INSIDE MAC OS X

Kernel Environment

December 2000

Contents

C H A P T E R 1

About This Book

The purpose of this book is to provide fundamental high-level information about the Mac OS X core operating-system architecture. It also provides background for system programmers and developers of device drivers, file systems, and network extensions. The book concentrates on those areas where Mac OS X system architecture differs from other, similar operating systems.

Therefore, this book does not delve deeply into the specific **APIs** or programmatic use of the individual components of the Mac OS X core operating system, collectively known as the **kernel**. These components include **Mach**, **BSD**, the **I/O Kit**, networking, and the file system. To learn more about how to program for these components, you should see the specific documentation for each of them.

Audience Profile

This book has a wide and diverse audience—specifically, the set of potential system software developers for Mac OS X, including the following sorts of developers:

- **Device-driver writers.** Device-driver writers make up the largest portion of the audience. You will be some of the first developers to start writing code for Mac OS X. Most device-driver writers will have come from one of the following platform backgrounds:
    - **Generic device drivers (third-party solutions).** Your company writes drivers for many operating systems—Mac OS, Microsoft Windows, Linux, and so forth. You'd like to know how writing for Mac OS X will be different (and how it will be similar).

    - **UNIX platforms and variants of UNIX platforms such as FreeBSD, Linux, Solaris, and others.**You've been writing drivers for platforms such as Linux and FreeBSD. You want to know how to modify your code (or change your habits) when writing for Mac OS X. You may have certain preconceived notions about writing device drivers. The kernel environment model in Mac OS X differs in several respects from what you are used to; you'll need to understand those differences.
    - **Windows NT.**You have been writing for the Windows NT platform. Now, you have decided to broaden your scope. You need to know how to write for Mac OS X.
    - **Mac OS (Classic).**You have been writing drivers for Mac OS for a long time and you know everything there is to know about Mac OS 8 and 9. However, Mac OS X is different. You need to know how to modify your code (or change your habits) when porting to Mac OS X.
- **Network-extension writers.**You need to know how the networking subsystem fits in with the rest of the core operating system. You come from a platform background similar to the device-driver writers.
- **File-system writers.**You want to support a file system such as AFS or NTFS. You need to understand how to fit your code into Mac OS X.
- **Developers of software requiring very low-level access to file-system data.** You are writing software that needs low-level access to the file system, applications such as on-the-fly compression, encryption, and virus checking. You need to understand how to write Virtual File System stacks to add value on top of Mac OS X.
- **System programmers familiar with BSD, Linux, and similar operating systems.**As a system programmer, you're wondering what Mac OS X has to offer you. This book addresses the differences between Mac OS X and the "standard" BSD and Mach 3.0 implementations.
- **Customers with special requirements.**Because the Mac OS X kernel technology is **Open Source**, some developers will be planning to make changes to the underlying operating system in order to meet special requirements at their sites (or example, a university customer may wish to add **Kerberos** support). This book tells you how the parts of the Mac OS X kernel fit together and interact.
- **Applications developers, students and others.** You're not a system programmer, but you're interested in how Mac OS X is put together. You may already be familiar with BSD, Linux, or other UNIX variants and possibly

Windows NT as well. Although you don’t expect to need to know a great deal about the kernel environment, you are nonetheless interested in some details of memory allocation, process management, and the like.

Road Map

The goal of this book is to describe the underlying global concepts of the core operating-system development environment. That is, it describes shared concepts that are not specific to any one of the primary subsystems: Mach, the I/O Kit, BSD, file systems, or networking. All concepts should be applicable to each of these subsystems and are therefore useful to developers from any background (such as device-driver writers).

This book does not delve deeply into the specific APIs or programmatic use of the individual subsystems of the operating system. Each of these subsystems will be the subject of its own documentation.

The chapters of this book describe the kernel environment from different angles. Discussion of specific APIs, however, is left to more in-depth component-specific documentation.

The next chapter provides an overview of the Mac OS X kernel architecture. There follow several chapters that discuss each of the architectural components of Mac OS X in more detail, one chapter per component. These are followed in turn by a chapter that discusses extending the kernel, from a conceptual viewpoint.

The last chapter covers available kernel services. For each service, it provides a brief description as well as listing which components are either a provider or a client. The book ends with a glossary of terms used throughout the preceding chapters as well as a comprehensive reference bibliography.

The glossary covers many of the terms used throughout the earlier chapters of this book; these terms are highlighted in bold when first used. Rather than stop and define each term as it appears, the definitions are all in the glossary. If a term seems familiar, it probably means what you think it does. If it’s unfamiliar, check the glossary. In any case, all readers may want to skim through the glossary, in case there are subtle differences between Mac OS X usage and that of other operating systems.

The bibliography provides numerous pointers to other reference materials. The goal of this book is very broad, providing a firm grounding in the fundamentals of Mac OS X kernel programming for developers from many backgrounds. Unfortunately, to do a complete and comprehensive job would fill an entire library, rather than one book. Instead, this book includes references to additional publications already in existence. Some of these are Apple publications; others are external documents. To make things easier, the bibliographic references are grouped into categories.

By the time you have finished this book, you should have a basic understanding of Mac OS X system internals and how to begin programming Mac OS X system software. You should also have a good idea of what you'll need to read next.

Other Apple Publications

This book, *Kernel Environment*, is part of a planned series, Inside Mac OS X. Be sure to read the first book in the series, *System Overview*, if you are not familiar with Mac OS X.

You can obtain other books in the Inside Mac OS X series (as they become available) using publish-on-demand. . To obtain a printed copy of an Inside Mac OS X book, go to the Fatbrain.com website at www.fatbrain.com.

Information on the Web

Apple maintains several websites where developers can go for general and technical information on Mac OS X.

- Apple Developer Connection: Developer Documentation (`developer.apple.com/techpubs`) . Features the same documentation that is installed on Mac OS X, except that often the documentation is more up-to-date. Also includes legacy documentation.

- Apple Developer Connection: Mac OS X (`developer.apple.com/macosx`) . Offers SDKs, release notes, product notes and news, and other resources and information related to Mac OS X.
- AppleCare Tech Info Library (`til.info.apple.com`) . Contains technical articles, tutorials, FAQs, technical notes, and other information.

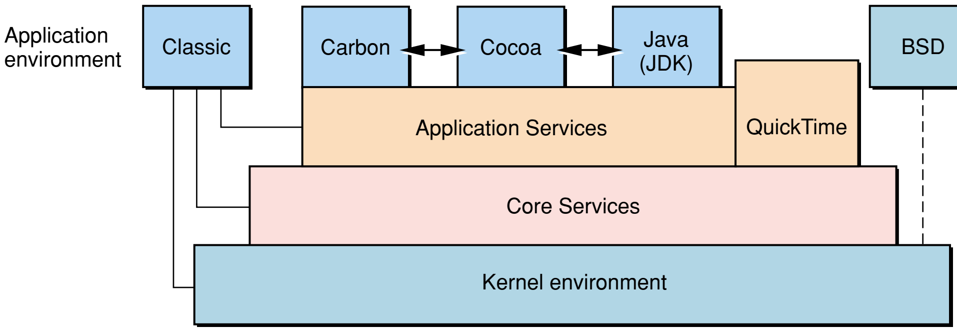
C H A P T E R 2

Kernel Architecture

Mac OS X provides many benefits to the Macintosh user and developer communities. These benefits include improved reliability and performance, enhanced networking features, an object-based system programming interface, and increased support for industry standards.

In creating Mac OS X, Apple has completely re-engineered the Mac OS core operating system. Forming the foundation of Mac OS X is the kernel. Figure 2-1 illustrates the Mac OS X architecture.

Figure 2-1 Mac OS X architecture



The kernel provides many enhancements for Mac OS X. These include **preemption**, **memory protection**, enhanced performance, improved networking facilities, support for both Macintosh (Extended and Standard) and non-Macintosh (UFS, ISO 9660) file systems, object-oriented APIs, and more. Two of these features, preemption and memory protection, lead to a more robust environment.

In Mac OS 8 and 9, applications cooperate to share processor time. Similarly, all applications share the memory of the computer among them. Mac OS 8 and 9 are **cooperative multitasking** environments. The responsiveness of all processes is compromised if even a single application doesn't cooperate. On the other hand, real-time applications such as multimedia need to be assured of predictable, time-critical, behavior.

In contrast, Mac OS X is a **preemptive multitasking** environment. In Mac OS X, the kernel provides enforcement of cooperation, scheduling processes to share time (preemption). This supports real-time behavior in applications that require it.

In Mac OS X, processes do not normally share memory. Instead, the kernel assigns each **process** its own **address space**, controlling access to these address spaces. This control ensures that no application can inadvertently access or modify another application's memory (protection). Size is not an issue; with the virtual memory system included in Mac OS X, each application has access to its own 4 GB memory address space.

Viewed together, all applications are said to run in user space, but this does not imply that they share memory. User space is simply a term for the combined address spaces of all user-level applications. The kernel itself has its own address space, called kernel space. In Mac OS X, no application can modify the memory of the system software (the kernel).

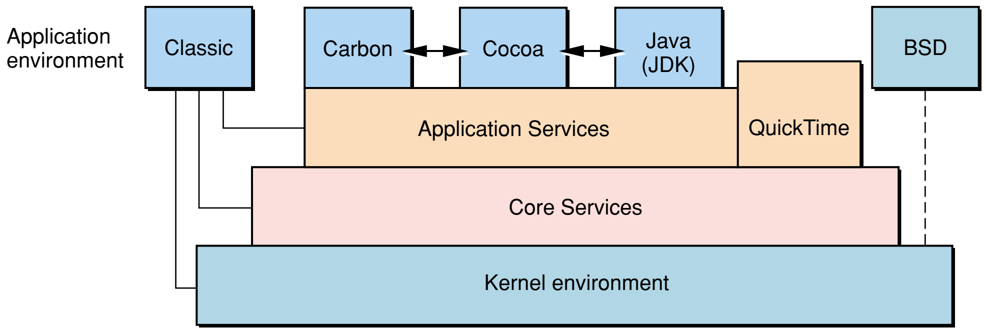
Although user processes do not share memory by default as in Mac OS 8 and 9, communication (and even memory sharing) between applications is still possible. For example, the kernel offers a rich set of primitives to permit some sharing of information among processes. These primitives include shared libraries and frameworks. Mach messaging provides another approach, handing memory from one process to another. Unlike Mac OS 8 and 9, however, memory sharing cannot occur without explicit action by the programmer.

Darwin

The Mac OS X kernel is a key part of Apple's **Open Source** initiative. The Mac OS X kernel is also the core of an operating system product called **Darwin**. Darwin is a complete operating system based on many of the same technologies that underlie Mac OS X. However, Darwin does not include Apple's proprietary graphics or applications layers, such as Quartz, QuickTime, or OpenGL.

Figure 2-2 shows the relationship between Darwin and Mac OS X. Both build upon the same kernel, but Mac OS X adds Core Services, Application Services and QuickTime, as well as the**Classic**, **Carbon**, **Cocoa**, and Java (JDK) application environments. Both Darwin and Mac OS X include the BSD command-line application environment; however, in Mac OS X, this environment is usually hidden.

Figure 2-2 Darwin and Mac OS X



Darwin technology is based on **BSD**, Mach 3.0, and Apple technologies. Best of all, Darwin technology is Open Source technology, which means that developers have full access to the source code. In effect, Mac OS X third-party developers can be part of the Darwin core system software development team. Developers can also see how Apple is doing things in the core operating system and adopt (or adapt) code to use within their own products. Refer to the **Apple Public Source License** for details.

Because the same system software forms the core of both Mac OS X and Darwin, system software developers can write software that runs on both Mac OS X and Darwin with few, if any, required changes. The only difference might be in the way the software interacts with the application environment.

The Mac OS X core operating system is based on proven technology from many sources. A large portion of this technology is derived from FreeBSD, a version of 4.4BSD that offers advanced networking, performance, security, and compatibility features. Other parts of the system software, such as Mach, are based on technology previously used in Apple’s MkLinux project, in Mac OS X Server, and in technology acquired from NeXT. Much of the code is platform-independent. All of the core operating-system code is available in source form.

The core technologies have been chosen for several reasons. Mach provides a clean set of abstractions for dealing with memory management, interprocess (and interprocessor) communication, and other low-level operating-system functions. In today’s rapidly changing hardware environment, this provides a useful layer of insulation between the operating system and the underlying hardware.

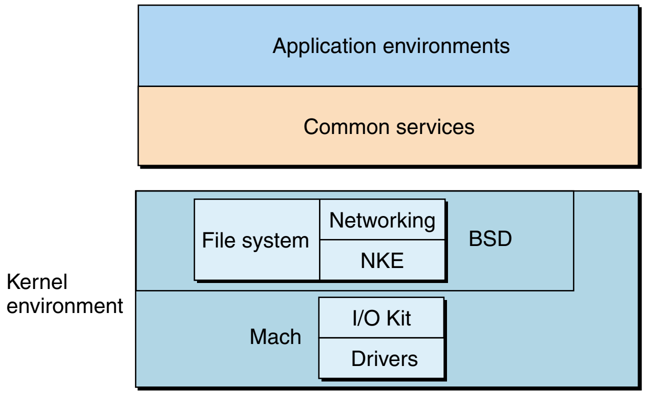
BSD is a carefully engineered, mature operating system with many capabilities. In fact, most of today’s commercial Linux, UNIX, and other similar operating systems contain a great deal of BSD code. BSD also provides a set of industry-standard APIs.

New technologies, such as the I/O Kit and Network Kernel Extensions (NKEs), have been designed and engineered by Apple to take advantage of advanced capabilities, such as those provided by an object-oriented programming model. Mac OS X combines these new technologies with time-tested industry standards to create an operating system that is stable, reliable, flexible, and extensible.

Architecture

The foundation layer of Darwin and Mac OS X is composed of several architectural components, as shown in Figure 2-3. Taken together, these components form the **kernel environment** or simply, the kernel.

Figure 2-3 Mac OS X kernel architecture



Important
Note that Mac OS X uses the term **kernel** somewhat differently than you may be used to seeing it used.

A kernel, in traditional operating-system terminology, is a small nucleus of software that provides only the minimal facilities necessary for implementing additional operating-system services. — from *The Design and Implementation of the 4.4 BSD Operating System*, McKusick, Bostic, Karels, and Quarterman, 1996

Instead, Mac OS X uses the term kernel to refer to everything that executes in the kernel address space.

The Mac OS X kernel includes Mach, BSD, the I/O Kit, file systems, and networking components. Each of these components is described briefly in the following sections. For further details, refer to the specific component chapters or to the reference material listed in the Bibliography.

Mach

Mach manages processor resources such as CPU usage and memory, handles scheduling, provides memory protection, and provides a messaging-centered infrastructure to the rest of the operating-system layers. The Mach component provides

- untyped **IPC** and **RPC**
- support for **SMP**
- support for **real-time** services
- an**external pager**
- modular architecture
- improved performance

BSD

Above the Mach layer, the BSD layer provides "OS personality" APIs and services. The BSD layer is based on the BSD kernel, primarily **FreeBSD** . The BSD component provides

- file systems
- networking
- basic security policies such as user IDs and permissions
- the system framework – a mechanism for exporting APIs to the application layers
- the BSD process model, including process IDs and signals
- FreeBSD kernel APIs
- many of the **POSIX** APIs
- **Pthreads** (POSIX threads implementation)

I/O Kit

The I/O Kit provides a framework for simplified driver development, supporting many categories of devices.The I/O Kit features an object-oriented I/O architecture implemented in a restricted subset of C++. The I/O Kit framework is both modular and extensible. The I/O Kit component provides

- true plug and play
- dynamic device management
- dynamic (“on-demand”) loading of drivers
- power management for desktop systems as well as portables
- multiprocessor capabilities

Networking

Mac OS X networking takes advantage of BSD’s advanced networking capabilities to provide support for modern features, such as Network Address Translation (**NAT**) and **firewalls**. The networking component provides

- 4.4BSD TCP/IP stack and socket APIs
- support for both IP and AppleTalk
- **multihoming**
- routing
- **multicast** support
- server tuning
- socket-based AppleTalk
- Mac OS Classic support
- Carbonized Open Transport APIs

File Systems

Mac OS X provides support for numerous types of file systems, including **HFS**, **HFS+**, **UFS**, **NFS**, **ISO 9660**, and others. The default file-system type is HFS+; Mac OS X boots (and "roots") from HFS+. Advanced features of Mac OS X file systems include an enhanced Virtual File System (**VFS**) design. VFS provides for a layered architecture (file systems are **stackable**). The file system component provides

- **UTF-8** (Unicode) support
- increased performance

Kernel Extensions

Mac OS X provides a kernel extension mechanism as a means of allowing dynamic loading of pieces of code into the kernel, without the need to recompile. These pieces of code are known generically as **plug-ins** or, in the Mac OS X kernel, as **kernel extensions** or **KEXTs**.

Because KEXTs provide both modularity and dynamic loadability, they are a natural choice for any relatively self-contained service that requires access to kernel internal interfaces. Many of the components of the kernel environment support this extension mechanism, although in different ways.

For example, some of the new networking features involve the use of network kernel extensions (**NKEs**). The ability to dynamically add a new file-system implementation is based on VFS KEXTs. Device drivers and device families in the I/O Kit are implemented using KEXTs. KEXTs make development much easier for developers writing drivers or those writing code to support a new volume format or networking protocol. KEXTs are discussed in more detail in the chapter "Extending the Kernel" (page 59).

C H A P T E R 3

Mach

The fundamental services and primitives of the Mac OS X kernel are based on Mach 3.0. Apple has modified and extended Mach to better meet Mac OS X functional and performance goals.

Mach 3.0 was originally conceived as a simple, extensible, communications microkernel. It is capable of running as a standalone kernel, with other traditional operating-system services such as I/O, file systems, and networking stacks running as user-mode servers.

However, in Mac OS X, Mach is linked with other kernel components into a single kernel address space. This is primarily for performance; it is much faster to make a direct call between linked components than it is to send messages or do **RPCs** between separate tasks. This modular structure results in a more robust and extensible system than a monolithic kernel would allow, without the performance penalty of a pure microkernel.

Thus in Mac OS X, Mach is not primarily a communication hub between clients and servers. Instead, its value consists of its abstractions, its extensibility, and its flexibility. In particular, Mach provides

- object-based APIs with communication channels (efor example, ports) as object references
- highly parallel execution, including preemptively scheduled threads and support for **SMP**
- a flexible scheduling framework, with support for real-time usage
- a complete set of **IPC** primitives, including messaging, **RPC**, synchronization, and notification
- support for large virtual address spaces, shared memory regions, and memory objects backed by persistent store

- proven extensibility and portability, for example across instruction set architectures and in distributed environments
- security and resource management as a fundamental principle of design; all resources are virtualized

Mach Kernel Abstractions

Mach provides a small set of abstractions that have been designed to be both simple and powerful. The main kernel abstractions are

- **Tasks** .The units of resource ownership; each task consists of a virtual address space, a **port right name space**, and a set of **threads**.
- **Threads**. The units of CPU execution.
- **Address space**. In conjunction with memory managers, Mach implements the notion of a sparse virtual address space and shared memory.
- **Memory objects**. The internal units of memory management. Memory objects include named entries and regions; they are representations of potentially-persistent data that may be mapped into address spaces.
- **Ports**. Secure, simplex communication channels, accessible only via send and receive capabilities (rights).
- **IPC**. Message queues, remote procedure calls, notifications, semaphores, and lock sets.
- **Time**. Clocks, timers, and waiting.

At the trap level, the interface to most Mach abstractions consists of messages sent to and from kernel ports representing those objects. The trap-level interfaces (such as `mach_msg_overwrite_trap`) and message formats are themselves abstracted in normal usage by the Mach Interface Generator (**MIG**). MIG is used to compile procedural interfaces to the message-based APIs, based on descriptions of those APIs.

Tasks and Threads

Mac OS X processes and POSIX threads (**PThreads**) are implemented on top of Mach tasks and threads, respectively. A thread is the point of control. A task exists to provide resources for the threads it contains. This split is made to provide for parallelism and resource sharing.

A thread:

- is a point of control flow in a task
- has access to all of the elements of the containing task
- executes (potentially) in parallel with other threads, even threads within the same task
- has minimal state information, for low overhead

A task:

- is a collection of system resources; these resources, with the exception of the address space, are referenced by ports. These resources may be shared with other tasks if rights to the ports are so distributed.
- provides a large, potentially sparse address space, referenced by machine address; portions of this space may be shared through inheritance or external memory management.
- contains some number of threads.

Note that a task has no life of its own; only threads execute instructions. When it is said that "task Y does X," what is really meant is that "a thread contained within task Y does X."

A task is a fairly expensive entity. It exists to be a collection of resources. All of the threads in a task share everything. Two tasks share nothing without an explicit action (although the action is often simple) and some resources (such as port receive rights) cannot be shared between two tasks at all.

A thread is a fairly lightweight entity. It is fairly cheap to create and has low overhead to operate. This is true because a thread has little state information(mostly its register state); its owning task bears the burden of resource management. On a multiprocessor machine, it is possible for multiple threads in a task to execute in parallel. Even when parallelism is not the goal, multiple threads have an advantage in that each thread can use a synchronous programming style, instead of attempting asynchronous programming with a single thread attempting to provide multiple services.

A thread is the basic computational entity. A thread belongs to one and only one task that defines its virtual address space. To affect the structure of the address space or to reference any resource other than the address space, the thread must execute a special trap instruction that causes the kernel to perform operations on behalf of the thread or to send a message to some agent on behalf of the thread. In general, these traps manipulate resources associated with the task containing the thread. Requests can be made of the kernel to manipulate these entities: to create them, delete them, and affect their state.

Mach provides a flexible framework for thread scheduling policies. Early versions of Mac OS X support both the **time-sharing** and **fixed-priority** policies. A timesharing thread's priority is raised and lowered to balance its resource consumption against other time-sharing threads.

Fixed-priority threads execute for a certain quantum of time, and then are put at the end of the queue of threads of equal priority. Setting a fixed priority thread's quantum level to infinity allows the thread to run until it blocks, or until it is preempted by a thread of higher priority. High priority real-time threads are usually fixed priority.

Future versions of Mac OS X may have additional scheduling policies, for more sophisticated real-time support.

Ports, Port Rights, Port Sets, and Port Name Spaces

With the exception of the task's virtual address space, all other Mach resources are accessed through a level of indirection known as a **port**. A port is an endpoint of a unidirectional communication channel between a client who requests a service and a server who provides the service. If a reply is to be provided to such a service request, a second port must be used.

In most cases, the resource that is accessed by the port (that is, named by it) is referred to as an object. Most objects named by a port have a single receiver and (potentially) multiple senders. That is, there is exactly one receive port, and at least one sending port, for a typical object such as a message queue.

The service to be provided by an object is determined by the manager that receives the request sent to the object. It follows that the receiver for ports associated with kernel-provided objects is the kernel and the receiver for ports associated with task-provided objects is the task providing that object.

For ports that name task-provided objects, it is possible to change the receiver of requests for that port to a different task, for example by passing the port to that task in a message. A single task may have multiple ports that refer to resources it supports. For that matter, any given entity can have multiple ports that represent it, each implying different sets of permissible operations. For example, many objects have a **name port** and a **control port** (sometimes called the privileged port). Access to the control port allows the object to be manipulated; access to the name port simply names the object, for example, to return information about it.

Tasks have permissions to access ports in certain ways (send, receive, send-once); these are called **port rights**. A port can be accessed only via a right. Ports are often used to grant clients access to objects within Mach. Having the right to send to the object's IPC port denotes the right to manipulate the object in prescribed ways. As such, port right ownership is the fundamental security mechanism within Mach. Having a right to an object is to have a capability to access or manipulate that object.

Port rights can be copied and moved between tasks via IPC. Doing so, in effect, passes capabilities to some object or server.

One type of object referred to by a port is a **port set**. As the name suggests, a port set is a set of port rights that can be treated as a single unit when receiving a message or event from any of the members of the set. Port sets permit one thread to wait on a number of message and event sources, for example in **work loops**.

Traditionally in Mach, the communication channel denoted by a port was always a queue of **messages**. However, Mac OS X supports additional types of communication channels, and these new types of IPC object are also represented by ports and port rights. See the section, "Task to Task Communication (IPC)" (page 28), for more details about messages and other IPC types.

Ports and port rights do not have systemwide names that allow arbitrary ports or rights to be manipulated directly. Ports can be manipulated by a task only if the task has a port right in its port name space. A port right is specified by a **port name**, an integer index into a 32-bit port name space. Each task has associated with it a single port name space.

Tasks acquire port rights when another task explicitly inserts them into its name space, when they receive rights in messages, by creating objects that return a right to the object, and via Mach calls for certain special ports (`mach_thread_self`, `mach_task_self`, and `mach_reply_port`.)

Memory Management

As with most modern operating systems, Mach provides addressing to large, sparse, virtual address spaces. Runtime access is made via virtual addresses that may not correspond to locations in physical memory at the initial time of the attempted access. Mach is responsible for reconciling a requested access in virtual space with a location in physical memory. It does so through demand paging.

A range of a virtual address space is populated with data when a memory object is mapped into that range. All data in an address space is ultimately provided through memory objects. Mach asks the owner of a memory object (a **pager**) for the contents of a page when establishing it in physical memory and returns the possibly modified data to the pager before reclaiming the page. Mac OS X includes two built-in pagers—the **default pager** and the **vnode pager**.

The default pager handles nonpersistent memory, known as **anonymous memory**. Anonymous memory is zero-initialized, and it exists only during the life of a task. The vnode pager maps files into memory objects. Mach exports an interface to memory objects to allow their contents to be contributed by user-mode tasks. This interface is known as the External Memory Management Interface, or **EMMI**.

The memory management subsystem exports virtual memory handles known as **named memory entries**. Like most kernel resources, these are denoted by ports. Having a named memory entry handle allows the owner to map the underlying virtual memory object or to pass the right to map the underlying object to others. Mapping a named entry in two different tasks results in a shared memory window between the two tasks, thus providing a flexible method for establishing shared memory.

Address ranges of virtual memory space may also be populated through direct allocation (using `vm_allocate`). The underlying virtual memory object is anonymous and backed by the default pager. Shared ranges of an address space may also be set up via inheritance. When new tasks are created, they are cloned from a parent. This cloning pertains to the underlying memory address space as well. Mapped portions of objects may be inherited as a copy, or as shared, or not at all, based on attributes associated with the mappings. Mach practices a form of delayed copy known as **copy-on-write** to optimize the performance of inherited copies on task creation.

Rather than directly copying the range, a copy-on-write optimization is accomplished by protected sharing. The two tasks share the memory to be copied, but with read-only access. When either task attempts to modify a portion of the range, that portion is copied at that time. This lazy evaluation of memory copies is an important optimization that permits simplifications in several areas, notably the messaging APIs.

One other form of sharing is provided by Mach, through the export of **named regions**. A named region is a form of a named entry, but instead of being backed by a virtual memory object, it is backed by a virtual map fragment. This fragment may hold mappings to numerous virtual memory objects. It is mappable into other virtual maps, providing a way of inheriting not only a group of virtual memory objects but also their existing mapping relationships. This feature offers significant optimization in task setup, for example when sharing a complex region of the address space used for shared libraries.

Task to Task Communication (IPC)

Communication between tasks is an important element of the Mach philosophy. Mach supports a client/server system structure in which tasks (clients) access services by making requests of other tasks (servers) via messages sent over a communication channel.

The endpoints of these communication channels in Mach are called ports, while port rights denote permission to use the channel. The forms of IPC provided by Mach include

- message queues
- semaphores
- notifications
- lock sets
- remote procedure calls (RPCs)

The type of IPC object denoted by the port determines the operations permissible on that port, and how (and whether) data transfer occurs.

Important
The IPC facilities in Mac OS X are in a state of transition. In early versions of the system, not all of these IPC types may be implemented.

There are two fundamentally different Mach APIs for raw manipulation of ports—the `mach_ipc` family and the `mach_msg` family. Within reason, both families may be used with any IPC object; however, the `mach_ipc` calls are preferred in new code. The `mach_ipc` calls maintain state information where appropriate in order to support the notion of a transaction. The `mach_msg` calls are supported for legacy code but deprecated; they are stateless.

IPC Transactions and Event Dispatching

When a thread calls `mach_ipc_dispatch`, it repeatedly processes events coming in on the registered port set. These events could be an argument block from an RPC object (as the results of a client's call), a lock object being taken (as a result of some other thread's releasing the lock), a notification or semaphore being posted, or a message coming in from a traditional message queue.

These events are handled via callouts from `mach_msg_dispatch`. Some events imply a transaction during the lifetime of the callout. In the case of a lock, the state is the ownership of the lock. When the callout returns, the lock is released. In the case of remote procedure calls, the state is the client's identity, the argument block, and the reply port. When the callout returns, the reply is sent.

When the callout returns, the transaction (if any) is completed, and the thread waits for the next event. The `mach_ipc_dispatch` facility is intended to support work loops.

Message Queues

Originally, the sole style of interprocess communication in Mach was the message queue. Only one task can hold the receive right for a port denoting a message queue. This one task is allowed to receive (read) messages from the port queue. Multiple tasks can hold rights to the port that allow them to send (write) messages into the queue.

A task communicates with another task by building a data structure that contains a set of data elements and then performing a message-send operation on a port for which it holds send rights. At some later time, the task with receive rights to that port will perform a message-receive operation.

A message may consist of some or all of the following:

- pure data
- copies of memory ranges
- port rights
- kernel implicit attributes, such as the sender's security token

The message transfer is an asynchronous operation. The message is logically copied into the receiving task, possibly with copy-on-write optimizations. Multiple threads within the receiving task can be attempting to receive messages from a given port, but only one thread can receive any given message.

Semaphores

Semaphore IPC objects support wait, post, and post all operations. These are counting semaphores, in that posts are saved (counted) if there are no current waiting threads. A post all operation wakes up all currently waiting threads. There is no data associated with a semaphore.

Notifications

Like semaphores, notification objects also support post and wait operations, but with the addition of a state field. The state is a fixed-size, fixed-format field that is defined when the notification object is created. Each post updates the state field; there is a single state, that is overwritten by each post.

Locks

A lock is a **mutex**. The primary interfaces to locks are transaction oriented (see "IPC Transactions and Event Dispatching" (page 29)). During the transaction, the thread holds the lock. When it returns from the transaction, the lock is released. There is no data associated with the lock.

Remote Procedure Calls (RPCs)

As the name implies, an RPC object is designed to facilitate and optimize remote procedure calls. The primary interfaces to RPC objects are transaction oriented (see "IPC Transactions and Event Dispatching" (page 29))

When an RPC object is created, a set of argument block formats is defined. When an RPC (a send on the object) is made by a client, it causes a message in one of the predefined formats to be created and queued on the object, then eventually passed to the server (the receiver). When the server returns from the transaction, the reply is returned to the sender. Mach tries to optimize the transaction by executing the server using the client's resources; this is called **thread migration**.

Time Management

The traditional abstraction of time in Mach is the clock, which provides a set of asynchronous alarm services based on `mach_timespec_t`. There are one or more clock objects, each defining a monotonically increasing time value expressed in nanoseconds. The real-time clock is built in, and is the most important, but there may be other clocks for other notions of time in the system. Clocks support operations to get the current time, sleep for a given period, set an alarm (a notification that is sent at a given time), and so forth.

The `mach_timespec_t` APIs are deprecated in Mac OS X. The newer and preferred APIs are based on timer objects, that in turn use `AbsoluteTime` as the basic data type. `AbsoluteTime` is a machine-dependent type, typically based on the platform-native time base. Routines are provided to convert `AbsoluteTime` values to and from other data types, such as nanoseconds. Timer objects support asynchronous, drift-free notification, cancellation, and premature alarms. They are more efficient and permit higher resolution than clocks.

Important
As with several other Mach services, time management is in a state of transition in Mac OS X. Early versions of the system may not implement timer objects.

C H A P T E R 4

BSD

The BSD portion of the Mac OS X kernel is derived from FreeBSD, a version of 4.4BSD that offers advanced networking, performance, security, and compatibility features. Specifically, the BSD layer is based on the 4.4BSD-Lite2 release from Computer Systems Research Group (CSRG) at the University of California at Berkeley. BSD provides many advanced features, including these:

- Preemptive multitasking with dynamic priority adjustment. Smooth and fair sharing of the computer between applications and users is ensured, even under the heaviest of loads.
- Multiuser access. Many people can use a Mac OS X system simultaneously for a variety of things. This means, for example, that system peripherals such as printers and disk drives are properly shared between all users on the system or the network and that individual resource limits can be placed on users or groups of users, protecting critical system resources from overuse.
- Strong TCP/IP networking with support for industry standards such as SLIP, PPP, NFS, DHCP, and NIS. Mac OS X can interoperate easily with other systems as well as act as an enterprise server, providing vital functions such as NFS (remote file access) and email services, or Internet services such as HTTP, FTP, routing, and firewall (security) services.
- Memory protection. Applications cannot interfere with each other. One application crashing does not affect others in any way.
- Virtual memory and dynamic memory allocation. Applications with large appetites for memory are satisfied while still maintaining interactive response to users. With the virtual memory system in Mac OS X, each application has access to its own 4 GB memory address space; this should satisfy even the most memory-hungry applications.

- Support for kernel threads, based on Mach threads. User-level threading packages are implemented on top of kernel threads. Each kernel thread is an independently scheduled entity. When a thread from a user process blocks in a system call, other threads from the same process can continue to execute on that or other processors. By default, a process in the conventional sense has one thread, the **main thread**. A user process can use the POSIX thread API to create other user threads.
- SMP support. Support is included for machines with multiple CPUs.
- Source code. Developers gain the greatest degree of control over the BSD programming environment because source is included.
- Many of the POSIX APIs.

BSD Facilities

The facilities that are available to a user process are logically divided into two parts: kernel facilities directly implemented by code running in the operating system, and system facilities implemented either by the system, or in cooperation with a server process.

The facilities implemented in the kernel define the **virtual machine** in which each process runs. Like many real machines, this virtual machine has memory management, an interrupt facility, timers, and counters.

The virtual machine also allows access to files and other objects through a set of descriptors. Each descriptor resembles a device controller and supports a set of operations. Like devices on real machines, some of which are internal to the machine and some of which are external, parts of the descriptor machinery are built into the operating system, while other parts are often implemented in server processes.

The BSD component provides the following kernel facilities

- processes and protection
    - host and process identifiers
    - process creation and termination

  - user and group IDs
  - process groups
- memory management
  - text, data, stack, and dynamic shared libraries
  - mapping pages
  - page protection control
  - synchronization primitives
- signals
  - signal types
  - signal handlers
  - sending signals
- timing and statistics
  - real time
  - interval time
- descriptors
  - fles
  - pipes
  - sockets
  - POSIX shared memory
  - POSIX semaphores
- resource controls
  - process priorities
  - resource utilization and resource limits
  - quotas

- system operation support
  - bootstrap operations
  - shut-down operations
  - accounting

BSD system facilities (facilities that may interact with user space) include

- generic input/output operations such as read and write, nonblocking and asynchronous operations
- file system operations
- interprocess communication
- handling of terminals and other devices
- process control
- networking operations

Differences between Mac OS X and BSD

Although the BSD portion of Mac OS X is primarily derived from FreeBSD, some changes have been made.

- The `sbrk()` system call for memory management has not been implemented in Mac OS X.
- The Mac OS X runtime model supports dynamic shared libraries. This model uses **Mach-O** and **PEF** binary file formats; the dynamic link editor (**dyld**) and the Code Fragment Manager (CFM) use these formats respectively. The kernel supports `execve()` with Mach-O binaries. Mapping and management of Mach-O dynamic shared libraries, as well as launching of PEF-based applications, are performed by user-space code.
- Mac OS X does not support memory-mapped devices through the `mmap()` API.

- The `swapon()` call is not supported; `macx_swapon()` is the equivalent call from the Mach pager.
- The Unified Buffer Cache implementation in Mac OS X differs from that found in FreeBSD.

In addition, several new features have been added that are specific to the Mac OS X (Darwin) implementation of BSD. These features are not found in FreeBSD.

- enhancements to file -system buffer cache and file I/O clustering
  - adaptive and speculative read ahead
  - user process controlled read ahead
  - time aging of the file-system buffer cache
- ehancements to file -system support
  - implementation of Apple extensions for ISO-9660 file systems
  - multi-threaded asynchronous I/O for NFS
  - addition of system calls to support semantics of Mac OS Extended file systems
  - additions to naming conventions for pathnames, as required for accessing multiple forks in Mac OS Extended file systems

For Further Reading

The BSD component of the Mac OS X kernel is complex. A complete description is beyond the scope of this document. However, many excellent references exist for this component. If you are interested in BSD, be sure to refer to the Bibliography for further information.

Although the BSD layer of Mac OS X is derived from 4.4BSD, keep in mind that it is not identical to 4.4BSD. Some functionality of 4.4 BSD has not been included in Mac OS X. Some new functionality has been added. The cited reference materials are recommended for additional reading. However, they should *not* be presumed as forming a definitive description of Mac OS X.

C H A P T E R 5

Device Drivers and the I/O Kit

Those of you who are already familiar with writing device drivers for Mac OS 8 and 9 or for BSD will discover that writing drivers for Mac OS X requires some new ways of thinking. In creating Mac OS X, Apple has completely redesigned the Macintosh I/O architecture, providing a framework for simplified driver development, supporting many categories of devices. This framework is called the **I/O Kit**.

The I/O Kit uses an object-oriented programming model, implemented in a restricted subset of C++. Use of object-oriented frameworks can dramatically increase developer productivity. Once you are familiar with the new model, you should find that it makes writing device drivers easier and more efficient than ever before.

From a programming perspective, the I/O Kit provides an abstract view of the system hardware to the upper layers of Mac OS X. By starting with properly designed base classes, you gain a head start in writing a new driver; with much of the driver code already written, the you need only to fill in the specific code that makes your driver different.

Part of the philosophy of the I/O Kit is to make the design completely open. Rather than hiding APIs in an attempt to protect developers from themselves, all of the I/O Kit source is available as part of Darwin. You can use the source code as an aid to designing (and debugging) new drivers.

Instead of hiding the interfaces, Apple's designers have chosen to lead by example. Sample code and classes show the recommended (easy) way to write a driver. However, developers are not prevented from doing things the hard way (or the wrong way). Instead, attention has been concentrated on making the "best" ways easy to follow.

Redesigning the I/O Model

You might ask why Apple chose to redesign the I/O model. At first glance, it might seem that reusing the model from Mac OS 9 or FreeBSD would have been an easier choice. There are several reasons for the decision, however.

Neither the Mac OS 8 and 9 driver model nor the FreeBSD model offered a sufficient feature set to meet the needs of Mac OS X. The underlying operating-system technology of Mac OS X is very different from that of Mac OS 8 and 9. The Mac OS X kernel is significantly more advanced than the previous Mac OS system architecture; Mac OS X needs to handle memory protection, preemption, multiprocessing, and other features not present in previous versions of Mac OS. Although FreeBSD is capable of handling these features, the BSD model did not offer the automatic configuration, stacking, power management, or dynamic device-loading features required in a modern, consumer-oriented operating system.

By redesigning the I/O architecture, Apple's engineers can take best advantage of the operating-system features in Mac OS X. For example, virtual memory (VM) is not a fundamental part of the operating system in Mac OS 8 and 9. Thus, every driver writer must know about (and write for) VM.This has presented certain complications for developers. In contrast, Mac OS X has simplified driver interaction with VM. VM capability is inherent in the Mac OS X operating system and cannot be turned off by the user. Thus, VM capabilities can be abstracted into the I/O Kit, and the code for handling VM need not be written for every driver.

Mac OS X offers an unprecedented opportunity to take advantage of hardware complexity without the requirement of encoding software complexity into each new device driver. Under Mac OS 9, for example, all software development kits (SDKs) were independent of each other, duplicating functionality between them. In Mac OS X, the I/O Kit is delivered as part of the single kernel development kit (KDK); all portions of the KDK rely on common underpinnings.

In contrast with traditional I/O models, the reusable code model provided by the I/O Kit can decrease your development work substantially. In porting drivers from Mac OS 9, for example, the Mac OS X counterparts have been up to 75% smaller.

In general, all hardware support is provided directly by I/O Kit entities. One exception to this rule is imaging devices such as printers, scanners, and digital cameras (although these do make some use of I/O Kit functionality). Specifically, although communication with these devices is handled by the I/O Kit (for instance, under the Firewire or USB families), support for particular device characteristics is handled by user-space code (see "Accessing Kernel APIs from User Space" (page 45) for further discussion). If you need to support imaging devices, you should employ the appropriate imaging software development kit (SDK).

The I/O Kit attempts to represent, in software, the same hierarchy that exists in hardware. Some things are difficult to abstract, however. When the hardware hierarchy is difficult to represent (for example, if layering violations occur), then the I/O Kit abstractions provide less help for writing drivers.

In addition, all drivers exist to drive hardware; all hardware is different. Even with the reusable model provided by the I/O Kit, you still need to be aware of any hardware quirks. The code to support those quirks still needs to be unique from driver to driver.

Although most developers should be able to take full advantage of I/O Kit device families (see "Families" (page 42)), there will occasionally be some who cannot. Even those developers should be able to make use of parts of the I/O Kit, however. In any case, the source code is always available. Developers who need to do so can replace functionality and modify the classes themselves.

In designing the I/O Kit, one goal has been to make developers' lives easier. Unfortunately, it is not possible to make all developers' lives uniformly easy. Therefore, a second goal of the I/O Kit design is to meet the needs of the majority of developers, without getting in the way of the minority that need lower level access to the hardware.

I/O Kit Architecture

The I/O Kit provides a model of system hardware in an object-oriented framework. Each type of service or device is represented by a C++ class; each discrete service or device is represented by an instance (object) of that class.

There are three major conceptual elements of the I/O Kit architecture:

- Families
- Drivers
- Nubs

Families

A **family** defines a collection of software abstractions that are common to all devices of a particular category. These abstractions are implemented in C code and C++ classes. Families may include headers, libraries, sample code, test harnesses, and documentation. If it seems more familiar, however, you can think of a family simply as a library.

Families define and implement the abstractions that are common to all devices of a particular category. They provide the APIs, generic support code, and at least one example driver (in the documentation).

Families provide services for many different categories of devices. For example, there are protocol families (such as SCSI, USB, and Firewire), storage families (disk), network families, and families to describe human interface devices (mouse, and keyboard). When devices have features in common, the software that supports those features is most likely found in a family.

Common abstractions are defined and implemented by the family, allowing all drivers in a family to share similar features easily. For example, all SCSI controllers have certain things they must do, such as scanning the SCSI bus. The SCSI family defines and implements the functionality that is common to SCSI controllers. Because this functionality has been included in the SCSI family,you do not need to include scanning code (for example) in your new SCSI controller driver.

Instead, you can concentrate on device-specific details that make your driver different from other SCSI drivers. The use of families means there is less code for a developer to write.

Families are dynamically loadable; they are loaded when needed and unloaded when no longer needed. Although some common families may be preloaded at system startup, all families should be considered to be dynamically loadable (and, therefore, potentially unloaded). See the “Connection Example” (page 44) for an illustration.

Drivers

A **driver** is an I/O Kit object that manages a specific piece of hardware, implementing the appropriate I/O Kit abstractions for controlling that hardware. When a driver is loaded, its required families are also loaded to provide necessary, common functionality. The request to load a driver causes all of its dependent requirements (and their requirements) to be loaded first. After all requirements are met, the requested driver is loaded as well. See the "Connection Example" (page 44) for an illustration.

Note that families are loaded upon demand of the driver, not the other way around. Occasionally, a family may already be loaded when a driver demands it; however, you should never assume this. To ensure that all requirements are met, each device driver should list all of its requirements in its **property list**.

Each driver is in a client-provider relationship, wherein every driver must know about both the family it inherits from and the family it connects to. A SCSI *controller* driver, for example, must be able to communicate with both the SCSI family and the PCI family (as a client of PCI and provider of SCSI). A SCSI *disk* driver communicates with both the SCSI and storage families.

Nubs

A **nub** is an I/O Kit object that represents a detected, controllable entity; that is, a nub represents a device or logical service. For example, a nub may represent a bus, a disk, a disk partition, a graphics adaptor, a keyboard, or any number of similar entities.

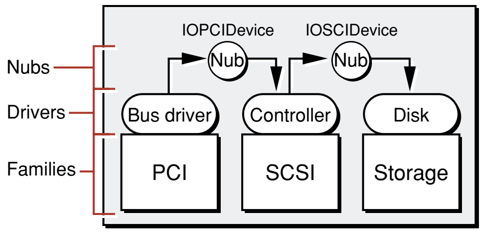
A nub is loaded as part of the family that instantiates it. Each nub provides access to the device or service that it represents and provides services such as matching, arbitration, and power management.

For example, nubs match devices to drivers. Each nub provides a bridge between two drivers (and, by extension, between two families). It is most common that a driver publishes one nub for each individual device or service it controls; however, it is also possible for a driver that controls only a single device or service to act as its own nub. See the "Connection Example" (page 44) for an illustration.

Connection Example

Figure 5-1 illustrates the I/O Kit architecture, using several example drivers and their corresponding families and nubs. Note that many different family and driver combinations are possible; this diagram shows only one possibility. Arrows represent order of creation or discovery.

Figure 5-1 I/O Kit Architecture example: families, drivers, and nubs



This example illustrates how a SCSI disk driver (Storage family) is connected to the PCI bus. The connection is made in several steps.

1. The PCI Bbs driver discovers a PCI device and announces its presence by creating a nub (`IOPCIDevice`). The nub's class is defined by the PCI family.
2. The nub identifies (matches) the correct device driver and requests that the driver be loaded. At the end of this matching process, a SCSI controller driver has been found and loaded. Loading the controller driver causes all required families to be loaded as well. In this case, the SCSI family is loaded; the PCI family (also required) is already present. The SCSI controller driver is given a reference to the `IOPCIDevice` nub.
3. The SCSI controller driver scans the SCSI bus for devices. Upon finding a device, it announces the presence of the device by creating a nub (`IOSCSIDevice`). The class of this nub is defined by the SCSI family.
4. The nub identifies (matches) the correct device driver and requests that the driver be loaded. At the end of this matching process, a disk driver has been found and loaded. Loading the disk driver causes all required families to be loaded as well. In this case, the Storage family is loaded; the SCSI family (also required) is already present. The disk driver is given a reference to the `IOSCSIDevice` nub.

Accessing Kernel APIs from User Space

Mac OS X draws a distinction between kernel space and user space. Applications in user space cannot interface directly with kernel-space APIs.

Some family services are never exported to user space; these services are available only inside the kernel. One such example is the PCI family. For stability and security reasons, direct access to PCI resources from user space is forbidden.

In other cases, however, family services may need to be accessed from user space. For example, a game may need to interact with system software to set monitor depth or sound volume. As another example, a disk backup program may need to act as the “driver” for a tape drive. Other examples of user applications that may need to interact with nubs in kernel space might include those running scanners, printers, digital cameras, and so forth.

User Client Access

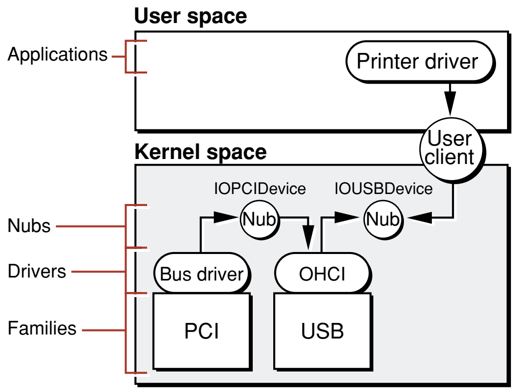
Many types of I/O Kit devices are made accessible across the user-kernel address space boundary by means of a **user client**. A user client is implemented in two parts and has a presence in both user and kernel space. The kernel portion is usually part of an appropriate family. The user portion is linked into the application as a library or Core Foundation plug-in (CFPlugin). A user client looks like a library when viewed from user space. From kernel space, it looks like a driver.

The user client handles negotiation, protection, authentication, and other tasks in user space as if it were an in-kernel driver. An application can communicate with a device by acquiring the device nub through an appropriate user client. The user client attaches to the (kernel-space) nub on behalf of the (user-space) application.

Figure 5-2 illustrates one example of a user client, in this case, a USB printer application (**OHCI** is the standard USB controller interface). The printer “driver” is in user space (recall that printer support is outside of the kernel). A user client permits the communication of raw USB commands across the user-kernel address space boundary.

Note that many different family and driver combinations are possible; this diagram shows only one possibility. Arrows represent order of creation or discovery. See "I/O Kit Architecture" (page 41) for a description of families, drivers, and nubs, as well as an explanation of their connection path.

Figure 5-2 Interaction with I/O Kit from user space



Many families already provide the necessary functionality for creating user clients. If you develop custom drivers that do not use I/O Kit families, however, you will need to write additional code. Any code that communicates between user space and kernel space must use of one or more of the following facilities available in Mac OS X:

- BSD system calls
- Mach IPC
- Mach shared memory

The I/O Kit uses primarily Mach IPC and Mach shared memory. In contrast, the networking and file system components of Mac OS X use primarily BSD system calls.

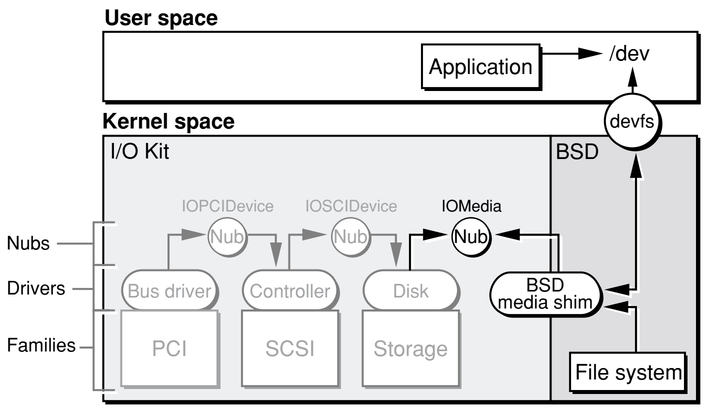
BSD Media Shim

The **BSD media shim** provides another way in which kernel APIs are exported into user space. The BSD media shim is not a user client; it is implemented entirely in kernel code. It provides a connection between a disk driver and BSD, by way of a nub (`IOMedia`) created by the disk driver. Although the BSD disk shim is included in the storage family, it does not inherit directly from the storage family.

The BSD media shim uses BSD system calls to provide user-space applications with access to disks by way of BSD-style device nodes (in the `/dev` directory). These device nodes are owned and managed by the device file system (`devfs`), a BSD analogue to an I/O Kit user client. The device file system uses a file-system model to represent devices rather than files.

Figure 5-3 shows the BSD media shim providing a connection for a SCSI disk. Note that many different family and driver combinations are possible; this diagram shows only one possibility. Arrows represent order of creation or discovery. See "I/O Kit Architecture" (page 41) for a description of families, drivers, and nubs, as well as an explanation of their connection path.

Figure 5-3 The BSD media shim providing user-space access to disks



Networking and Network Kernel Extensions

Network kernel extensions (NKEs) represent a specific case of a Mac OS X kernel extension. NKEs provide a way to extend and modify the networking infrastructure of Mac OS X dynamically, without recompiling or relinking the kernel. The effect is immediate and does not require rebooting the system.

Much of the content of this chapter has been excerpted from Chapter 1 of *Inside Mac OS X: Network Kernel Extensions*. For further information on to this topic, you should refer to that book.

NKEs can be used to

- monitor network traffic
- modify network traffic
- receive notification of asynchronous events from the driver layer

In the last case, such events are received by the data link and network layers. Examples of these events include power management events and interface status changes. See Figure 6-1 (page 51) for an illustration of the data link and network layers.

Specifically, NKEs allow you to

- create protocol stacks that can be loaded and unloaded dynamically and configured automatically
- create modules that can be loaded and unloaded dynamically at specific positions in the network hierarchy.

The Kernel Extension Manager dynamically adds NKEs to the running Mac OS X kernel inside the kernel's address space. An installed and enabled NKE is invoked automatically, depending on its position in the sequence of protocol components, to process an incoming or outgoing packet.

All NKEs provide initialization and termination routines that the Kernel Extension Manager invokes when it loads or unloads the NKE. The initialization routine handles any operations that are needed to complete the incorporation of the NKEs into the kernel, such as updating `protosw` and `domain` structures. Similarly, the termination routine must remove references to the NKE from these structures to unload itself successfully. NKEs must provide a mechanism, such as a reference count, to ensure that the NKE can terminate without leaving dangling pointers.

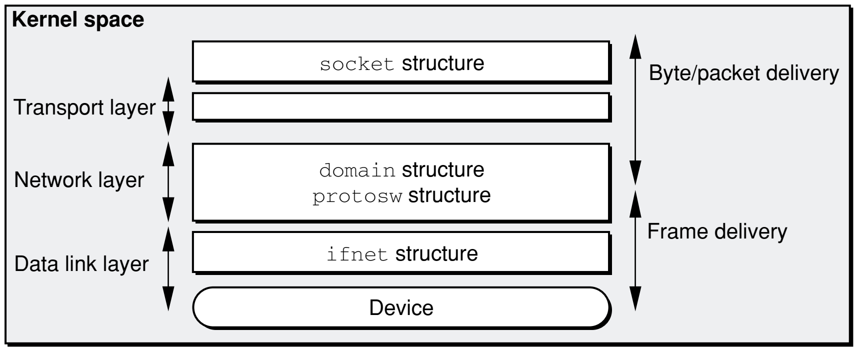
Review of 4.4BSD Network Architecture

Mac OS X is based on the 4.4BSD operating system. The following structures control the 4.4BSD network architecture:

- `socket` structure—used to keep track of network information on a per-file descriptor basis. The `socket` structure is referenced by file descriptors from user space.
- `domain` structure—describes protocol families.
- `protosw` structure—describes protocol handlers. (A protocol handler is the implementation of a particular protocol in a protocol family.)
- `ifnet` structure—describes a network interface.

None of these structures is used uniformly throughout the 4.4BSD networking infrastructure. Instead, each structure is used at a specific level, as shown in Figure 6-1.

Figure 6-1 4.4 BSD network architecture



Above the network layer, packets are isolated on a per-user (per-file descriptor) basis. That is, packets are isolated based upon their ownership. Below the network layer, packets are isolated based on which device they go to (or originate from). The network layer provides a transition in how packets are viewed and processed. In the protocol stack (network layer) and the data link layer, the point of view is per-packet. Above these, in the socket structure, the point of view is the stream.

NKE Types

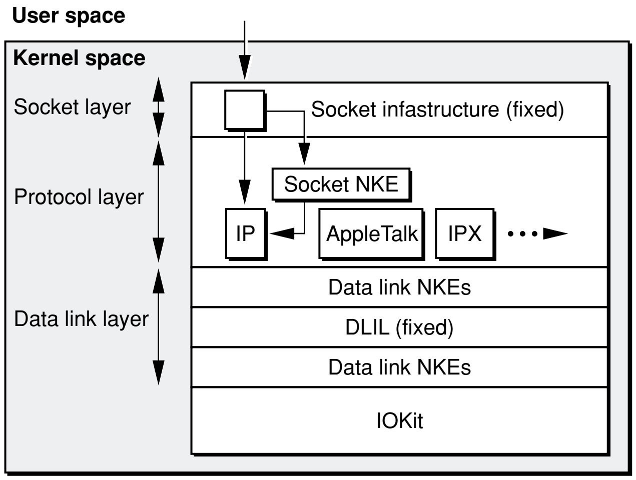
Making the 4.4BSD network architecture dynamically extensible requires several NKE types, for use at specific places in the kernel.

- Socket NKEs —which reside between the socket layer and the transport protocol handlers and are invoked through a `protosw` structure. Socket NKEs use a new set of dispatch vectors that intercept specific socket and socket buffer utility functions.
- Protocol family NKEs— which are collections of protocols that share a common addressing structure. Internally, a `domain` structure and a chain of `protosw` structures describe each protocol.

- Protocol handler NKEs—which process packets for a particular protocol within the context of a protocol family. A `protosw` structure describes a protocol handler and provides the mechanism by which the handler is invoked to process incoming and outgoing packets and for invoking various control functions.
- Data link NKEs—which are inserted below the protocol layer and above the network interface layer. This type of NKE can passively observe traffic as it flows in and out of the system (for example, a sniffer) or can modify the traffic (for example, by encryptiion or address translation).

Figure 6-2 summarizes the NKE architecture.

Figure 6-2 NKE architecture



Socket NKEs operate in one of two modes: programmatic or global. Data link NKEs operate only in global mode.

A programmatic NKE is a socket NKE that is enabled under program control, using socket options, for a specific socket. That is, a program is responsible for enabling these on a specific socket. Programmatic NKEs must be specified by a name (a 32-bit integer handle); these should be registered with Apple. NKE handles use the same **name space** as type and creator handles.

In contrast, global socket NKEs as well as data link NKEs are automatically enabled when they are loaded and initialized. The developer (or application) need not know the names of the global NKEs that are enabled.

Modifications to 4.4BSD Networking Architecture

To support NKEs in Mac OS X, the 4.4BSD `domain` and `protosw` structures were modified as follows:

- The `protosw` array referenced by the domain structure is now a linked list, thereby removing the array's upper bound. The new `max_protohdr` member defines the maximum protocol header size for the domain. The new `dom_refs` member is a reference count that is incremented when a new socket for this address family is created and is decremented when a socket for this address family is closed.
- The `protosw` structure is no longer an array. The pr_next member has been added to link the structures together. This change has implications for `protox` usage for `AF_INET` and `AF_ISO` input packet processing. The `pr_flags` member is an unsigned integer instead of a short. NKE hooks have been added to link NKE descriptors together.

C H A P T E R 7

File Systems and VFS Stacks

Mac OS X provides "out-of-the-box" support for several different file systems. These include Mac OS Extended format (**HFS+**), the BSD standard file system format (**UFS**), **NFS** (an industry standard for networked file systems), and ISO 9660 (used for CD-ROM).

Support is also included for reading the older, Mac OS Standard format (**HFS**) file-system type; however, you should not plan to format new volumes using Mac OS Standard format. Mac OS X cannot boot from these file systems, nor does the Mac OS Standard format provide some of the information required by Mac OS X.

Mac OS X boots and "roots" from Mac OS Extended format. That is, Mac OS X can mount a Mac OS Extended Format volume and use it as the primary, or root, file system. The Mac OS Extended format provides many of the same characteristics as Mac OS Standard format but adds additional support for modern features such as file permissions, longer filenames, Unicode, both hard and symbolic links, and larger disk sizes.

Other file systems can be mounted, allowing users to gain access to additional volume formats and features. For example, UFS provides case sensitivity and other characteristics that may be expected by BSD commands. In contrast, Mac OS Extended Format is not case-sensitive (but is case-preserving).

NFS provides access to network servers as if they were locally mounted file systems. The Carbon application environment mimics many expected behaviors of Mac OS Extended format on top of both UFS and NFS. These include such characteristics as Finder Info, file ID access, and aliases.

By using the Mac OS X Virtual File System (VFS) capability and writing kernel extensions, you can add support for other file systems. Examples of file systems that are not currently supported in Mac OS X but that you may wish to add to the system include the Andrew file system (AFS) and the Windows NT file system (NTFS). If you want to support a new volume format or networking protocol, you'll need to write a file-system kernel extension.

Working With the File System

In Mac OS X, the **vnode** structure provides the internal representation of a file or directory (folder). There is a unique vnode allocated for each active file or folder, including the root.

Within a file system, operations on specific files and directories are implemented via vnodes and **VOP** (vnode operation) calls. VOP calls are used for operations on individual files or directories (such as open, close, read, or write). Examples include `VOP_OPEN` to open a file and `VOP_READ` to read file contents. In contrast, file-system-wide operations are implemented using VFS calls. VFS calls are primarily used for operations on entire file systems; examples include `VFS_MOUNT` and `VFS_UNMOUNT` to mount or unmount a file system, respectively. File-system writers need to provide stubs for each of these sets of calls.

Supporting a new volume format requires implementing a new file-system type. However, it is not always necessary to implement a new file-system type in order to change the way in which a user interacts with files. VFS stacks allow developers to create and layer new capabilities onto an existing file-system type.

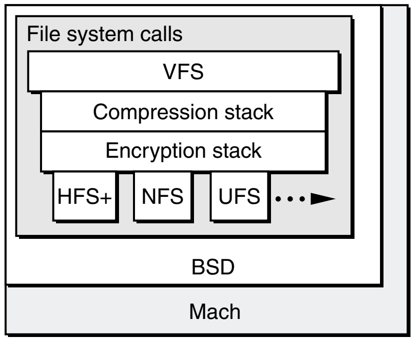
VFS stacks provide filters between the user and the underlying file system. As implied by Figure 7-1 (page 57), VFS stacks can run on top of any type of file system. If your application does not need to support a volume format or networking protocol, but does need to intercept data going into or out of the file system, implementing a VFS stack may be the appropriate choice.

For example, VFS stacks may be used in the following sorts of application areas:

- virus checking —automatically check a file for viruses before reading its data
- compression—perform compression or decompression on-the-fly when opening (reading) and writing files
- encryption—automatically encrypt a file as it is written, then decrypt it (with a password) when it is opened or read

Figure 7-1 illustrates the file system architecture with several example VFS stacks and file systems shown.

Figure 7-1 File systems and VFS stacks



When writing a VFS stack, you must create a stub for each vnode operation. In some cases, the stub simply calls the routine of the same name in the underlying layer. Note that stacks may be implemented directly on top of a file system or on top of other stacks, so you cannot be sure exactly what the underlying layer will do with a given VFS or VOP call.

In other cases, such as when creating, reading, or writing a file, your VFS stack will intercept a call rather than simply passing it to the underlying layer. For example, an encryption stack would intercept read and write calls in order to add encryption or decryption filters. A virus-checking stack might intercept the open and read calls.

VFS stacks are KEXTs. The Kernel Extension Manager dynamically adds VFS stacks and support for additional file-system types to the running Mac OS X kernel as part of the kernel's address space. An installed and enabled file system can be mounted automatically or manually. Further file access goes through that file system's calls.

A Politically Correct Example

The Politically Correct File System is an example of a VFS stack. In this example, all calls are ignored (passed to the underlying layer) except for those that create, read, or write a file (or folder).

Upon receiving a request to create a file or folder, the Politically Correct (PC) stack intercepts the call before it can be executed by the underlying file system. The PC version of the create call checks the requested filename against a table of names. If the name is deemed politically incorrect, for example if the user chooses to name a file "vulgarity", the PC `create` call chooses a more pleasing name, for example, "politeness". The new name is passed to the `create` routine of the underlying file system.

Similarly, when a user opens a file to read or write it, such as with a text editor, the PC `read` and `write` routines first examine the data buffer, possibly substituting preferred words and phrases for their undesirable counterparts. After the substitutions are made, the buffer is handed to the underlying routine, which displays the data or writes it to disk.

Thus, if a user attempted to save a file containing a sentence such as this:

The beleaguered computer company's woes continue, despite rising stock prices.

the PC `write` routine might intercept and filter this sentence to a more desirable version:

The aspiring computer company's joys continue, due to rising stock prices.

C H A P T E R 8

Extending the Kernel

As discussed in the chapter "Kernel Architecture" (page 13), Mac OS X provides a kernel extension mechanism as a means of allowing dynamic loading of code into the kernel, without the need to recompile or relink. Because these kernel extensions (KEXTs) provide both modularity and dynamic loadability, they are a natural choice for any relatively self-contained service that requires access to internal kernel interfaces.

Because KEXTs run in supervisor mode in the kernel's address space, they are also harder to write and debug than user-level modules, and must conform to strict guidelines. Further, kernel resources are wired (permanently resident in memory) and are thus more costly to use than resources in a user-space task of equivalent functionality.

In addition, although memory protection keeps applications from crashing the system, no such safeguards are in place inside the kernel. A badly behaved kernel extension in Mac OS X can actually cause more trouble than a badly behaved application or extension could in Mac OS 8 or 9.

Bugs in KEXTs can have far more severe consequences than bugs in user-level code. For example, a memory access error in a user application can, at worst, cause that application to crash. In contrast, a memory access error in a KEXT causes a system **panic**, crashing the operating system.

Finally, for security reasons, some customers restrict or don't permit the use of third-party KEXTs. As a result, use of KEXTs is strongly discouraged in situations where user-level solutions are feasible. Mac OS X guarantees that user threads are just as efficient as kernel threads, so efficiency should not be an issue. Unless your application requires low-level access to kernel interfaces or the data stream, you should use a higher level of abstraction when developing code for Mac OS X.

When you are trying to determine if a piece of code should be a KEXT, the default answer is generally *no*. In particular, if your code was a system extension in Mac OS 8 or 9, that does not imply that it must necessarily be a kernel extension in Mac OS X. There are only a few good reasons for a developer to write a kernel extension:

- Your code needs to take a primary interrupt, that is, something in the hardware needs to interrupt the CPU.
- The primary client of your code is inside the kernel, for example, a block device whose primary client is a file system.
- A sufficiently large number of running applications require a resource that your code provides; for example, you have written a file-system stack.
- Your code needs to multiplex between multiple client applications that require high speed, excellent synchronization, or low latency.

If your code does not meet any of the above criteria, you should consider developing it as a library or a user-level daemon, or using one of the user-level plug-in architectures (such as QuickTime components or the Core Graphics framework) instead of writing a kernel extension.

If you are writing device drivers or code to support a new volume format or networking protocol, however, KEXTs may be the only feasible solution. Fortunately, while KEXTs may be more difficult to write than user-space code, several tools and procedures are available to enhance the development and debugging process. See “Debugging Your KEXT” (page 63) for more information.

This chapter provides a conceptual overview of KEXTs and how to create them. If you are interested in building a simple KEXT, see the Apple tutorials listed in the Bibliography. These provide step-by-step instructions for creating a simple, generic KEXT or a basic I/O Kit driver.

KEXT Implementation

KEXTs are implemented as **bundles**, folders that the Finder treats as single files. See the chapter about bundles in *Inside Mac OS X: System Overview* for a discussion of bundles.The KEXT bundle can contain the following:

- **Information property list** — text file that describes the contents, settings, and requirements of the KEXT. This file is required. A KEXT bundle need contain nothing more than this file, although most KEXTs contain one or more kernel modules as well. See the chapter about software configuration in *Inside Mac OS X: System Overview* for further information about property lists.
- **Kernel module**— a file in Mach-O format, containing the actual binary code used by the KEXT. A kernel module (or **KMOD**) represents the minimum unit of code that can be loaded into the kernel. A KEXT usually contains one KMOD. If no KMODs are included, the information property list file must contain a reference to a module in another KEXT and change its default settings.
- **Resources**— for example, icons or localization dictionaries. Resources are optional; they may be useful for KEXTs that need to display a dialog or menu. At present, no resources are explicitly defined for use with KEXTs.

KMOD Dependencies

Any KMOD can declare that it is dependent upon any other KMOD. The developer lists these dependencies in the “Requires” field of the module’s property list file.

Before a KMOD is loaded, all of its requirements are checked. Those required modules (and their requirements) aree loaded first, iterating back through the lists until there are no more required modules to load. Only after all requirements are met, is the requested KMOD loaded as well.

For example, device drivers (a type of KEXT) are dependent upon (require) certain families (another type of KEXT). When a driver is loaded, its required families are also loaded to provide necessary, common functionality. To ensure that all requirements are met, each device driver should list all of its requirements (families and other drivers) in its property list. See the chapter "Device Drivers and the I/O Kit" (page 39), for an explanation of drivers and families.

It is important to list all dependencies for each KMOD. If your KEXT fails to do so, your KMOD may not load due to unrecognized symbols, thus rendering the KEXT useless. Dependencies in KMODs can be considered analogous to required header files or libraries in code development; in fact, the Kernel Extension Manager uses the standard linker to resolve KMOD requirements.

Building and Testing Your KEXT

After creating the necessary property list and C (or C++) source files, you use Project Builder to build your KEXT as well. Any errors in the source code are brought to your attention during the build and you are given the chance to edit your source files and try again.

To test your KEXT, however, you need to leave Project Builder and work in the Terminal application (or in **console** mode). In console mode, all system messages are written directly to your screen, as well as to a log file (`/var/log/system.log`). If you work in the Terminal application, you must view system messages in the log file.You also need to log in to the **root** account (or use the `su` command), since only the root account can load kernel extensions.

When testing your KEXT, you can load and unload it manually, as well as check the load status. You can use the `kextload` command to load any KEXT. This command handles matching for I/O Kit drivers, then calls `kmodload`. If you are not working with the I/O Kit you can run `kmodload` directly. Manual pages for these, as well as the `kmodunload` and `kmodstat` commands, are included in Mac OS X.

Note that these commands are only useful when developing a KEXT. Eventually, after it has been tested and debugged, you install your KEXT in one of the standard places (see "Installed KEXTs" (page 64) for details). Then, it will be loaded and unloaded automatically at system startup and shutdown or whenever it is needed (such as when a new device is detected).

Debugging Your KEXT

KEXT debugging can be complicated. Before you can debug a KEXT, you must first enable kernel debugging, as Mac OS X is not normally configured to permit debugging the kernel. Only the root account can enable kernel debugging, and you need to reboot Mac OS X for the changes to take effect.

Kernel debugging is performed using two Mac OS X machines, called the development machine and target machine. These machines must be connected over a reliable network connection on the same subnet (or within a single local network). Specifically, there must not be any intervening IP routers or other devices that could make hardware-based Ethernet addressing impossible.

The KEXT is registered (and the KMODs loaded and run) on the target machine. The debugger is launched and run on the development machine. You can also rebuild your KEXT on the development machine, after you fix any errors you find.

Debugging must be performed in this fashion because you must temporarily halt the kernel on the target machine in order to use the debugger. When you halt the kernel, all other processes on that machine stop. However, a debugger running remotely can continue to run and can continue to examine (or modify) the kernel on the target machine.

Note that bugs in KEXTs may cause the target kernel to freeze or panic. If this happens, you may not be able to continue debugging, even over a remote connection; you have to reboot the target and start over, setting a breakpoint just before the code where the KEXT crashed and working very carefully up to the crash point.

KEXTs are debugged using **GDB**, a source-level debugger with a command-line interface. You will need to work in the Terminal application to run GDB. For detailed information about using GDB, see the documentation included with Mac OS X. You can also use the `help` command from within GDB.

Because KEXT debugging happens at such a low level, you won't be able to take advantage of all features of GDB. For example:

- You can't use GDB to call a function or method in a KEXT.
- You can't use GDB to debug interrupt routines.

Use care that you do not halt the kernel for too long when you are debugging (for example, when you set breakpoints). In a short time, internal inconsistencies can appear that cause the target kernel to panic or freeze, forcing you to reboot the target machine.

Installed KEXTs

The Kernel Extension Manager (KEXT Manager) is responsible for loading and unloading all installed KMODs (commands such as `kextload` are used only during development). Installed KMODs are dynamically added to the running Mac OS X kernel as part of the kernel's address space. An installed and enabled KMOD is invoked as needed.

Important
Note that KEXTs are only wrappers (bundles) around a property list, KMODs (or references to KMODs), and optional resources. The KEXT describes what is to be loaded; it is the KMODs that are actually loaded.

KEXTs are usually installed in the Extensions folder (at `/System/Libraries/Extensions`.) The Kernel Extension Manager (in the form of a **daemon**, `kextd)`, always checks here. KEXTs can also be installed in several other locations:

- in ROM
- in the Driver partition on a disk
- inside an application bundle

The last location allows an application to register KEXTs without the need to install them permanently elsewhere within the system hierarchy. This may be more convenient and allows the KMOD to be associated with a specific, running application. When it starts, the application can call the Kernel Extension Manager and register a KEXT.

For example, a network packet sniffer application might employ a Network Kernel Extension (NKE). A tape backup application would require that a tape driver be loaded during the duration of the backup process. When the application exits, the kernel extension is no longer needed and can be unloaded.

Note that, although the application is responsible for registering the KEXT, this is no guarantee that the corresponding KMODs are actually ever loaded. It is still up to a kernel component, such as the I/O Kit, to determine a need, such as matching a piece of hardware to a desired driver, and tell the KEXT Manager to load the appropriate KMODs (and their dependencies).

C H A P T E R 9

Kernel Services

In a typical preemptive multitasking operating system such as Mac OS X, FreeBSD, or Linux, user applications are not allowed direct access to shared resources such as RAM, disks, printers, and other devices. Instead, the kernel provides controlled access to these resources, and can thus be viewed as a service provider.

Recall that each application exists in its own (user) address space and that the kernel exists in a separate (kernel) address space. Privileged operations, such as opening a file, initiating network traffic, or shutting down the computer, are performed in kernel space and are thus available only to the kernel.

Applications that need to have privileged operations performed must request the appropriate services from the kernel. The kernel provides these operations as services to the processes, mapping any associated parameters in and out of user space.

Application processes include applications that are explicitly launched and run by the user, as well as various system processes, such as **daemons**, that keep the system running smoothly.

Any code to communicate between user space and kernel space must take advantage of one or more of the following facilities available in Mac OS X:

- BSD system calls
- Mach IPC
- shared memory

In Mac OS X, where the kernel itself is modular, interaction between the various kernel components is also in the form of services. Each component, such as Mach, networking, or the file system, is therefore both a provider of services to applications and other components as well as a client of kernel services itself. Kernel space, however, is a single address space; memory is shared between kernel components. Thus, kernel components are able to communicate more freely with each other than with applications in user space.

Available Services

Most of the commonly-used kernel services are described below. For each service, the provider component is named as well as the client components. A brief description is also given. For more complete information, see the available documentation for the component itself.

In the API listings below, header files are listed as they would be included in real code. The default compiler flags should locate the correct file in the "well known places".

In addition, the following header files are assumed to be included at all times:

```
#include <sys/param.h>      /* useful defines and limits */
#include <sys/types.h>      /* exported data types */
#include <sys/systm.h>      /* "systm" and NOT "system"; prototypes */
#include <libkern/libkern.h>    /* more prototypes */
```

BSD Media Shim

Provider:	I/O Kit
Clients:	BSD, File systems

The BSD media shim uses BSD system calls and the I/O Kit user client facility to export device driver interfaces into user space as BSD-style device nodes in the `/dev` directory. The BSD media shim also communicates with the file system and VFS stacks. Support for user processes is provided via `devfs`.

Device Driver Management

Provider:	I/O Kit
Clients:	I/O Kit KEXTs (families and drivers), user processes
APIs:	

```
IOKit/IOfamily/*
```

These services support device driver instantiation, matching, service notification. Family APIs publish services; drivers use devices.

Events

Provider:	Mach
Clients:	All kernel components, user processes

Specific services include port notifications, notification ports, and notification events.

Exceptions, Traps

Provider:	Mach
Clients:	BSD, user processes

This service supports BSD signals, interrupts, and debugging, as well as various system calls that can be accessed by user processes.

Families

Provider:	I/O Kit
Clients:	BSD, file systems, networking, user processes
APIs:	

```
IOKit/IOfamily/*
```

This service provides APIs for I/O Kit families, including support for networking, block, graphics, FireWire, USB, human interface, and many other device categories.

File Descriptor Management

Provider:	BSD
Clients:	file systems, networking, user processes
APIs:	

```
#include <sys/filedesc.h>
#include <sys/file.h>
```

File descriptors provide per-process unique, nonnegative integers that are used to identify an open file (or socket). For user processes, all interaction with files is done via file descriptors. File descriptors are also used for access and manipulation of POSIX semaphores and POSIX shared memory.

Host Manipulation and Inquiry

Provider:	Mach
Clients:	All kernel components

These services are used to get and set host-based information, such as page size and processor count.

Interprocess Communication (IPC)

Provider:	Mach
Clients:	All kernel components

This service provides various specialized forms of communication between tasks (processes) on the local machine. The particular form of IPC in use dictates how (and whether) data is processed. Specific services include: send and receive operations, as well as primitives for servicing ports and/or port sets. See also: "Port Right Management" (page 72), "Task and Thread Management" (page 75), "Memory and Address Space Management" (page 72), and "Synchronization Primitives (Low Level)" (page 74).

Kernel Loadable Module Support

Provider:	I/O Kit
Clients:	file systems, networking, loadable modules

Tools: See the man pages for the following utility programs:

```
kextload
kextunload
kmodload
kmodstat
kmodunload
```

This service provides support for loading and unloading KEXTs.

Kernel Tracing

Provider: BSD

Clients: Mach, I/O Kit, file systems, networking, loadable modules

APIs:

```
#include <sys/kdebug.h>
```

This service provides information for performance analysis and debugging support, as well as trace points for user processes.

Lock Management

Provider: BSD

Clients: file systems, networking, loadable modules, user processes

APIs:

```
#include <sys/lock.h>
```

BSD, file systems, and networking code should use this service for management of locking operations. Note that this API is quite different from the one defined in `osfmk/kern/lock.h`.

Mach Interface Generator (MIG)

Provider: Mach

Clients: All kernel components, user processes (Project Builder)

MIG is used to specify IPC formats that are valid on a given port. It is used mostly in Remote Procedure Call (RPC) situations, but supports other forms of communication as well. MIG also provides a set of runtime services for dispatching incoming communications to the appropriate handler. Project Builder has special rules and targets for generating stubs for both sides of the MIG interface.

mbuf Management

Provider:	Networking
Clients:	NKEs (third-party), file systems
APIs:	

```
#include <sys/mbuf.h>
```

These services provide support for the `mbuf` data structure, which is used to manage I/O for network devices.

Memory and Address Space Management

Provider:	Mach
Clients:	All kernel components, user processes

Specific services include virtual memory management, address space allocation, page read and write, external memory managers (EMMI), and memory objects.

Port Right Management

Provider:	Mach
Clients:	All kernel components

Port right ownership is the fundamental security mechanism within Mach. Specific services include creation and destruction, reference management, copying, explicit insertion and removal from other tasks, passing via IPC, grouping of rights into sets, and requesting asynchronous notifications about changes in a port's status. See also: "Task and Thread Management" (page 75), "Interprocess Communication (IPC)" (page 70).

Processor Management

Provider: Mach
Clients: All kernel components

These services provide low-level hardware support, including processor start, processor stop, and power management.

Registry

Provider: I/O Kit
Clients: I/O Kit family APIs, user processes.
APIs:

```
#include <IOKit/IORegistryEntry.h>
```

These services support publishing of I/O Kit devices or services and device information and relationships.

Queue Management

Provider: BSD
Clients: file systems, networking, loadable modules, user processes
APIs:

```
#include <sys/queue.h>
```

BSD, file systems, and networking code use this service for queue management. It provides support for singly and doubly linked lists and queues. Note that there are subtle differences between this API and the queues found in `osfmk/kern/queue.h`.

Socket Management

Provider: Networking
Clients: NKEs (third-party), file systems, user processes
APIs:

```
#include <sys/socket.h>
#include <sys/socketvar.h>
```

These services provide support for the management of *sockets*.

Network Kernel Extension Support

Provider: Networking
Clients: NKEs (third-party)
APIs:

```
#include <net/kext_net.h>
```

These services provide general support for network kernel extensions.

Scheduling

Provider: Mach
Clients: BSD

Specific services include priority-based thread scheduling, preemption, and processor resource allocation, based on the following policies: time-sharing, round-robin, and *FIFO* fixed priority.

Synchronization Primitives (Low Level)

Provider: Mach
Clients: All kernel components, IPC services for exporting to user space.

This Mach service provides low-level implementation support for basic asynchronous primitives (wait queues, semaphores) as well as basic locking primitives (machine-specific locks, spin locks, mutexes, shared/exclusive read/write locks).

Synchronization Primitives

Provider: BSD
Clients: File systems, Networking, loadable modules, user processes
APIs:

```
#include <sys/proc.h>
#include <machine/spl.h>
```

This BSD service provides higher level support for `sleep()` and `wakeup()` calls as well as *SPLs*.

sysctl

Provider:	BSD
Clients:	File systems, networking, loadable modules
APIs:	

```
#include <sys/sysctl.h>
```

This service provides a formalized interface for kernel global manipulation and tuning.

Task and Thread Management

Provider:	Mach
Clients:	BSD

This service provides the underlying implementation for BSD process management; a process is based on one Mach task and one or more Mach threads. A task is the unit of resource ownership. A thread is an independently schedulable execution path.

Timing Services

Provider:	Mach
Clients:	BSD, user processes

The kernel provides several different timing services to user processes. Timing services support profiling, statistics gathering, and various types of timers, as well as current date and time-of-day functionality.

VFS Infrastructure

Provider:	BSD
Clients:	file system

APIs:

```
#include <sys/buf.h>
#include <sys/namei.h>
#include <sys/vnode.h>
#include <sys/vnode_if.h>
#include <vfs/vfs_support.h>
```

This service provides VFS management routines and default library routines in support of virtual file system functionality.

Vnode Management

Provider:	BSD
Clients:	file system
APIs:	

```
#include <sys/vnode.h>
```

This service provides allocation, referencing, and serialization functionality in support of vnode management.

Zone allocator

Provider:	Mach
Clients:	BSD, networking

This service provides support for efficient kernel memory allocation.

Glossary

abstraction The process of picking out (abstracting) common features of objects and procedures. In programming, an abstraction provides the API that all instances of a thing conform to; the abstraction defines shared features. In the I/O Kit, abstractions take the form of classes.

address space The address space of a process describes the ranges of memory (both physical and virtual) that it uses while running. In Mac OS X, processes do not share address space.

anonymous memory Virtual memory backed by the default pager to swap files, rather than by a persistent object. Anonymous memory is zero-initialized and exists only for the life of the task. See also **default pager**, **task**

API (Application programming interface) The interface (calling conventions) by which an application program accesses operating system services.

Apple Public Source License Apple's Open Source license, available at `http://www.apple.com/publicsource`. Darwin is distributed under this license. See also **Open Source**

AppleTalk A suite of network protocols that is standard on Macintosh computers.

ASCII (American Standard Code for Information Interchange) A 7-bit character set (commonly represented using 8 bits) that defines 128 unique character codes. See also **Unicode**

BSD (Berkeley Software Distribution. Formerly known as the Berkeley version of UNIX, BSD is now simply called the BSD operating system. The BSD portion of the Mac OS X kernel is based on FreeBSD, a version of BSD.

BSD media shim Specifically `IOMediaBSDClient`; part of the I/O Kit storage family. The BSD media shim provides access to all storage devices being managed by I/O Kit drivers via traditional BSD device nodes.

bundle A packaging mechanism (implemented as a directory) that stores executable code and the software resources related to that code. Applications, plug-ins, and frameworks represent types of bundles. Except for frameworks, bundles are file packages, presented by the Finder as a single file.

Carbon An application environment in Mac OS X that features a set of programming interfaces derived from earlier versions of the Mac OS. The Carbon APIs have been modified to work properly with Mac OS X, especially with the foundation of the operating system, the kernel environment.

Carbon applications can run on Mac OS X, Mac OS 9, and all versions of Mac OS 8 later than Mac OS 8.1.

Classic An application environment in Mac OS X that lets users run non-Carbon legacy Mac OS software. It supports programs built for both Power PC and 68K processor architectures and is fully integrated with the Finder and the other application environments.

clock An object used to abstract time in Mach.

Cocoa An advanced object-oriented development platform on Mac OS X. Cocoa is a set of frameworks with programming interfaces in both Java and Objective-C. It is based on the integration of OPENSTEP, Apple technologies, and Java.

condition variable A type of variable provided by the POSIX threads functions to help synchronize the threads in a task.

console A special window that displays system log messages, as well as output written to the standard error and standard output streams by applications launched from the Finder. Also, an application by the same name that displays this information.

control port In Mach, access to the control port allows an object to be manipulated. Also called the privileged port. See also **port**; **name port**

cooperative multitasking A multitasking environment in which a running program can receive processing time only if other programs allow it; each application must give up control of the processor cooperatively in order to allow others to run. Mac OS 8 and 9 are cooperative multitasking environments. See also **preemptive multitasking**

copy-on-write A delayed copy optimization used in Mach. The object to be copied is write protected instead, and physically copied only if some thread tries to write to it. See also **thread**.

Darwin An Open Source project that includes the Darwin kernel, the BSD commands and C libraries, and several additional features.The Darwin kernel is synonymous with the Mac OS X kernel.

daemon A long-lived process, usually without a visible user interface, that performs a system-related service. Daemons are usually spawned automatically by the system and may either live forever or be regenerated at intervals.

default pager In Mach, one of two built-in pagers. The default pager handles nonpersistent (anonymous) memory. See also **anonymous memory**; **vnode pager**; **pager**

demand paging An operating-system facility that brings pages of data from disk into physical memory only as they are needed.

DLIL (Data Link Interface Layer) The part of the Mac OS X kernel's networking infrastructure that provides the interface between protocol handling and network device drivers in the I/O Kit. A generalization of the BSD "ifnet" architecture.

DMA Direct memory access; a means of transferring data between host memory and a peripheral device without involving the host processor.

driver Software that deals with getting data to and from a device, as well as control of that device. In the I/O Kit, an object that manages a piece of hardware (a device), implementing the appropriate I/O Kit abstractions for that device. See also **object**

DVD (Digital Versatile Disc) Originally, Digital Video Disc. An optical storage medium that provides greater capacity and bandwidth than CD-ROM; DVDs are frequently used for multimedia as well as data storage.

dyld (Dynamic link editor) A utility that allows programs to dynamically load (and link to) needed functions.

EMMI (External Memory Management Interface) Mach's interface to memory objects that allows their contents to be contributed by user-mode tasks. See also **external pager**

Ethernet A high-speed local area network technology.

exception An interruption to the normal flow of program control, caused by the program itself or by executing an illegal instruction.

exception port A Mach port on which a task or thread receives messages when exceptions occur.

external pager A module that manages the relationship between virtual memory and its backing store. External pagers are clients of Mach's EMMI. They may be either in the kernel or in user space. The built-in pagers in Mac OS X are the default pager and the vnode pager. See also **EMMI**

family In the I/O Kit, a family defines a collection of software abstractions that are common to all devices of a particular category (for example, PCI, storage, USB). Families provide functionality and services to drivers. See also **driver**

FAT (File Allocation Table) A data structure used in the MS-DOS file system. Also synonymous with the file system that uses it. The FAT file system is also used as part of Microsoft Windows and has been adopted for use inside devices such as digital cameras.

fat files Mach-O files containing object code for more than one machine architecture.

FIFO (First-in First-out) A data processing scheme in which data is read in the order in which it was written, processes are run in the order in which they were scheduled, and so forth.

file descriptor A per-process unique, nonnegative integer used to identify an open file (or socket).

firewall Software (or a computer running such software) that prevents unauthorized access to a network by users outside of the network.

fixed-priority policy In Mach, a scheduling policy in which threads execute for a certain quantum of time, and then are put at the end of the queue of threads of equal priority.

fork A stream of data that can be opened and accessed individually under a common filename. The Macintosh Standard and Extended file systems store a separate "data" fork and a "resource" fork as part of every file; data in each fork can be accessed and manipulated independently of the other. Also, in BSD, `fork` is a system call that creates a new process.

framework A basic structure that holds the parts of some thing together. In Mac OS X, specifically, a bundle containing a dynamic shared library and associated resources, including image files, header files, and documentation. Also used to describe the barrier between user and system functions.

FreeBSD A variant of the BSD operating system. See `http://www.freebsd.org` for details.

GDB (GNU Debugger) GDB is a powerful, source-level debugger with a command line interface. GDB is a popular Open Source debugger and is included with the Mac OS X developer tools.

host The computer that's running (is host to) a particular program. The term is usually used to refer to a computer on a network.

host processor The microprocessor on which an application program resides. When an application is running, the host processor may call other, peripheral microprocessors, such as a digital signal processor, to perform specialized operations.

HFS (Hierarchical file system) The Mac OS Standard filesystem format, used to represent a collection of files as a hierarchy of directories (folders), each of which may contain either files or folders themselves.

HFS+ (Hierarchical file system plus) The Mac OS Extended file system format. This filesystem format was introduced as part of Mac OS 8.1, adding support for filenames longer than 31 characters, Unicode representation of file and directory names, and efficient operation on very large disks.

IDE (Interactive development environment or integrated development environment) An application or set of tools that allows a programmer to write, compile, edit, and perhaps test and debug within an integrated, interactive environment.

inheritance attribute In Mach, a value indicating the degree to which a parent process and its child process share the parent process's address space. A memory page can be inherited copy-on-write, shared, or not at all.

in-line data Data that's included directly in a Mach message, as opposed to referred to by a pointer. See also **out-of-line data**

I/O (Input/Output) The sending and retrieving of information into the memory of a program, usually to and from a file or a peripheral device.

I/O Kit Apple's object-oriented I/O development model. The I/O Kit provides a framework for simplified driver development, supporting many families of devices. See also **family**.

Info Plist See **Information property list**

information property list A special form of property list with predefined keys for specifying basic bundle attributes and information of interest, such as supported document types and offered services. See also **bundle**; **property list**

IPC (Inter-process communication) The transfer of information between processes.

Kerberos An authentication system based on symmetric key cryptography. Used in MIT Project Athena and adopted by the Open Software Foundation (OSF).

kernel The complete Mac OS X core operating system environment that includes Mach, BSD, the I/O Kit, file systems, and networking components.

kernel extension See **KEXT**

kernel port A Mach port whose receive right is held by the kernel. See also **task port**; **thread port**

KEXT (kernel extension) Kernel extensions extend the functionality of the kernel. The I/O Kit, File system, and Networking components are designed to allow and expect the creation and use of KEXTs.

KMOD (kernel module) A file (or files) in Mach-O format, containing the actual binary code used by a KEXT. A KMOD is the minimum unit of code that can be loaded into the kernel. See also **KEXT**, **Mach-O**.

Mach The lowest level of the Mac OS X kernel. Mach provides such basic services and abstractions as threads, tasks, ports, IPC, scheduling, physical and virtual address space management, VM, and timers.

Mach-O Mach object file format. The preferred object file format for Mac OS X.

Mach factor A measurement of how busy a Mach-based system (such as Mac OS X) is. Unlike a load average (as used in Linux or BSD systems), higher Mach factors mean the system is less busy.

Mach server A task that provides services to clients, using a MIG-generated RPC interface. See also **MIG**

main thread By default, a process has one thread, the main thread. If a process has multiple threads, the main thread is the first thread in the process. A user process can use the POSIX thread API to create other user threads.

makefile A makefile details the files, dependencies, and rules by which an executable application is built or by which a set of programs may be run.

memory-mapped files A facility that maps virtual memory onto a physical file. Thereafter, any access to that part of virtual memory causes the corresponding page of the physical file to be accessed. The contents of the file can be changed by changing the contents in memory.

memory object An object managed by a pager, that represents a file (for example) in memory. See also **pager**

memory protection A system of memory management in which programs are prevented from being able to modify or corrupt the memory partition of another program. Mac OS 8 and 9 do not have memory protection; Mac OS X does.

message In Mach, a message consists of a header and a variable-length body; some operating-system services are invoked by passing a message from a thread to the Mach port representing the task that provides the desired service.

microkernel A kernel implementing a minimal set of abstractions. Typically, higher-level OS services such as file systems and device drivers are implemented in layers above a microkernel, possibly in trusted user-mode servers. See also **monolithic kernel**

MIG (Message interface generator)) MIG provides a procedure call interface to Mach's system of interprocess messaging.

monolithic kernel A kernel architecture in which all pieces of the kernel are closely intertwined. A monolithic kernel provides substantial performance improvements; however, it is difficult to evolve the individual components independently. The Mac OS X kernel is a hybrid of the monolithic and microkernel models. See also **microkernel**

multicast A process in which a single packet can be addressed to multiple recipients. Multicast is used, for example, in streaming video, in which many megabytes of data are sent over the network.

multihoming The ability to have multiple network addresses in one computer. For example, multihoming might be used to create a system in which one address is used to talk to hosts outside a firewall and the other to talk to hosts inside; the computer provides facilities for passing information between the two.

multitasking Describes an operating system that allows the concurrent execution of multiple programs. Mac OS X uses preemptive multitasking. Mac OS 8 and 9 use cooperative multitasking.

mutex (Mutual exclusion variable) A type of variable provided by the POSIX threads functions to help protect critical regions in a multiple-thread task.

name port In Mach, access to the name port names the object. See also **port**; **control port**

name space An agreed-upon context in which names (identifiers) can be defined. Within a given name space, all names must be unique.

named memory entry A handle (a port) to a mapable object backed by a memory manager. The object can be a region or a memory object.

named region In Mach, a form of named memory entry that provides a form of memory sharing.

NAT (Network address translation) A scheme that transforms network packets at a gateway so network addresses that are valid on one side of the gateway are translated into addresses that are valid on the other side.

network A group of hosts that can directly communicate with each other.

NFS (Network file system) An NFS file server allows users on the network to share files as if the files were on the user's local disk.

NKE (Network kernel extension) NKEs provide a way to extend and modify the networking infrastructure of Mac OS X dynamically, without recompiling or relinking the kernel. The effect is immediate and does not require rebooting the system.

NMI (Nonmaskable interrupt) An interrupt produced by a particular keyboard sequence or button. It can be used to interrupt a hung system.

notify port A Mach port on which a task receives messages from the kernel advising it of changes in port access rights and of the status of messages it has sent.

nonsimple message In Mach, a message that contains either a reference to a port or a pointer to data. See also **simple message**.

nub An I/O Kit object that represents a device or logical service. Each nub provides access to the device or service it represents, and provides such services as matching, arbitration, and power management. It is most common that a driver publishes one nub for each individual device or service it controls; it is possible for a driver that vends only a single device or service to act as its own nub.

NVRAM (Nonvolatile RAM) RAM storage that retains its state even when the power is off. See also **RAM**

object In object-oriented programming, an instance of a class.

OHCI (Open Host Controller Interface) The register-level standard that is used by most USB controller chips.

Open Transport A communications architecture for implementing network protocols and other communication features on computers running Mac OS. Open Transport provides a set of programming interfaces that supports, among other things, both the AppleTalk and TCP/IP protocols.

Open Source A definition of software that includes freely available access to source code, redistribution, modification, and derived works. The full definition is available at `http://www.opensource.org`.

out-of-line data Data that's passed by reference in a Mach message, as opposed to being included in the message. See also **in-line data**.

packet An individual piece of information sent on a network.

page The unit of measurement used to divide memory.

pager A module responsible for providing the data for the pages of a memory object. See also **default pager**, **vnode pager**.

panic An unrecoverable system failure detected by the kernel.

physical address An address to which a hardware device, such as a memory chip, can directly respond. Programs, including the Mach kernel, use virtual addresses that are translated to physical addresses by mapping hardware controlled by the Mach kernel.

PEF (Preferred Executable Format) The format of executable files used for applications and shared libraries in Mac OS 8 and 9; supported in Mac OS X. See also **Mach-O** (the preferred format for Mac OS X).

POSIX (Portable Operating System Interface) An operating system interface standardization effort supported by ISO/IEC, IEEE, and The Open Group.

port In Mach, a secure unidirectional channel for communication between tasks running on a single system. In IP transport protocols, an integer identifier used to select a receiver for an incoming packet, or to specify the sender of an outgoing packet.

port name In Mach, an integer index into a port name space; a port right is specified by its port name. See also **port rights**.

port rights In Mach, the ability to send to or receive from a Mach port. Also known as port access rights.

port set In Mach, a set of zero or more Mach ports. A thread can receive messages sent to any of the ports contained in a port set by specifying the port set as a parameter to `msg_receive()`.

preemptive multitasking A type of multitasking in which the operating system can interrupt a currently running task in order to run another task, as needed. See also **cooperative multitasking**.

preemption The act of interrupting a currently running program in order to give time to another task.

priority In Mach scheduling, a number between 0 and 127 that indicates how likely a thread is to run. The higher the thread's priority, the more likely the thread is to run. See also **scheduling policy**.

process A BSD abstraction for a running program. A process' resources include a virtual address space, threads, and file descriptors. In Mac OS X, a process is based on one Mach task and one or more Mach threads.

process identifier, or process ID A number that uniquely identifies a process.

protected memory See **memory protection**.

protocol handler A network module that extracts data from input packets (giving the data to interested programs) and inserts data into output packets (giving the output packet to the appropriate network device driver).

programmed I/O I/O in which the CPU accomplishes data transfer with explicit load and store instructions to device registers, as opposed to DMA. Byte-by-byte or word-by-word data transfer to a device. Also known as direct I/O. See also **DMA**.

property list A textual way to represent data. Elements of the property list represent data of certain types, such as arrays, dictionaries, and strings. System routines allow programs to read property lists into memory and convert the textual data representation into "real" data. See also **information property list**.

Pthreads POSIX threads implementation. See also **POSIX**, **threads**

quantum The fixed amount of time a thread or process can run before being preempted.

RAM (Random-access memory) Memory that a microprocessor can either read or write to.

real-time Used to describe a system that must guarantee a response to an external event within a given time. Real-time support is important for applications such as multimedia.

receive rights In Mach, the ability to receive messages on a Mach port. Only one task at a time can have receive rights for any one port. See also **send rights**.

reply port A Mach port associated with a thread that is used in remote procedure calls.

ROM (Read-only memory) , that is, memory that cannot be written to.

root An administrative account with special privileges. For example, only the root account can load kernel extensions. Also, the root file system (the root of the file system inverted tree).

RPC (Remote Procedure Call) In Mach, RPCs are implemented using MIG-generated messages.

SCSI Small Computer Systems Interface. A standard connector and communications protocol used for connecting devices such as disk drives to computers.

scheduling The determination of when each process or task runs, including assignment of start times.

scheduling policy In Mach, a thread's scheduling policy determines how the thread's priority is set and under what circumstances the thread runs. See also **priority**.

send rights In Mach, the ability to send messages to a Mach port. Many tasks can have send rights for the same port. See also **receive rights**

simple message In Mach, a message that contains neither references to ports nor pointers to data. See also **nonsimple message**.

SMP (Symmetric Multi-processing) An operating system architecture in which two or more processors are managed by one kernel, sharing the same memory, having equal access to I/O devices, and in which any task, including kernel tasks, can run on any processor.

SPL Set Priority Level. A request that sets the current processor priority level, the level used by the kernel to control interrupt delivery to the CPU.

socket In BSD-derived systems, a socket refers to different entities in user and kernel space. For a user process, a socket is a file descriptor that has been allocated using `socket(2)`. In the kernel, a socket is the data structure allocated when the kernel's implementation of the `socket(2)` call is made. In AppleTalk protocols, a socket serves the same purpose as a port in IP transport protocols.

stackable file system A filesystem layer that has as its input the standard VFS file system interfaces and that may call other filesystem layers beneath it to implement fil esystem operations. All stackable file systems support the same interface and can be layered on top of one another to add unique functionality.

task A Mach abstraction, consisting of a virtual address space and a port name space. A task itself performs no computation; rather, it is the framework in which threads run. See also **threads**

task port A kernel port that represents a task and is used to manipulate that task. See also **kernel port**, **thread port**.

TCP/IP Transmission Control Protocol/Internet Protocol. An industry standard protocol used to deliver messages between computers over the network. TCP/IP is the primary networking protocol used in Mac OS X.

thread In Mach, the unit of CPU utilization. A thread consists of a program counter, a set of registers, and a stack pointer. See also **task**.

thread port A kernel port that represents a thread and is used to manipulate that thread. See also **kernel port**, **task port**.

thread-safe Used to describe code that can be executed safely by several threads simultaneously.

time-sharing policy In Mach, a scheduling policy in which a thread's priority is raised and lowered to balance its resource consumption against other timesharing threads

UFS (UNIX File system) An industry standard filesystem format used in UNIXand similar operating systems such as BSD. UFS in Mac OS X is a derivative 4.4BSD UFS. Specifically, its disk layout is not compatible with other BSD UFS implementations.

UDF (Universal Disk Format.) The file system format used in DVD disks.

Unicode A 16-bit character set that defines unique character codes for characters in a wide range of languages. Unlike ASCII, which defines 128 distinct characters typically represented in 8 bits, there are as many as 65,536 distinct Unicode characters that represent the unique characters used in most foreign languages.

USB Universal Serial Bus. A multiplatform bus standard that can support up to 127 peripheral devices, including printers, digital cameras, keyboards and mice, and storage devices.

user client In I/O Kit, a means of allowing user-level code to communicate across the user-kernel address space boundary, as, for example, in a printer or scanner application.

UTF-8 (Unicode Transformation Format 8) A format used to represent a sequence of 16-bit Unicode characters with an equivalent sequence of 8-bit characters, none of which are zero. This sequence of characters can be represented using an ordinary C language string.

virtual address An address that is usable by software. Each task has its own range of virtual addresses, beginning at address zero. The Mach operating system makes the CPU hardware map these addresses onto physical memory only when necessary, using disk memory at other times. See also **physical address**.

virtual memory The use of a disk partition or a file on disk to provide the same facilities usually provided by RAM. The virtual memory provides 32 bit (minimum) protected address space for each task and facilitates efficient sharing of that address space.

VFS Virtual File System. A set of standard internal file system interfaces and utilities that facilitate support for additional file systems. VFS provides an infrastructure for file systems built in the kernel.

VM See **virtual memory**.

vnode A data structure containing information about a file.

vnode pager In Mach, one of two built-in pagers. The vnode pager maps files into memory objects. See also **default pager**; **pager**

work loop The main loop of an application or module that waits repeatedly for incoming events and dispatches them.

XML (Extensible Markup Language.) An "extremely simple" dialect of SGML (Standard Generalized Markup Language), XML provides a metalanguage containing rules for constructing specialized markup languages. XML users can create their own tags, making XML very flexible.

Bibliography

Apple Mac OS X Publications

Hello Kernel: Creating a Kernel Extension With Project Builder (tutorial).

Hello IOKit: Creating a Device Driver With Project Builder (tutorial).

Inside Mac OS X: Network Kernel Extensions.

Inside Mac OS X: System Overview.

General UNIX and Open Source Resources

A Quarter Century of Unix. Peter H. Salus. Addison-Wesley, 1994.ISBN 0-201-54777-5.

Berkeley Software Distribution. CSRG, UC Berkeley. USENIX and O'Reilly, 1994. ISBN 1-56592-082-1.

The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary. Eric S. Raymond. O'Reilly & Associates, 1999. ISBN 1-56592-724-9.

The New Hacker's Dictionary. 3rd. Ed., Eric S. Raymond. MIT Press, 1996. ISBN 0-262-68092-0.

Open Sources: Voices from the Open Source Revolution. Edited by Chris DiBona, Sam Ockman & Mark Stone. O'Reilly & Associates, 1999. ISBN 1-56592-582-3.

Proceedings of the First Conference on Freely Redistributable Software. Free Software Foundation. FSF, 1996. ISBN 1-882114-47-7.

The UNIX Desk Reference: The hu.man Pages. Peter Dyson. Sybex, 1996. ISBN 0-7821-1658-2.

The UNIX Programming Environment. Brian W. Kernighan, Rob Pike. Prentice Hall, 1984. ISBN 0-13-937681-X; Osborne, 1996. ISBN 0-07-882189-4.

Internals

Advanced Topics in UNIX: Processes, Files, & Systems. Ronald J. Leach. Wiley, 1996. ISBN 1-57176-159-4.

The Complete FreeBSD. Greg Lehey, Walnut Creek CDROM Books, 1999. ISBN 1-57176-246-9.

The Design and Implementation of the 4.4BSD UNIX Operating System. Marshall Kirk McKusick, et al. Addison-Wesley, 1996. ISBN 0-201-54979-4.

The Design of the UNIX Operating System. Maurice J. Bach. Prentice Hall, 1986. ISBN 0-13-201799-7.

Linux Kernel Internals. Michael Beck, et al. Addison-Wesley, 1996. ISBN 0-201-87741-4.

Lions' Commentary on UNIX 6th Edition with Source Code. John Lions. Peer-to-Peer, 1996. ISBN 1-57398-013-7.

Panic!: UNIX System Crash Dump Analysis. Chris Drake, Kimberly Brown. Prentice Hall, 1995. ISBN 0-13-149386-8.

UNIX Internals: The New Frontiers. Uresh Vahalia. Prentice-Hall, 1995. ISBN 0-13-101908-2.

UNIX Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers. Curt Schimmel. Addison-Wesley, 1994. ISBN 0-201-63338-8.

Optimizing PowerPC Code. Gary Kacmarcik. Addison-Wesley Publishing Company, 1995. ISBN 0-201-40839-2.

Berkeley Software Architecture Manual 4.4BSD Edition. William Joy, Robert Fabry, Samuel Leffler, M. Kirk McKusick, Michael Karels. Computer Systems Research Group, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley.

Mach

CMU Computer Science: A 25th Anniversary Commemorative. Richard F. Rashid, Ed. ACM Press, 1991. ISBN 0-201-52899-1.

Load Distribution: the Implementation of the Mach Microkernal. Dejan S. Milojicic. Vieweg, 1994. ISBN 3-528-05424-7.

Programming under Mach. Boykin, et al. Addison-Wesley, 1993. ISBN 0-201-52739-1.

Mach Workshop Proceedings. USENIX Association. October, 1990.

Mach Symposium Proceedings. USENIX Association. November, 1991.

Mach III Symposium Proceedings. USENIX Association. April, 1993, ISBN 1-880446-49-9.

Mach 3 Documentation Series. Open Group Research Institute (RI):

- *Final Draft Specifications OSF/1 1.3 Engineering Release*. RI. May 1993.
- *OSF Mach Final Draft Kernel Principles*. RI. May, 1993.
- *OSF Mach Final Draft Kernel Interfaces*. RI. May, 1993.
- *OSF Mach Final Draft Server Writer's Guide*. RI. May, 1993.
- *OSF Mach Final Draft Server Library Interfaces*, RI, May, 1993.

Research Institute Microkernel Series. Open Group Research Institute (RI):

- *Operating Systems Collected Papers*. Volume I. RI. March, 1993.
- *Operating Systems Collected Papers*. Volume II. RI. October,1993.
- *Operating Systems Collected Papers*. Volume III. RI. April, 1994.
- *Operating Systems Collected Papers*. Volume IV. RI. October, 1995.

Mach: A New Kernel Foundation for UNIX Development. Proceedings of the Summer 1986 USENIX Conference. Atlanta, GA., http://www.usenix.org.

UNIX as an Application Program. Proceedings of the Summer 1990 USENIX Conference. Anaheim, CA., http://www.usenix.org.

OSF RI papers (Spec '93):

OSF Mach Final Draft Kernel Interfaces.

OSF Mach Final Draft Kernel Principles.

OSF Mach Final Draft Server Library Interfaces.

OSF Mach Final Draft Server Writer's Guide.

OSF Mach Kernel Interface Changes.

OSF RI papers (Spec '94):

OSF RI 1994 Mach Kernel Interfaces Draft.

OSF RI 1994 Mach Kernel Interfaces Draft (Part A).

OSF RI 1994 Mach Kernel Interfaces Draft (Part B).

OSF RI 1994 Mach Kernel Interfaces Draft (Part C).

OSF RI papers (miscellaneous):

Debugging an object oriented system using the Mach interface.

Unix File Access and Caching in a Multicomputer Environment.

Untyped MIG: The Protocol.

Untyped MIG: What Has Changed and Migration Guide.

Towards a World-Wide Civilization of Objects.

A Preemptible Mach Kernel.

A Trusted, Scalable, Real-Time Operating System Environment.

Mach Scheduling Framework.

Networking

UNIX Network Programming. Volume 1, *Networking APIs: Sockets and XTI*. W. Richard Stevens. Prentice Hall, 1998, ISBN 0-13-490012-X.

UNIX Network Programming. Volume 2, *Interprocess Communications*. W. Richard Stevens. Prentice Hall, 1998. ISBN 0-13-081081-9.

TCP/IP Illustrated. Volume 1, *The Protocols*. W. Richard Stevens. Addison-Wesley, 1994. ISBN 0-201-63346-9.

TCP/IP Illustrated. Volume 2, *The Implementation*. W. Richard Stevens. Addison-Wesley, 1995. ISBN 0-201-63354-X.

TCP/IP Illustrated. Volume 3, *TCP for Transactions, HTTP, NNTP, and the UNIX Domain Protocols*. W. Richard Stevens. Addison-Wesley, 1996. ISBN 0-201-63495-3.

Operating Systems

Advanced Computer Architecture: Parallelism, Scalability, Programmability. Kai Hwang. McGraw-Hill, 1993. ISBN 0-07-031622-8.

Concurrent Systems: An Integrated Approach to Operating Systems, Database, and Distributed Systems. Jean Bacon. Addison-Wesley, 1993. ISBN 0-201-41677-8.

Distributed Operating Systems. Andrew S. Tanenbaum. Prentice Hall, 1995. ISBN 0-13-219908-4.

Distributed Operating Systems: The Logical Design. A. Goscinski. Addison-Wesley, 1991. ISBN 0-201-41704-9.

Distributed Systems, Concepts, and Designs. G. Coulouris, et al. Addison-Wesley, 1994. ISBN 0-201-62433-8.

Operating System Concepts. 4th Ed., Abraham Silberschatz, Peter Galvin. Addison-Wesley, 1994. ISBN 0-201-50480-4.

POSIX

Information Technology-Portable Operating System Interface (POSIX): System Application Program Interface (API) (C Language). ANSI/IEEE Std. 1003.1. 1996 Edition. ISO/IEC 9945-1: 1996. IEEE Standards Office. ISBN 1-55937-573-6.

Programming with POSIX threads. David R. Butenhof. Addison Wesley Longman, Inc., 1997. ISBN 0-201-63392-2.

Programming

Advanced Programming in the UNIX Environment. Richard W. Stevens. Addison-Wesley, 1992. ISBN 0-201-56317-7.

Debugging with GDB: The GNU Source-Level Debugger for GDB version 4.18. Richard Stallman. Cygnus Support; Out of print—See http://www.redhat.com/support/manuals/gnupro99r1/ for the online version.

Open Source Development with CVS, Karl Franz Fogel. Coriolis Group, 1999. ISBN: 1-57610-490-7.

Porting UNIX Software: From Download to Debug. Greg Lehey. O'Reilly, 1995. ISBN 1-56592-126-7.

The Standard C Library. P.J. Plauger. Prentice Hall, 1992. ISBN 0-13-131509-9.

Websites and Online Resources

Apple's developer website (http://www.apple.com/developer) is a general repository for developer documentation. Additionally, the following sites provide more domain-specific information.

Apple's Public Source projects and Darwin OS

http://www.publicsource.apple.com

The Berkeley Software Distribution (BSD)

http://www.FreeBSD.org

http://www.NetBSD.org

http://www.OpenBSD.org

BSD Networking

http://www.kohala.com/start/

CVS (Concurrent Versions System)

http://www.publicsource.apple.com/tools/cvs/cederquist

Embedded C++

http://www.caravan.net/ec2plus

GDB, GNUPro Toolkit 99r1 Documentation

http://www.redhat.com/support/manuals/gnupro99r1/

The Internet Engineering Task Force (IETF)

http://www.ietf.org

jam

http://www.perforce.com/jam/jam.html

The PowerPC CPU

http://www.motorola.com/SPS/PowerPC/

The Single UNIX Specification Version 2

http://www.opengroup.org/onlinepubs/007908799

Stackable File Systems

http://www.isi.edu/~johnh/WORK/stacking_faq.htm

The USENIX Association; USENIX Proceedings

http://www.usenix.org

http://www.usenix.org/publications/library/

Index