arbitration
- FIGURE 11–5 Using Disk Arbitration to obtain a disk's description
- FIGURE 11–6 Expressing dissent against a mount operation
- FIGURE 11–7 Monitoring the appearance and disappearance of storage devices
- FIGURE 11–8 An I/O Kit stack depicting a disk device and its partitions
- FIGURE 11–9 The Mac OS X block and character device switch structures
- FIGURE 11–10 A typical read request's journey to a disk device
- FIGURE 11–11 How a file descriptor leads to a file in Mac OS X
- FIGURE 11–12 An overview of the vnode/vfs layer's role in the operating system
- FIGURE 11–13 A mounted file system and its vnodes
- FIGURE 11–14 Configuration information for file system types supported by the kernel
- FIGURE 11–15 Displaying information about all available file system types
- FIGURE 11–16 Vnode operations vectors in the VFS layer
- FIGURE 11–17 Initialization of vnode operations vectors during bootstrap
- FIGURE 11–18 Mac OS X file system APIs
- FIGURE 11–19 Structure of a symbolic link on the MS-DOS file system on Mac OS X
- FIGURE 11–20 Implementation of the WebDAV file system
- FIGURE 11–21 Architecture of the Spotlight system
- FIGURE 11–22 Retrieving and setting an MDItem attribute
- FIGURE 11–23 An overview of the fsevents mechanism's implementation

FIGURE 11–24 Event generation in the fsevents mechanism

FIGURE 11–25 A file system change logger based on the fsevents mechanism

FIGURE 11–26 Programmatically launching the Spotlight search window

FIGURE 11–27 Pseudocode for creating and running a Spotlight query using the MDQuery interface

FIGURE 11–28 A program for executing raw Spotlight queries

FIGURE 11–29 A program to create and set an ACL

FIGURE 11–30 An overview of the kauth mechanism

FIGURE 11–31 Kauth scopes and actions

FIGURE 11–32 An overview of the scope and listener data structures in the kauth mechanism

FIGURE 11–33 Authorization in the vnode scope

FIGURE 11–34 The design of a simple vnode-level file system activity monitor

FIGURE 11–35 Common header file for the vnode-level file system activity monitor

FIGURE 11–36 Source for the vnode-level file system activity monitor kernel extension

FIGURE 11–37 Source for the user-space retrieval program for the vnode-level file system activity monitor

FIGURE 12–1 A disk containing two HFS+ volumes

FIGURE 12–2 The structure of an HFS+ B-Tree node

FIGURE 12–3 The structure of an HFS+ B-Tree header node

FIGURE 12–4 The structure of an HFS+ B-Tree record

FIGURE 12–5 The contents of a hypothetical HFS+ B-Tree

FIGURE 12–6 The structure of an HFS+ volume

FIGURE 12–7 The contents of an HFS+ volume header

FIGURE 12–8 Finder information contained in the volume header of a boot volume

FIGURE 12–9 An HFS wrapper volume containing an embedded HFS+ volume

FIGURE 12–10 The contents of a master directory block

FIGURE 12–11 Hinting to the file system where to look for free space on a volume

FIGURE 12–12 Examining allocation block consumption on a volume

FIGURE 12–13 Searching in the Catalog B-Tree

FIGURE 12–14 An overview of accessing a file's contents

FIGURE 12–15 Using the Carbon File Manager API to convert a CNID to a Unix pathname

FIGURE 12–16 The contents of a Catalog B-Tree’s header node

FIGURE 12–17 Searching in the Extents Overflow B-Tree

FIGURE 12–18 The contents of an Extents Overflow B-Tree’s header node

FIGURE 12–19 Searching in the Attributes B-Tree

FIGURE 12–20 Programmatically setting an extended attribute

FIGURE 12–21 Unicode filenames in Mac OS X

FIGURE 12–22 Algorithm for determining ownership rights to a file system object

FIGURE 12–23 Processing of the `F_FULLFSYNC` file control operation

FIGURE 12–24 The VFS-layer journaling interface in Mac OS X

FIGURE 12–25 An overview of the file-system-independent journal used by HFS+

FIGURE 12–26 Examining the HFS+ private metadata folder and hard-link creation

FIGURE 12–27 Use of the private metadata folder for storing unlinked busy files

FIGURE 12–28 A Python program to resolve an alias

FIGURE 12–29 A Perl program to create a file with eight fragments on an HFS+ volume

FIGURE 12–30 Layout of the HFS+ metadata zone

FIGURE 12–31 Transitions between Hot File Clustering stages

FIGURE 12–32 Searching in the Hot Files B-Tree

FIGURE 12–33 Viewing VFS event notifications using `kqueue/kevent`

FIGURE 12–34 Querying a volume’s capabilities

FIGURE A–1 Using the `blesstool` command to view boot-related information in the volume header

FIGURE A–2 Using command-line tools to display partitioning-related information

FIGURE A–3 Compiling an “empty” C program on the x86, PowerPC, and 64-bit PowerPC

FIGURE A–4 Running a program both natively and under Rosetta

FIGURE A–5 Testing a nonexecutable stack on the x86 version of Mac OS X

FIGURE A–6 Setting up a newly created thread’s state on the x86 version of Mac OS X

FIGURE A–7 Invoking a system call on the x86 version of Mac OS X