Inside Mac OS X

Kernel Programming

November 2002

Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Figures and Tables

About This Document

The purpose of this document 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. In addition, it goes into detail about topics of interest to kernel programmers as a whole.

This is *not* a document on drivers. It covers device drivers at a high level only. It does, however, cover some areas of interest to driver writers, such as crossing the user-kernel boundary. If you are writing device drivers, you should primarily read the document *Inside Mac OS X: I/O Kit Fundamentals*, but you may still find this document helpful as background reading.

Who Should Read This Document

This document 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
- network-extension writers
- file-system writers
- developers of software that modifies file system data on-the-fly
- system programmers familiar with BSD, Linux, and similar operating systems
- developers who want to learn about kernel programming

If you fall into one of these categories, you may find this document helpful. It is important to stress the care needed when writing code that resides in the kernel, however, as noted in "Keep Out" (page 7).

Road Map

The goal of this document is to describe the various major components of Mac OS X at a conceptual level, then provide more detailed programming information for developers working in each major area. It is divided into several parts.

The first part is a kernel programming overview, which discusses programming guidelines that apply to all aspects of kernel programming. This includes issues such as security, SMP safety, style, performance, and the Mac OS X kernel architecture as a whole. This part contains the chapters "Keep Out" (page 7), "Kernel Architecture Overview" (page 10), "Security Considerations" (page 9), "Performance Considerations" (page 26), and "Kernel Programming Style" (page 33).

The next part describes Mach and the bootstrap task, including information about IPC, bootstrap contexts, ports and port rights, and so on. This includes the chapters "Mach Overview" (page 42), "Memory and Virtual Memory" (page 53), "Mach Scheduling and Thread Interfaces" (page 67), and "Bootstrap Contexts" (page 81).

The third part describes the I/O Kit and BSD. The I/O Kit is described at only a high level, since it is primarily of interest to driver developers. The BSD subsystem is covered in more detail, including descriptions of BSD networking and file systems. This includes the chapters "I/O Kit Overview" (page 85), "BSD Overview" (page 94), "File Systems Overview" (page 100), and "Network Architecture" (page 102).

The fourth part describes kernel services, including boundary crossings, synchronization, queues, clocks, timers, shutdown hooks, and boot option handling. This includes the chapters "Boundary Crossings" (page 107), "Synchronization Primitives" (page 129), and "Miscellaneous Kernel Services" (page 139).

The fifth part explains how to build and debug the kernel and kernel extensions. This includes the chapters "Kernel Extension Overview" (page 146) and "Building and Debugging Kernels" (page 153).

Each part begins with an overview chapter or chapters, followed by chapters that address particular areas of interest.

The document ends with a glossary of terms used throughout the preceding chapters as well as a bibliography which provides numerous pointers to other reference materials.

Glossary terms are highlighted in bold when first used. While most terms are defined when they first appear, the definitions are all in the glossary for convenience. 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 goal of this document is very broad, providing a firm grounding in the fundamentals of Mac OS X kernel programming for developers from many backgrounds. Due to the complex nature of kernel programming and limitations on the length of this document, however, it is not always possible to provide introductory material for developers who do not have at least some background in their area of interest. It is also not possible to cover every detail of certain parts of the kernel. If you run into problems, you should join the appropriate Darwin discussion list and ask questions. You can find the lists at http://www.lists.apple.com/.

For this reason, the bibliography contains high-level references that should help familiarize you with some of the basic concepts that you need to understand fully the material in this document.

This document is, to a degree, a reference document. The introductory sections should be easily read, and we recommend that you do so in order to gain a general understanding of each topic. Likewise, the first part of each chapter, and in many cases, of sections within chapters, will be tailored to providing a general understanding of individual topics. However, you should not plan to read this document cover to cover, but rather, take note of topics of interest so that you can refer back to them when the need arises.

Other Apple Publications

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

You can obtain other documents 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 document, go to the developer documentation website at http://developer.apple.com/techpubs and follow the links for hardcopy documents.

Mach API Reference

If you plan to do extensive work inside the Mac OS X kernel, you may find it convenient to have a complete Mach API reference, since this document only documents the most common and useful portions of the Mach API. In order to better understand certain interfaces, it may also be helpful to study the implementations that led up to those used in Mac OS X, particularly to fill in gaps in understanding of the fundamental principles of the implementation.

Mac OS X is based on the Mach 3.0 microkernel, designed by Carnegie Mellon University, and later adapted to the Power Macintosh by Apple and the Open Software Foundation Research Institute (now part of Silicomp). This was known as osfmk, and was part of MkLinux (http://www.mklinux.org). Later, this and code from OSF's commercial development efforts were incorporated into Darwin's kernel. Throughout this evolutionary process, the Mach APIs used in Mac OS X diverged in many ways from the original CMU Mach 3 APIs.

You may find older versions of the Mach source code interesting, both to satisfy historical curiosity and to avoid remaking mistakes made in earlier implementations. MkLinux maintains an active CVS repository with their recent versions of Mach kernel source code. Older versions can be obtained through

various Internet sites. You can also find CMU Mach white papers by searching for Mach on the CMU computer science department's website (http://www.cs.cmu.edu), along with various source code samples.

Up-to-date versions of the Mach 3 APIs that Mac OS X provides are described in the Mach API reference in the kernel sources. The kernel sources can be found in the xnu project on http://www.opensource.apple.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 (http://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 (http://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 (http://kbase.info.apple.com/). Contains technical articles, tutorials, FAQs, technical notes, and other information.

CHAPTER 2

Keep Out

This document assumes a broad general understanding of kernel programming concepts. There are many good introductory operating systems texts. This is not one of them. For more information on basic operating systems programming, you should consider the texts mentioned in the bibliography at the end of this document.

Many developers are justifiably cautious about programming in the kernel. A decision to program in the kernel is not to be taken lightly. Kernel programmers have a responsibility to users that greatly surpasses that of programmers who write user programs.

Why You Should Avoid Programming in the Kernel

Kernel code must be nearly perfect. A bug in the kernel could cause random crashes, data corruption, or even render the operating system inoperable. It is even possible for certain errant operations to cause permanent and irreparable damage to hardware, for example, by disabling the cooling fan and running the CPU full tilt.

Kernel programming is a black art that should be avoided if at all possible. Fortunately, kernel programming is usually unnecessary. You can write most software entirely in user space. Even most device drivers (FireWire and USB, for example) can be written as applications, rather than as kernel code. A few low-level drivers must be resident in the kernel's address space, however, and this document might be marginally useful if you are writing drivers that fall into this category.

CHAPTER 2

Keep Out

Despite parts of this document being useful in driver writing, this is *not* a document about writing drivers. In Mac OS X, you write device drivers using the I/O Kit. While this document covers the I/O Kit at a conceptual level, the details of I/O Kit programming are beyond the scope of this document. Driver writers are encouraged to read *Inside Mac OS X: I/O Kit Fundamentals* for detailed information about the I/O Kit.

This document covers most aspects of kernel programming *with the exception* of device drivers. Covered topics include scheduling, virtual memory pagers and policies, Mach IPC, file systems, networking protocol stacks, process and thread management, kernel security, synchronization, and a number of more esoteric topics.

To summarize, kernel programming is an immense responsibility. You must be exceptionally careful to ensure that your code does not cause the system to crash, does not provide any unauthorized user access to someone else's files or memory, does not introduce remote or local root exploits, and does not cause inadvertent data loss or corruption.

CHAPTER 3

Security Considerations

Kernel-level security can mean many things, depending on what kind of kernel code you are writing. This chapter points out some common security issues at the kernel or near-kernel level and where applicable, describes ways to avoid them. These issues are covered in the following sections:

Many of these issues are also relevant for application programming, but are crucial for programmers working in the kernel. Others are special considerations that application programers might not expect or anticipate.

Note: The terms cleartext and plaintext both refer to unencrypted text. These terms can generally be used interchangeably, although in some circles, the term cleartext is restricted to unencrypted transmission across a network. However, in other circles, the term plaintext (or sometimes plain text) refers to plain ASCII text (as opposed to HTML or rich text. To avoid any potential confusion, this chapter will use the term cleartext to refer to unencrypted text.

In order to understand security in Mac OS X, it is important to understand that there are two security models at work. One of these is the kernel security model, which is based on users, groups, and very basic per-user and per-group rights. The other is a user-level security model, which is based on keys, keychains, groups, users, password-based authentication, and a host of other details that are beyond the scope of this document.

The user level of security contains two basic features that you should be aware of as a kernel programmer: Security Server and Keychain Manager.

The Security Server consists of a daemon and various access libraries for caching permission to do certain tasks, based upon various means of authentication, including passwords and group membership. When a program requests permission to do something, the Security Server basically says "yes" or "no," and caches that decision so that further requests from that user (for similar actions within a single context) do not require reauthentication for a period of time.

The Keychain Manager is a daemon that provides services related to the keychain, a central repository for a user's encryption / authentication keys. For more high level information on keys, see "Key-based Authentication and Encryption" (page 18).

The details of the user-level security model use are far beyond the scope of this document. However, if you are writing an application that requires services of this nature, you should consider taking advantage of the Security Server and Keychain Manager from the user-space portion of your application, rather than attempting equivalent services in the kernel. More information about these services can be found in the Technical Publications section of Apple's website (http://developer.apple.com/techpubs).

Security Implications of Paging

Paging has long been a major problem for security-conscious programmers. If you are writing a program that does encryption, the existence of even a small portion of the cleartext of a document in a backing store could be enough to reduce the complexity of breaking that encryption by orders of magnitude.

Indeed, many types of data, such as hashes, unencrypted versions of sensitive data, and authentication tokens, should generally not be written to disk due to the potential for abuse. This raises an interesting problem. There is no good way to deal with this in user space (unless a program is running as `root`). However, for kernel code, it is possible to prevent pages from being written out to a backing store. This process is referred to as "wiring down" memory, and is described further in "Memory Mapping and Block Copying" (page 126).

The primary purpose of wired memory is to allow DMA-based I/O. Since hardware DMA controllers generally do not understand virtual addressing, information used in I/O must be physically in memory at a particular location and must not move until the I/O operation is complete. This mechanism can also be used to prevent sensitive data from being written to a backing store.

Because wired memory can never be paged out (until it is unwired), wiring large amounts of memory has drastic performance repercussions, particularly on systems with small amounts of memory. For this reason, you should take care not to wire down memory indiscriminately and only wire down memory if you have a very good reason to do so.

In Mac OS X, memory may be wired down at the time of allocation. In I/O Kit, you specify `IOMalloc` and `IOFree` to allocate wired memory. In Mach, `kmem_alloc_wired` (and `kmem_free`) can be used. It may also be wired down after allocation. For more information on wired memory, see "Memory Mapping and Block Copying" (page 126).

Buffer Overflows and Invalid Input

Buffer overflows are one of the more common bugs in both application and kernel programming. The most common cause is failing to allocate space for the NULL character that terminates a string in C or C++. However, user input can also cause buffer overflows if fixed-size input buffers are used and appropriate care is not taken to prevent overflowing these buffers.

The most obvious protection, in this case, is the best one. Either don't use fixed-length buffers or add code to reject input that overflows the buffer. The implementation details in either case depend on the type of code you are writing.

Other types of invalid input can be somewhat harder to handle, however. As a general rule, you should be certain that switch statements have a default case unless you have listed every legal value for the width of the type.

A common mistake is assuming that listing every possible value of an `enum` type provides protection. An enum is generally implemented as either a `char` or an `int` internally. A careless or malicious programmer could easily pass any value to a kernel function, including those not explicitly listed in the type, simply by using a different prototype that defines the parameter as, for example, an `int`.

Another common mistake is to assume that you can dereference a pointer passed to your function by another function. You should always check for null pointers before dereferencing them. Starting a function with

```
int do_something(bufptr *bp, int flags) {
    char *token = bp->b_data;
```

is the surest way to guarantee that someone else will pass in a null buffer pointer, either maliciously or because of programmer error. In a user program, this is annoying. In a file system, it is devastating.

Security is particularly important for kernel code that draws input from a network. Assumptions about packet size are frequently the cause of security problems. Always watch for packets that are too big and handle them in a reasonable way. Likewise, always verify checksums on packets. This can help you determine if a packet was modified, damaged, or truncated in transit, though it is far from foolproof. If the validity of data from a network is of vital importance, you should use remote authentication, signing, and encryption mechanisms such as those described in "Remote Authentication" (page 14) and "Key-based Authentication and Encryption" (page 18).

User Credentials

As described in the introduction to this chapter, Mac OS X has two different means of authenticating users. The user-level security model (including the Keychain Manager and the Security Server) is beyond the scope of this document. The kernel security model, however, is of greater interest to kernel developers, and is much more straightforward than the user-level model.

The kernel security model is based on relatively simple credentials that are passed around within the kernel to identify the user and group of the calling process's owner.

These user credentials used in the kernel are stored in a variable of type `struct ucred`.

This structure has four fields:

- `cr_ref`—reference count (used internally)
- `cr_uid`—user ID
- `cr_ngroups`—number of groups in `cr_groups`
- `cr_groups[NGROUPS]`—list of groups to which the user belongs

This structure has an internal reference counter to prevent unintentionally freeing the memory associated with it while it is still in use. For this reason, you should not indiscriminately copy this object but should instead either use `crdup` to duplicate it or use `crcopy` to duplicate it and (potentially) free the original. You should be sure to `crfree` any copies you might make. You can also create a new, empty `ucred` structure with `crget`.

The prototypes for these functions follow:

- `struct ucred *crdup(struct ucred *cr)`
- `struct ucred *crcopy(struct ucred *cr)`
- `struct ucred *crget(void)`
- `void crfree(struct ucred *cr)`

One of the most important things to remember when working with credentials is that they are per process, not per context. This is important because a process may not be running as the console user. Two examples of this are processes started from an `ssh` session (since ssh runs in the startup context) and `setuid` programs (which run as a different user in the *same* login context).

It is crucial to be aware of these issues. If you are communicating with a `setuid root` GUI application in a user's login context, and if you are executing another application or are reading sensitive data, you probably want to treat it as if it had the same authority as the console user, not the authority of the effective user ID caused by running `setuid`. This is particularly problematic when dealing with

programs that run as `setuid root` if the console user is not in the admin group. Failure to perform reasonable checks can lead to major security holes down the road.

However, this is not a hard and fast rule. Sometimes it is not obvious whether to use the credentials of the running process or those of the console user. In such cases, it is often reasonable to have a helper application show a dialog box on the console to require interaction from the console user. If this is not possible, a good rule of thumb is to assume the lesser of the privileges of the current and console users, as it is almost always better to have kernel code occasionally fail to provide a needed service than to provide that service unintentionally to an unauthorized user or process.

It is generally easier to determine the console user from a user space application than from kernel space code. Thus, you should generally do such checks from user space. If that is not possible, however, the variable `console_user` (maintained by the VFS subsystem) will give you the uid of the last owner of `/dev/console` (maintained by a bit of code in the `chown` system call). This is certainly not an ideal solution, but it does provide the most likely identity of the console user. Since this is only a "best guess," however, you should use this only if you cannot do appropriate checking in user space.

Remote Authentication

This is one of the more difficult problems in computer security: the ability to identify someone connecting to a computer remotely. One of the most secure methods is the use of public key cryptography, which is described in more detail in "Key-based Authentication and Encryption" (page 18). However, many other means of authentication are possible, with varying degrees of security.

Some other authentication schemes include:

- blind trust
- IP-only authentication
- password (shared secret) authentication
- combination of IP and password authentication

- one-time pads (challenge-response)
- time-based authentication

Most of these are obvious, and require no further explanation. However, one-time pads and time-based authentication may be unfamiliar to many people outside security circles, and are thus worth mentioning in more detail.

One-Time Pads

Based on the concept of “challenge-response” pairs, one-time pad (OTP) authentication requires that both parties have an identical list of pairs of numbers, words, symbols, or whatever, sorted by the first item. When trying to access a remote system, the remote system prompts the user with a challenge. The user finds the challenge in the first column, then sends back the matching response. Alternatively, this could be an automated exchange between two pieces of software.

For maximum security, no challenge should ever be issued twice. For this reason, and because these systems were initially implemented with a paper pad containing challenge-response, or CR pairs, such systems are often called one-time pads.

The one-time nature of OTP authentication makes it impossible for someone to guess the appropriate response to any one particular challenge by a brute force attack (by responding to that challenge repeatedly with different answers). Basically, the only way to break such a system, short of a lucky guess, is to actually know some portion of the contents of the list of pairs.

For this reason, one-time pads can be used over insecure communication channels. If someone snoops the communication, they can obtain that challenge-response pair. However, that information is of no use to them, since that particular challenge will never be issued again. (It does not even reduce the potential sample space for responses, since only the challenges must be unique.)

Time-based authentication

This is probably the least understood means of authentication, though it is commonly used by such technologies as SecurID. The concept is relatively straightforward. You begin with a mathematical function that takes a small number

of parameters (two, for example) and returns a new parameter. A good example of such a function is the function that generates the set of Fibonacci numbers (possibly truncated after a certain number of bits, with arbitrary initial seed values).

Take this function, and add a third parameter, `t`, representing time in units of `k` seconds. Make the function be a generating function on `t`, with two seed values, a and b, where

$$f(x,y) = (x + y) \text{ MOD } (2^{32})$$
$$g(t) = a,\ 0 \le t < k$$
$$g(t) = b,\ k \le t < 2k$$
$$g(t) = f(g(\lfloor \log_k t \rfloor - 2), g(\lfloor \log_k t \rfloor - 1))$$

In other words, every `k` seconds, you calculate a new value based on the previous two and some equation. Then discard the oldest value, replacing it with the second oldest value, and replace the second oldest value with the value that you just generated.

As long as both ends have the same notion of the current time and the original two numbers, they can then calculate the most recently generated number and use this as a shared secret. Of course, if you are writing code that does this, you should use a closed form of this equation, since calculating Fibonacci numbers recursively without additional storage grows at `O(2^(t/k))`, which is not practical when `t` is measured in years and `k` is a small constant measured in seconds.

The security of such a scheme depends on various properties of the generator function, and the details of such a function are beyond the scope of this document. For more information, you should obtain an introductory text on cryptography,. such as Bruce Schneier's *Applied Cryptography*.

Temporary Files

Temporary files are a major source of security headaches. If a program does not set permissions correctly and in the right order, this can provide a means for an attacker to arbitrarily modify or read these files. The security impact of such modifications depends on the contents of the files.

Temporary files are of much less concern to kernel programmers, since most kernel code does not use temporary files. Indeed, kernel code should generally not use files at all. However, many people programming in the kernel are doing so to facilitate the use of applications that may use temporary files. As such, this issue is worth noting.

The most common problem with temporary files is that it is often possible for a malicious third party to delete the temporary file and substitute a different one with relaxed permissions in its place. Depending on the contents of the file, this could range from being a minor inconvenience to being a relatively large security hole, particularly if the file contains a shell script that is about to be executed with the permissions of the program's user.

/dev/mem and /dev/kmem

One particularly painful surprise to people doing security programming in most UNIX or UNIX-like environments is the existence of `/dev/mem` and `/dev/kmem`. These device files allow the `root` user to arbitrarily access the contents of physical memory and kernel memory, respectively. There is absolutely nothing you can do to prevent this. From a kernel perspective, root is omnipresent and omniscient. If this is a security concern for your program, then you should consider whether your program should be used on a system controlled by someone else and take the necessary precautions.

Key-based Authentication and Encryption

Key-based authentication and encryption are ostensibly some of the more secure means of authentication and encryption, and can exist in many forms. The most common forms are based upon a shared secret. The DES, 3DES (triple-DES), IDEA, twofish, and blowfish ciphers are examples of encryption schemes based on a shared secret. Passwords are an example of an authentication scheme based on a shared secret.

The idea behind most key-based encryption is that you have an encryption key of some arbitrary length that is used to encode the data, and that same key is used in the opposite manner (or in some cases, in the same manner) to decode the data.

The problem with shared secret security is that the initial key exchange must occur in a secure fashion. If the integrity of the key is compromised during transmission, the data integrity is lost. This is not a concern if the key can be generated ahead of time and placed at both transport endpoints in a secure fashion. However, in many cases, this is not possible or practical because the two endpoints (be they physical devices or system tasks) are controlled by different people or entities. Fortunately, an alternative exists, known as zero-knowledge proofs.

The concept of a zero-knowledge proof is that two seemingly arbitrary key values, x and y, are created, and that these values are related by some mathematical function *f* in such a way that

```
f(f(a,k1),k2) = a
```

That is, applying a well-known function to the original cleartext using the first key results in ciphertext which, when that same function is applied to the ciphertext using the second key returns the original data. This is also reversible, meaning that

```
f(f(a,k2),k1) = a
```

If the function `f` is chosen correctly, it is extremely difficult to derive x from y and vice-versa, which would mean that there is no function that can easily transform the ciphertext back into the cleartext based upon the key used to encode it.

An example of this is to choose the mathematical function to be

```
f(a,k)=((a*k) MOD 256) + ((a*k)/256)
```

where a is a byte of cleartext, and k is some key 8 bits in length. This is an extraordinarily weak cipher, since the function f allows you to easily determine one key from the other, but it is illustrative of the basic concept.

Pick k1 to be 8 and k2 to be 32. So for a=73, (a * 8)=584. This takes two bytes, so add the bits in the high byte to the bits of the low byte, and you get 74. Repeat this process with 32. This gives you 2368. Again, add the bits from the high byte to the bits of the low byte, and you have 73 again.

This mathematical concept (with very different functions), when put to practical use, is known as public key (PK) cryptography, and forms the basis for RSA and DSA encryption.

Public Key Weaknesses

Public key encryption can be very powerful when used properly. However, it has a number of inherent weaknesses. A complete explanation of these weaknesses is beyond the scope of this document. However, it is important that you understand these weaknesses at a high level to avoid falling into some common traps. Some commonly mentioned weakness of public key cryptography include:

- Trust model for key exchange
- Pattern sensitivity
- Short data weakness

Trust Models

The most commonly discussed weakness of public key cryptography is the initial key exchange process itself. If someone manages to intercept a key during the initial exchange, he or she could instead give you his or her own public key and intercept messages going to the intended party. This is known as a man-in-the-middle attack.

For such services as ssh, most people either manually copy the keys from one server to another or simply assume that the initial key exchange was successful. For most purposes, this is sufficient.

In particularly sensitive situations, however, this is not good enough. For this reason, there is a procedure known as **key signing**. There are two basic models for key signing: the central authority model and the web of trust model.

The central authority model is straightforward. A central certifying agency signs a given key, and says that they believe the owner of the key is who he or she claims to be. If you trust that authority, then by association, you trust keys that the authority claims are valid.

The web of trust model is somewhat different. Instead of a central authority, individuals sign keys belonging to other individuals. By signing someone's key, you are saying that you trust that the person is really who he or she claims to be and that you believe that the key really belongs to him or her. The methods you use for determining that trust will ultimately impact whether others trust your signatures to be valid.

There are many different ways of determining trust, and thus many groups have their own rules for who should and should not sign someone else's key. Those rules are intended to make the trust level of a key depend on the trust level of the keys that have signed it.

The line between central authorities and web of trust models is not quite as clear-cut as you might think, however. Many central authorities are hierarchies of authorities, and in some cases, they are actually webs of trust among multiple authorities. Likewise, many webs of trust may include centralized repositories for keys. While those repositories don't provide any certification of the keys, they do provide centralized access. Finally, centralized authorities can easily sign keys as part of a web of trust.

There are many websites that describe webs of trust and centralized certification schemes. A good general description of several such models can be found at http://world.std.com/~cme/html/web.html.

Sensitivity to Patterns and Short Messages

Existing public key encryption algorithms do a good job at encrypting semi-random data. They fall short when encrypting data with certain patterns, as these patterns can inadvertently reveal information about the keys. The particular patterns depend on the encryption scheme. Inadvertently hitting such a pattern does not allow you to determine the private key. However, they can reduce the search space needed to decode a given message.

Short data weakness is closely related to pattern sensitivity. If the information you are encrypting consists of a single number, for example the number 1, you basically get a value that is closely related mathematically to the public key. If the intent is to make sure that only someone with the private key can get the original value, you have a problem.

In other words, public key encryption schemes generally do not encrypt all patterns equally well. For this reason (and because public key cryptography tends to be slower than single key cryptography), public keys are almost never used to encrypt end-user data. Instead, they are used to encrypt a **session key**. This session key is then used to encrypt the actual data using a shared secret mechanism such as 3DES, AES, blowfish, and so on.

Using Public Keys for Message Exchange

Public key cryptography can be used in many ways. When both keys are private, it can be used to send data back and forth. However this use is no more useful than a shared secret mechanism. In fact, it is frequently weaker, for the reasons mentioned earlier in the chapter. Public key cryptography becomes powerful when one key is made public.

Assume that Ernie and Bert want to send coded messages. Ernie gives Bert his public key. Assuming that the key was not intercepted and replaced with someone else's key, Bert can now send data to Ernie securely, because data encrypted with the public key can only be decrypted with the private key (which only Ernie has).

Bert uses this mechanism to send a shared secret. Bert and Ernie can now communicate with each other using a shared secret mechanism, confident in the knowledge that no third party has intercepted that secret. Alternately, Bert could give Ernie his public key, and they could both encrypt data using each other's public keys, or more commonly by using those public keys to encrypt a session key and encrypting the data with that session key.

Using Public Keys for Identity Verification

Public key cryptography can also be used for verification of identity. Kyle wants to know if someone on the Internet who claims to be Stan is really Stan. A few months earlier, Stan handed Kyle his public key on a floppy disk. Thus, since Kyle already has Stan's public key (and trusts the source of that key), he can now easily verify Stan's identity.

To achieve this, Kyle sends a cleartext message and asks Stan to encrypt it. Stan encrypts it with his private key. Kyle then uses Stan's public key to decode the ciphertext. If the resulting cleartext matches, then the person on the other end must be Stan (unless someone else has Stan's private key).

Using Public Keys for Data Integrity Checking

Finally, public key cryptography can be used for signing. Ahmed is in charge of meetings of a secret society called the Stupid Acronym Preventionists club. Abraham is a member of the club and gets a TIFF file containing a notice of their next meeting, passed on by way of a fellow member of the science club, Albert. Abraham is concerned, however, that the notice might have come from Bubba, who is trying to infiltrate the SAPs.

Ahmed, however, was one step ahead, and took a checksum of the original message and encrypted the checksum with his private key, and sent the encrypted checksum as an attachment. Abraham used Ahmed's public key to decrypt the checksum, and found that the checksum did not match that of the actual document. He wisely avoided the meeting. Isaac, however, was tricked into revealing himself as a SAP because he didn't remember to check the signature on the message.

The moral of this story? One should always beware of geeks sharing TIFFs—that is, if the security of some piece of data is important and if you do not have a direct, secure means of communication between two applications, computers, people, and so on, you must verify the authenticity of any communication using signatures, keys, or some other similar method. This may save your data and also save face.

Encryption Summary

Encryption is a powerful technique for keeping data secure if the initial key exchange occurs in a secure fashion. One means for this is to have a public key, stored in a well-known (and trusted) location. This allows for one-way encrypted communication through which a shared secret can be transferred for later two-way encrypted communication.

You can use encryption not only for protecting data, but also for verifying the authenticity of data by encrypting a checksum. You can also use it to verify the identity of a client by requiring that the client encrypt some random piece of data as proof that the client holds the appropriate encryption key.

Encryption, however, is not the final word in computer security. Because it depends on having some form of trusted key exchange, additional infrastructure is needed in order to achieve total security in environments where communication can be intercepted and modified.

Console Debugging

WARNING
Failure to follow this advice can unintentionally expose security-critical information.

In traditional UNIX and UNIX-like systems, the console is owned by root. Only root sees console messages. For this reason, print statements in the kernel are relatively secure.

In Mac OS X, any user can run the Console application. This represents a major departure from other UNIX-like systems. While it is never a good idea to include sensitive information in kernel debugging statements, it is particularly important not to do so in Mac OS X. You must assume that any information displayed to the console could potentially be read by any user on the system (since the console is virtualized in the form of a user-viewable window).

Printing any information involving sensitive data, including its location on disk or in memory, represents a security hole, however slight, and you should write your code accordingly. Obviously this is of less concern if that information is only printed when the user sets a debugging flag somewhere, but for normal use, printing potentially private information to the console is strongly discouraged.

You must also be careful not to inadvertently print information that you use for generating password hashes or encryption keys, such as seed values passed to a random number generator.

This is, by necessity, not a complete list of information to avoid printing to the console. You must use your own judgement when deciding whether a piece of information could be valuable if seen by a third party, and then decide if it is appropriate to print it to the console.

Code Passing

There are many ways of passing executable code into the kernel from user space. For the purposes of this section, executable code is not limited to compiled object code. It includes any instructions passed into the kernel that significantly affect control flow. Examples of passed-in executable code range from simple rules such as the filtering code uploaded in many firewall designs to bytecode uploads for a SCSI card.

If it is possible to execute your code in user space, you should not even contemplate pushing code into the kernel. For the rare occasion where no other reasonable solution exists, however, you may need to pass some form of executable code into the kernel. This section explains some of the security ramifications of pushing code into the kernel and the level of verification needed to ensure consistent operation.

Here are some guidelines to minimize the potential for security holes:

1. No raw object code.

   Direct execution of code passed in from user space is *very* dangerous. Interpreted languages are the only reasonable solution for this sort of problem, and even this is fraught with difficulty. Traditional machine code can't be checked sufficiently to ensure security compliance.

2. Bounds checking.

   Since you are in the kernel, you are responsible for making sure that any uploaded code does not randomly access memory and does not attempt to do direct hardware access. You would normally make this a feature of the language itself, restricting access to the data element on which the bytecode is operating.

3. Termination checking.

   With very, very few exceptions, the language chosen should be limited to code that can be verified to terminate, and you should verify accordingly. If your driver is stuck in a tightly rolled loop, it is probably unable to do its job, and may impact overall system performance in the process. A language that does not allow (unbounded) loops (for example, allowing `for` but not `while` or `goto` could be one way to ensure termination.

4. Validity checking.

   Your bytecode interpreter would be responsible for checking ahead for any potentially invalid operations and taking appropriate punitive actions against the uploaded code. For example, if uploaded code is allowed to do math, then proper protection must be in place to handle divide by zero errors.

5. Sanity checking.

   You should verify that the output is something remotely reasonable, if possible. It is not always possible to verify that the output is correct, but it is generally possible to create rules that prevent egregiously invalid output.

   For example, a network filter rule should output something resembling packets. If the checksums are bad, or if other information is missing or corrupt, clearly the uploaded code is faulty, and appropriate actions should be taken. It would be highly inappropriate for Mac OS X to send out bad network traffic.

In general, the more restrictive the language set, the lower the security risk. For example, interpreting simple network routing policies is less likely to be a security problem than interpreting packet rewriting rules, which is less likely to be an issue than running Java bytecode in the kernel. As with anything else, you must carefully weigh the potential benefits against the potential drawbacks and make the best decision given the information available.

Performance Considerations

Performance is a key aspect of any software system. Nowhere is this more true than in the kernel, where small performance problems tend to be magnified by repeated execution. For this reason, it is extremely important that your code be as efficient as possible.

This chapter discusses the importance of low interrupt latency and fine-grained locking and tells you how to determine what portions of your code would benefit most from more efficient design.

Interrupt Latency

In Mac OS X, you will probably never need to write code that runs in an interrupt context. In general, only motherboard hardware requires this. However, in the unlikely event that you do need to write code in an interrupt context, interrupt latency should be a primary concern.

Interrupt latency refers to the delay between an interrupt being generated and an interrupt handler actually beginning to service that interrupt. In practice, the worst case interrupt latency is closely tied to the amount of time spent in **supervisor mode** (also called kernel mode) with interrupts off while handling some other interrupt. Low interrupt latency is necessary for reasonable overall performance, particularly when working with audio and video. In order to have reasonable soft real-time performance (for example, performance of multimedia applications), the interrupt latency caused by every device driver must be both small and bounded.

Mac OS X takes great care to bound and minimize interrupt latency for built-in drivers. It does this primarily through the use of **interrupt service threads** (also known as I/O service threads).

When Mac OS X takes an interrupt, the low-level trap handlers call up to a generic interrupt handling routine that clears the pending interrupt bit in the interrupt controller and calls a device-specific interrupt handler. That device-specific handler, in turn, sends a message to an interrupt service thread to notify it that an interrupt has occurred, and then the handler returns. When no further interrupts are pending, control returns to the currently executing thread.

The next time the interrupt service thread is scheduled, it checks to see if an interrupt has occurred, then services the interrupt. As the name suggests, this actually is happening in a thread context, not an interrupt context. This design causes two major differences from traditional operating system design:

- Interrupt latency is near zero, since the code executing in an interrupt context is very small.
- It is possible for an interrupt to occur while a device driver is executing. This means that traditional (threaded) device drivers can be preempted and must use locking or other similar methods to protect any shared data (although they need to do so anyway to work on computers with multiple processors).

This model is crucial to the performance of Mac OS X. You should not attempt to circumvent this design by doing large amounts of work in an interrupt context. Doing so will be detrimental to the overall performance of the system.

Locking Bottlenecks

It is difficult to communicate data between multiple threads or between thread and interrupt contexts without using locking or other synchronization. This locking protects your data from getting clobbered by another thread. However, it also has the unfortunate side effect of being a potential bottleneck.

In some types of communication (particularly n-way), locking can dramatically hinder performance by allowing only one thing to happen at a time. Read-write locks, discussed in "Synchronization Primitives" (page 129), can help alleviate this problem in the most common situation where multiple clients need to be able to read information but only rarely need to modify that data.

However, there are many cases where read-write locks are not helpful. This section discusses some possible problems and ways of improving performance within those constraints.

Working With Highly Contended Locks

When many threads need to obtain a lock (or a small number of threads need to obtain a lock frequently), this lock is considered highly contended. Highly contended locks frequently represent faulty code design, but they are sometimes unavoidable. In those cases, the lock tends to become a major performance bottleneck.

Take, for example, the issue of many-to-many communication that must be synchronized through a common buffer. While some improvement can be gained by using read-write locks instead of an ordinary mutex, the issue of multiple writers means that read-write locks still perform badly.

One possible solution for this many-to-many communication problem is to break the lock up into multiple locks. Instead of sharing a single buffer for the communication itself, make a shared buffer that contains accounting information for the communication (for example, a list of buffers available for reading). Then assign each individual buffer its own lock. The readers might then need to check several locations to find the right data, but this still frequently yields better performance, since writers must only contend for a write lock while modifying the accounting information.

Another solution for many-to-many communications is to eliminate the buffer entirely and communicate using message passing, sockets, IPC, RPC, or other methods.

Yet another solution is to restructure your code in a way that the locking is unnecessary. This is often much more difficult. One method that is often helpful is to take advantage of flags and atomic increments, as outlined in the next paragraph. For simplicity, a single-writer, single-reader example is presented, but it is possible to extend this idea to more complicated designs.

Take a buffer with some number of slots. Keep a read index and a write index into that buffer. When the write index and read index are the same, there is no data in the buffer. When writing, clear the next location. Then do an atomic increment on the pointer. Write the data. End by setting a flag at that new location that says that the data is valid.

Note that this solution becomes much more difficult when dealing with multiple readers and multiple writers, and as such, is beyond the scope of this section.

Reducing Contention by Decreasing Granularity

One of the fundamental properties of locks is granularity. The granularity of a lock refers to the amount of code or data that it protects. A lock that protects a large block of code or a large amount of data is referred to as a coarse-grained lock, while a lock that protects only a small amount of code or data is referred to as a fine-grained lock. A coarse-grained lock is much more likely to be contended (needed by one thread while being held by another) than a more finely grained lock.

There are two basic ways of decreasing granularity. The first is to minimize the amount of code executed while a lock is held. For example, if you have code that calculates a value and stores it into a table, don't take the lock before calling the function and release it after the function returns. Instead, take the lock in that piece of code right before you write the data, and release it as soon as you no longer need it.

Of course, reducing the amount of protected code is not always possible or practical if the code needs to guarantee consistency where the value it is writing depends on other values in the table, since those values could change before you obtain the lock, requiring you to go back and redo the work.

It is also possible to reduce granularity by locking the data in smaller units. In the above example, you could have a lock on each cell of the table. When updating cells in the table, you would start by determining the cells on which the destination cell depends, then lock those cells and the destination cell in some fixed order. (To avoid deadlock, you must always either lock cells in the same order or use an appropriate `try` function and release all locks on failure.)

Once you have locked all the cells involved, you can then perform your calculation and release the locks, confident that no other thread has corrupted your calculations. However, by locking on a smaller unit of data, you have also reduced the likelihood of two threads needing to access the same cell.

A slightly more radical version of this is to use read-write locks on a per-cell basis and always upgrade in a particular order. This is, however, rather extreme, and difficult to do correctly.

Code Profiling

Code profiling means determining how often certain pieces of code are executed. By knowing how frequently a piece of code is used, you can more accurately gauge the importance of optimizing that piece of code. There are a number of good tools for profiling user space applications. However, code profiling in the kernel is a very different beast, since it isn't reasonable to attach to it like you would a running process. (It is possible by using a second computer, but even then, it is not a trivial task.)

This section describes two useful ways of profiling your kernel code: counters and lock profiling. Any changes you make to allow code profiling should be done only during development. These are not the sort of changes that you want to release to end users.

Using Counters for Code Profiling

The first method of code profiling is with counters. To profile a section of code with a counter, you must first create a global variable whose name describes that piece of code and initialize it to zero. You then add something like

```
#ifdef PROFILING
            foo_counter++;
#endif
```

in the appropriate piece of code. If you then define `PROFILING`, that counter is created and initialized to zero, then incremented each time the code in question is executed.

One small snag with this sort of profiling is the problem of obtaining the data. This can be done in several ways. The simplest is probably to install a `sysctl`, using the address of `foo_counter` as an argument. Then, you could simply issue the `sysctl` command from the command line and read or clear the variable. Adding a `sysctl` is described in more detail in "BSD sysctl API" (page 118).

In addition to using `sysctl`, you could also obtain the data by printing its value when unloading the module (in the case of a KEXT) or by using a remote debugger to attach to the kernel and directly inspecting the variable. However, a `sysctl` provides the most flexibility. With a `sysctl`, you can sample the value at any time, not just when the module is unloaded. The ability to arbitrarily sample the value makes it easier to determine the importance of a piece of code to one particular action.

If you are developing code for use in the I/O Kit, you should probably use your driver's `setProperties` call instead of a `sysctl`.

Lock Profiling

Lock profiling is another useful way to find the cause of code inefficiency. Lock profiling can give you the following information:

- how many times a lock was taken
- how long the lock was held on average
- how often the lock was unavailable

Put another way, this allows you to determine the contention of a lock, and in so doing, can help you to minimize contention by code restructuring.

There are many different ways to do lock profiling. The most common way is to create your own lock calls that increment a counter and then call the real locking functions. When you move from debugging into a testing cycle before release, you can then replace the functions with defines to cause the actual functions to be called directly. For example, you might write something like this:

```
extern struct timeval time;

boolean_t    mymutex_try(mymutex_t *lock) {
    int ret;
    ret=mutex_try(lock->mutex);
    if (ret) {
        lock->tryfailcount++;
    }
    return ret;
}

void    mymutex_lock(mymutex_t *lock) {
```

```
    if (!(mymutex_try(lock))) {
        mutex_lock(lock->mutex);
    }
    lock->starttime = time.tv_sec;
}
void    mymutex_unlock(mymutex_t *lock) {
    lock->lockheldtime += (time.tv_sec - lock->starttime);
    lock->heldcount++;
    mutex_unlock(lock->mutex);
}
```

This routine has accuracy only to the nearest second, which is not particularly accurate. Ideally, you want to keep track of both `time.tv_sec` and `time.tv_usec` and roll the microseconds into seconds as the number gets large.

From this information, you can obtain the average time the lock was held by dividing the total time held by the number of times it was held. It also tells you the number of times a lock was taken immediately instead of waiting, which is a valuable piece of data when analyzing contention.

As with counter-based profiling, after you have written code to record lock use and contention, you must find a way to obtain that information. A `sysctl` is a good way of doing this, since it is relatively easy to implement and can provide a "snapshot" view of the data structure at any point in time. For more information on adding a `sysctl`, see "BSD sysctl API" (page 118).

Another way to do lock profiling is to use the built-in ETAP (Event Trace Analysis Package). This package consists of additional code designed for lock profiling. However, since this requires a kernel recompile, it is generally not recommended.

CHAPTER 5

Kernel Programming Style

As described in "Keep Out" (page 7), programming in the kernel is fraught with hazards that can cause instability, crashes, or security holes. In addition to these issues, programming in the kernel has the potential for compatibility problems. If you program only to the interfaces discussed in this document or other Apple documents, you will avoid the majority of these.

However, even limiting yourself to documented interfaces does not protect you from a handful of pitfalls. The biggest potential problem that you face is namespace collision, which occurs when your function, variable, or class name is the same as someone else's. Since this makes one kernel extension or the other fail to load correctly (in a non-deterministic fashion), Apple has established function naming conventions for C and C++ code within the kernel. These are described in "Standard C Naming Conventions" (page 35) and "C++ Naming Conventions" (page 33), respectively.

In addition to compatibility problems, kernel extensions that misbehave can also dramatically decrease the system's overall performance or cause crashes. Some of these issues are described in "Performance and Stability Tips" (page 38). For more thorough coverage of performance and stability, you should also read the chapters "Security Considerations" (page 9) and "Performance Considerations" (page 26).

C++ Naming Conventions

Basic I/O Kit C++ naming conventions are defined in the document *Inside Mac OS X: Writing I/O Kit Drivers*. This section refines those conventions in ways that should make them more useful to you as a programmer.

Basic Conventions

The primary conventions are as follows:

- Use the Java-style reverse DNS naming convention, substituting underscores for periods. For example, `com_apple_foo`.
- Avoid the following reserved prefixes:
  - OS
  - os
  - IO
  - io
  - Apple
  - apple
  - AAPL
  - aapl

This ensures that you will not collide with classes created by other companies or with future classes added to the operating system by Apple. It does not protect you from other projects created within your company, however, and for this reason, some additional guidelines are suggested.

Additional Guidelines

These additional guidelines are intended to minimize the chance of accidentally breaking your own software and to improve readability of code by developers. This is particularly of importance for open source projects.

- Begin each function name within a class with the name of the class. For example, if the class is `com_apple_iokit_pickle`, and the function would be `eat`, you should name the function `pickle_eat`. This makes it easier to see class associations, particularly when called from other files.
- Name classes based on project names. For example, if you are working on project Schlassen, and one of its classes was `pickle`, you would name the class `com_apple_iokit_schlassen_pickle`.

- Name classes hierarchically if your organization is large. For example, if Apple's marketing department were working on the Schlassen project, then they might name the class `com_apple_iokit_marketing_schlassen_pickle`. If they had another project that was in the BSD layer that interfaced with this one, then that BSD extension's class could be `com_apple_bsd_marketing_schlassen_pickle`.
- If you anticipate that the last part of the class name may be the same as the last part of another class name, consider beginning each function name with a larger portion of the class name. For example, you might have `bsd_pickle_eat` and `iokit_pickle_eat`.

These are only suggested guidelines. Your company or organization should adopt its own set of guidelines within the constraints of the basic conventions described in the previous section. These guidelines should provide a good starting point.

Standard C Naming Conventions

The naming conventions for C++ have been defined for some time in the document *Inside Mac OS X: Writing I/O Kit Drivers*. However, no conventions have been given for standard C code. Because standard C has an even greater chance of namespace collision than C++, it is essential that you follow these guidelines when writing C code for use in the kernel.

Because C does not have the benefit of classes, it is much easier to run into a naming conflict between two functions. For this reason, the following conventions are suggested:

- Declare all functions and (global) variables static where possible to prevent them from being seen in the global namespace. If you need to share these across files within your KEXT, you can achieve a similar effect by declaring them `__private_extern__`.
- Each function name should use Java-style reverse DNS naming. For example, if your company is apple.com, you should begin each function with `com_apple_`.
- Follow the reverse DNS name with the name of your project. For example, if you work at Apple and were working on project Schlassen, you would start each function name with `com_apple_iokit_schlassen_`.

- Use hierarchical names if you anticipate multiple projects with similar names coming from different parts of your company or organization.
- Use macro expansion to save typing, for example `PROJECT_eat` could expand to `com_apple_iokit_schlassen_pickle_eat`.
- If you anticipate that the last part of a function name may be the same as the last part of another function name (for example, `PROJECT1_eat` and `PROJECT2_eat`), you should change the names to avoid confusion (for example, `PROJECT1_eatpickle` and `PROJECT2_eatburger`).
- Avoid the following reserved prefixes:
  - OS
  - os
  - IO
  - io
  - Apple
  - apple
  - AAPL
  - aapl
- Avoid conflicting with any names already in the kernel, and do not use prefixes similar to those of existing kernel functions that you may be working with.
- Never begin a function name with an underscore (_).
- Under no circumstances should you use common names for your functions without prefixing them with the name of your project in some form. These are some examples of unacceptable names:
  - getuseridentity
  - get_user_info
  - print
  - find
  - search
  - sort
  - quicksort

- merge
- console_log

In short, picking any name that you would normally pick for a function is generally a bad idea, because every other developer writing code is likely to pick the same name for his or her function.

Occasional conflicts are a fact of life. However, by following these few simple rules, you should be able to avoid the majority of common namespace pitfalls.

Commonly Used Functions

One of the most common problems faced when programming in the kernel is use of "standard" functions—things like `printf` or `bcopy`. Many commonly used standard C library functions are implemented in the kernel. In order to use them, however, you need to include the appropriate prototypes, which may be different from the user space prototypes for those functions, and which generally have different names when included from kernel code.

In general, any non–I/O Kit header that you can safely include in the kernel is located in `xnu/bsd/sys` or `xnu/osfmk/mach`, although there are a few specialized headers in other places like `libkern` and `libsa`. Normal headers (those in `/usr/include`) cannot be used in the kernel.

Table 5-1 lists some commonly used C functions, variables, and types, and gives the location of their prototypes.

Table 5-1 Commonly used C functions

Function name	Header path
`printf`	`<sys/systm.h>`
Buffer cache functions (`bread`, `bwrite`, and `brelse`)	`<sys/buf.h>`
Directory entries	`<sys/dirent.h>`

Table 5-1 Commonly used C functions

Function name	Header path
Error numbers	`<sys/errno.h>`
Kernel special variables	`<sys/kernel.h>`
Spinlocks	`<sys/lock.h>`
`malloc`	`<sys/malloc.h>`
Queues	`<sys/queue.h>`
Random number generator	`<sys/rand.h>`
`bzero`, `bcopy`, `copyin`, and `copyout`	`<sys/systm.h>`
`timeout` and `untimeout`	`<sys/system.h>`
Various time functions	`<sys/time.h>`
Standard type declarations	`<sys/types.h>` `<mach/mach_types.h>`
User credentials	`<sys/ucred.h>`
OS and system information	`<sys/utsname.h>`

If the standard C function you are trying to use is not in one of these files, chances are the function is not supported for use within the kernel, and you need to implement your code in another way.

Performance and Stability Tips

This section includes some basic tips on performance and stability. You should read the sections on security and performance for additional information. These tips cover only style issues, not general performance or stability issues.

Performance and Stability Tips

Programming in the kernel is subject to a number of restrictions that do not exist in application programming. The first and most important is the stack size. The kernel has a limited amount of space allocated for thread stacks, which can cause problems if you aren't aware of the limitation. This means the following:

- Recursion *must* be bounded (to no more than a few levels).
- Recursion should be rewritten as iterative routines where possible.
- Large stack variables (function local) are dangerous. Do not use them. This also applies to large local arrays.
- Dynamically allocated variables are preferred (using `malloc` or equivalent) over local variables for objects more than a few bytes in size.
- Functions should have as few arguments as possible.
  - Pass pointers to structures, not the broken out elements.
  - Don't use arguments to avoid using global or class variables.
  - Do name global variables in a way that protects you from collision.
- C++ functions should be declared static.
- Functions not obeying these rules can cause a kernel panic, or in extreme cases, do not even compile.

In addition to issues of stack size, you should also avoid doing anything that would generate unnecessary load such as polling a device or address. A good example is the use of mutexes rather than spinlocks. You should also structure your locks in such a way to minimize contention and to minimize hold times on the most highly contended locks.

Also, since unused memory (and particularly wired memory) can cause performance degradation, you should be careful to deallocate memory when it is no longer in use, and you should never allocate large regions of wired memory. This may be unavoidable in some applications, but should be avoided whenever possible and disposed of at the earliest possible opportunity. Allocating large contiguous blocks of memory at boot time is almost never acceptable, because it cannot be released.

Finally, the kernel takes a speed penalty whenever floating-point math is used in the kernel, as floating-point registers are only maintained when they are in use. Additional code is also required to use floating-point in the kernel. Where possible, you should avoid doing floating-point math in the kernel. It is not forbidden, but is strongly discouraged.

Stability Tips

- Don't sleep while holding resources (locks, for example). While this is not forbidden, it is strongly discouraged to avoid deadlock.
- Be careful to allocate and free memory with matching calls. For example, do not use allocation routines from the I/O Kit and deallocation routines from BSD. Likewise, do not use `IOMallocContiguous` with `IOFreePageable`.
- Use reference counts to avoid freeing memory that is still in use elsewhere. Be sure to deallocate memory when its reference count reaches zero, but not before.
- Lock objects before operating on them, even to change reference counts.
- Never dereference pointers without verifying that they are not `NULL`. In particular, never do this:

  ```
  int foo = *argptr;
  ```

  unless you have already verified that `argptr` cannot possibly be `NULL`.
- Test code in sections and try to think up likely edge cases for calculations.
- Never assume that your code will be run only on big endian processors.
- Never assume that the size of an instance of a type will never change. Always use `sizeof` if you need this information.
- Never assume that a pointer will always be the same size as an `int` or `long`.

Style Summary

Kernel programming style is very much a matter of personal preference, and it is not practical to programmatically enforce the guidelines in this chapter. However, we strongly encourage you to follow these guidelines to the maximum extent possible. These guidelines were created based on frequent problems reported by developers writing code in the kernel. No one can force you to use good style in your programming, but if you do not, you do so at your own peril.

CHAPTER 6

Mach Overview

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 stand–alone 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 remote procedure calls **(RPC)** 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 (for 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. These are the main kernel abstractions:

- **Tasks**. The units of resource ownership; each task consists of a virtual address space, a **port right namespace**, and one or more **threads**. (Similar to a process.)
- **Threads**. The units of CPU execution within a task.
- **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 (known as port 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 a point of control flow in a task. 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 virtual 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 computer, 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 **time-sharing** and **fixed-priority** policies. A time-sharing 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.

Mac OS X also provides time constraint scheduling for real-time performance. This scheduling allows you to specify that your thread must get a certain time quantum within a certain period of time.

Mach scheduling is described further in "Mach Scheduling and Thread Interfaces" (page 67).

Ports, Port Rights, Port Sets, and Port Namespaces

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. This is comparable to a (unidirectional) pipe in UNIX parlance.

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 kernel is the receiver for ports associated with kernel-provided objects and that the receiver for ports associated with task-provided objects is the task providing those objects.

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 so that you can obtain information about it or perform other non-privileged operations against 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 "Interprocess Communication (IPC)" (page 49), 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 namespace. A port right is specified by a **port name**, an integer index into a 32-bit port namespace. Each task has associated with it a single port namespace.

Tasks acquire port rights when another task explicitly inserts them into its namespace, 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 taking a requested virtual address and assigning it a corresponding 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 entries** or **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.

Beginning in Mac OS X 10.1, the EMMI system was enhanced to support "portless" EMMI. In traditional EMMI, two Mach ports were created for each memory region, and likewise two ports for each cached vnode. Portless EMMI, in its initial implementation, replaces this with direct memory references (basically pointers). In a future release, ports will be used for communication with pagers outside the kernel, while using direct references for communication with pagers that reside in kernel space. The net result of these changes is that early versions of portless EMMI do not support pagers running outside of kernel space. This support is expected to be reinstated in a future release.

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.

Interprocess 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 threads currently waiting in that semaphore's wait queue. A post all operation wakes up all currently waiting threads.

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 an object that provides mutually exclusive access to a critical section. The primary interfaces to locks are transaction oriented (see "IPC Transactions and Event Dispatching" (page 50)). During the transaction, the thread holds the lock. When it returns from the transaction, the lock is released.

Remote Procedure Call (RPC) Objects

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 50))

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` API is deprecated in Mac OS X. The newer and preferred API is 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.

CHAPTER 7

Memory and Virtual Memory

This chapter describes allocating memory and the low-level routines for modifying memory maps in the kernel. It also describes a number of commonly used interfaces to the virtual memory system. It does not describe how to make changes in paging policy or add additional pagers. Mac OS X does not support external pagers, although much of the functionality can be achieved in other ways, some of which are covered at a high level in this chapter. The implementation details of these interfaces are subject to change, however, and are thus left undocumented.

With the exception of the section "Allocating Memory in the Kernel" (page 64), this chapter is of interest only if you are writing file systems or are modifying the virtual memory system itself.

Mac OS X VM Overview

The VM system used in Mac OS X is a descendent of Mach VM, which was created at Carnegie Mellon University in the 1980s. To a large extent, the fundamental design is the same, although some of the details are different, particularly when enhancing the VM system. It does, however, support the ability to request certain paging behavior through the use of **universal page lists (UPLs)**. See "Universal Page Lists (UPLs)" (page 58) for more information.

The design of Mach VM centers around the concept of physical memory being a cache for virtual memory.

At its highest level, Mach VM consists of address spaces and ways to manipulate the contents of those address spaces from outside the space. These address spaces are sparse and have a notion of protections to limit what tasks can access their contents.

At a lower level, the object level, virtual memory is seen as a collection of VM objects and memory objects, each with a particular owner and protections. These objects can be modified with object calls that are available both to the task and (via the back end of the VM) to the pagers.

The calls available to an application include `vm_map` and `vm_allocate`, which can be used to map file data or anonymous memory into the address space. This is possible only because the address space is initially sparse. In general, an application can either map a file into its address space (through file mapping primitives, abstracted by BSD) or it can map an object (after being passed a handle to that object). In addition, a task can change the protections of the objects in its address space and can share those objects with other tasks.

In addition to the mapping and allocation aspects of virtual memory, the VM system contains a number of other subsystems. These include the back end (pagers) and the shared memory subsystem. There are also other subsystems closely tied to VM, including the VM shared memory server. These are described in "Other VM and VM-Related Subsystems" (page 62).

The VM object is internal to the virtual memory system, and includes basic information about accessing the memory. The memory object is provided by the pager. The contents of the memory associated with that memory object can be retrieved from disk or some other backing store by exchanging messages with the memory object. Implicitly, each VM object is associated with a given pager through its memory object.

VM objects are cached with system pages (RAM), which can be any power of two multiple of the hardware page size. In the Mac OS X kernel, system pages are the same size as hardware pages. Each system page is represented in a given address space by a map entry. Each map entry has its own protection and inheritance. A given map entry can have an inheritance of `shared`, `copy`, or `none`. If a page is marked `shared` in a given map, child tasks share this page for reading and writing. If a page is marked `copy`, child tasks get a copy of this page (using copy-on-write). If a page is marked `none`, the child's page is left unallocated.

VM objects are managed by the machine-independent VM system, with the underlying virtual to physical mappings handled by the machine-dependent **pmap system**. The `pmap` system actually handles page tables, translation lookaside buffers, segments, and so on, depending on the design of the underlying hardware.

When a VM object is duplicated (for example, the data pages from a process that has just called `fork`), a **shadow object** is created. A shadow object is initially empty, and contains a reference to another object. When the contents of a page are modified, the page is copied from the parent object into the shadow object and then modified. When reading data from a page, if that page exists in the shadow object, the page listed in the shadow object is used. If the shadow object has no copy of that page, the original object is consulted. A series of shadow objects pointing to shadow objects or original objects is known as a **shadow chain**.

Shadow chains can become arbitrarily long if an object is heavily reused in a copy-on-write fashion. However, since `fork` is frequently followed by `exec`, which replaces all of the material being shadowed, long chains are rare. Further, Mach automatically garbage collects shadow objects, removing any intermediate shadow objects whose pages are no longer referenced by any (nondefunct) shadow object. It is even possible for the original object to be released if it no longer contains pages that are relevant to the chain.

Memory Maps Explained

Each Mach task has its own memory map. In Mach, this memory map takes the form of an ordered doubly linked list. As described in "Mac OS X VM Overview" (page 53), each of these objects contains a list of pages and shadow references to other objects.

In general, you should never need to access a memory map directly unless you are modifying something deep within the VM system. The `vm_map_entry` structure contains task-specific information about an individual mapping along with a reference to the backing object. In essence, it is the glue between an VM object and a VM map.

While the details of this data structure are beyond the scope of this document, a few fields are of particular importance.

The field `is_submap` is a Boolean value that tells whether this map entry is a normal VM object or a **submap**. A submap is a collection of mappings that is part of a larger map. Submaps are often used to group mappings together for the purpose of sharing them among multiple Mach tasks, but they may be used for many purposes. What makes a submap particularly powerful is that when several tasks have mapped a submap into their address space, they can see each other's changes, not only to the contents of the objects in the map, but to the objects themselves. This means that as additional objects are added to or deleted from the submap, they appear in or disappear from the address spaces of all tasks that share that submap.

The field `behavior` controls the paging reference behavior of a specified range in a given map. This value changes how pageins are clustered. Possible values are `VM_BEHAVIOR_DEFAULT`, `VM_BEHAVIOR_RANDOM`, `VM_BEHAVIOR_SEQUENTIAL`, and `VM_BEHAVIOR_RSEQNTL`, for default, random, sequential, or reverse-sequential pagein ordering.

The `protection` and `max_protection` fields control the permissions on the object. The `protection` field indicates what rights the task currently has for the object, while the `max_protection` field contains the maximum access that the current task can obtain for the object.

You might use the `protection` field when debugging shared memory. By setting the protection to be read-only, any inadvertent writes to the shared memory would cause an exception. However, when the task actually needs to write to that shared region, it could increase its permissions in the `protection` field to allow writes.

It would be a security hole if a task could increase its own permissions on a memory object arbitrarily, however. In order to preserve a reasonable security model, the task that owns a memory object must be able to limit the rights granted to a subordinate task. For this reason, a task is not allowed to increase its protection beyond the permissions granted in `max_protection`.

Possible values for `protection` and `max_protection` are described in detail in `xnu/osfmk/mach/vm_prot.h`.

Finally, the `use_pmap` field indicates whether a submap's low-level mappings should be shared among all tasks into which the submap is mapped. If the mappings are not shared, then the structure of the map is shared among all tasks, but the actual contents of the pages are not.

For example, shared libraries are handled with two submaps. The read-only shared code section has `use_pmap` set to true. The read-write (nonshared) section has `use_pmap` set to false, forcing a clean copy of the library's `DATA` segment to be mapped in from disk for each new task.

Named Entries

The Mac OS X VM system provides an abstraction known as a **named entry**. A named entry is nothing more than a handle to a shared object or a submap.

Shared memory support in Mac OS X is achieved by sharing objects between the memory maps of various tasks. Shared memory objects must be created from existing VM objects by calling `vm_allocate` to allocate memory in your address space and then calling `mach_make_memory_entry_64` to get a handle to the underlying VM object.

The handle returned by `mach_make_memory_entry_64` can be passed to `vm_map` to map that object into a given task's address space. The handle can also be passed via IPC or other means to other tasks so that they can map it into their address spaces. This provides the ability to share objects with tasks that are not in your direct lineage, and also allows you to share additional memory with tasks in your direct lineage after those tasks are created.

The other form of named entry, the submap, is used to group a set of mappings. The most common use of a submap is to share mappings among multiple Mach tasks. A submap can be created with `vm_region_object_create`.

What makes a submap particularly powerful is that when several tasks have mapped a submap into their address space, they can see each other's changes to both the data and the structure of the map. This means that one task can map or unmap a VM object in another task's address space simply by mapping or unmapping that object in the submap.

Universal Page Lists (UPLs)

A universal page list, or UPL, is a data structure used when communicating with the virtual memory system. UPLs can be used to change the behavior of pages with respect to caching, permissions, mapping, and so on. UPLs can also be used to push data into and pull data from VM objects. The term is also often used to refer to the family of routines that operate on UPLs. The flags used when dealing with UPLs are described in `osfmk/mach/memory_object_types.h`.

The life cycle of a UPL looks like this:

1. A UPL is created based on the contents of a VM object. This UPL includes information about the pages within that object.
2. That UPL is modified in some way.
3. The changes to the UPL are either committed (pushed back to the VM system) or aborted, with `ubc_upl_commit` or `ubc_upl_abort`, respectively.

If you have a control handle for a given VM object (which generally means that you are inside a pager), you can use `vm_object_upl_request` to get a UPL for that object. Otherwise, you must use the `vm_map_get_upl` call. In either case, you are left with a handle to the UPL.

When a pagein is requested, the pager receives a list of pages that are locked against the object, with certain pages set to not valid. The pager must either write data into those pages or must abort the transaction to prevent invalid data in the kernel. Similarly in pageout, the kernel must write the data to a backing store or abort the transaction to prevent data loss. The pager may also elect to bring additional pages into memory or throw additional pages out of memory at its discretion.

Because pagers can be used both for virtual memory and for memory mapping of file data, when a pageout is requested, the data may need to be freed from memory, or it may be desirable to keep it there and simply flush the changes to disk. For this reason, the flag `UPL_CLEAN_IN_PLACE` exists to allow a page to be flushed to disk but not removed from memory.

When a pager decides to page in or out additional pages, it must determine which pages to move. A pager can request all of the dirty pages by setting the `RETURN_ONLY_DIRTY` flag. It can also request all pages that are not in memory using the `RETURN_ONLY_ABSENT` flag.

There is a slight problem, however. If a given page is marked as `BUSY` in the UPL, a request for information on that page would normally block. If the pager is doing prefetching or preflushing, this is not desirable, since it might be blocking on itself or on some other pager that is blocked waiting for the current transaction to complete. To avoid such deadlock, the UPL mechanism provides the `UPL_NOBLOCK` flag. This is frequently used in the anonymous pager for requesting free memory.

The flag `QUERY_OBJECT_TYPE` can be used to determine if an object is physically contiguous and to get other properties of the underlying object.

The flag `UPL_PRECIOUS` means that there should be only one copy of the data. This prevents having a copy both in memory and in the backing store. However, this breaks the adjacency of adjacent pages in the backing store, and is thus generally not used to avoid a performance hit.

The flag `SET_INTERNAL` is used by the BSD subsystem to cause all information about a UPL to be contained in a single memory object so that it can be passed around more easily. It can only be used if your code is running in the kernel's address space.

Since this handle can be used for multiple small transactions (for example, when mapping a file into memory block-by-block), the UPL API includes functions for committing and aborting changes to only a portion of the UPL. These functions are `upl_commit_range` and `upl_abort_range`, respectively.

To aid in the use of UPLs for handling multi-part transactions, the `upl_commit_range` and `upl_abort_range` calls have a flag that causes the UPL to be freed when there are no unmodified pages in the UPL. If you use this flag, you must be very careful not to use the UPL after all ranges have been committed or aborted.

Finally, the function `vm_map_get_upl` is frequently used in file systems. It gets the underlying VM object associated with a given range within an address space. Since this returns only the first object in that range, it is your responsibility to determine whether the entire range is covered by the resulting UPL and, if not, to make additional calls to get UPLs for other objects. Note that while the `vm_map_get_upl` call is against an address space range, most UPL calls are against a `vm_object`.

Using Mach Memory Maps

WARNING
This section describes the low-level API for dealing with Mach VM maps. These maps cannot be modified in this way from a kernel extension. These functions are not available for use in a KEXT. They are presented strictly for use within the VM system and other parts of Mach. If you are not doing in-kernel development, you should be using the methods described in the chapter "Boundary Crossings" (page 107).

From the context of the kernel (*not* from a KEXT), there are two maps that you will probably need to deal with. The first is the kernel map. Since your code is executing in the kernel's address space, no additional effort is needed to use memory referenced in the kernel map. However, you may need to add additional mappings into the kernel map and remove them when they are no longer needed.

The second map of interest is the memory map for a given task. This is of most interest for code that accepts input from user programs, for example a `sysctl` or a Mach RPC handler. In nearly all cases, convenient wrappers provide the needed functionality, however.

The low-level VM map API includes the following functions:

```
kern_return_t vm_map_copyin(vm_map_t src_map, vm_offset_t src_addr,
            vm_size_t len, boolean_t src_destroy,
            vm_map_copy_t *copy_result);

kern_return_t vm_map_copyout(vm_map_t map, vm_offset_t *addr, /* Out */
            register vm_map_copy_t copy);

kern_return_t vm_map_copy_overwrite(vm_map_t dst_map,
            vm_offset_t dst_address,vm_map_copy_t copy,
            boolean_t interruptible, pmap_t pmap);

void vm_map_copy_discard(vm_map_copy_t copy);

void vm_map_wire(vm_map_t map, vm_offset_t start, vm_offset_t end,
```

```
            vm_prot_t access_type, boolean_t user_wire);

void vm_map_unwire(vm_map_t map, vm_offset_t start, vm_offset_t end,
            boolean_t user_wire);
```

The function `vm_map_copyin` copies data from an arbitrary (potentially non–kernel) memory map into a copy list and returns the copy list pointer in `copy_result`. If something goes wrong and you need to throw away this intermediate object, it should be freed with `vm_map_copy_discard`.

In order to actually get the data from the copy list, you need to overwrite a memory object in the kernel's address space with `vm_map_copy_overwrite`. This overwrites an object with the contents of a copy list. For most purposes, the value passed for `interruptible` should be `FALSE`, and `pmap` should be `NULL`.

Copying data from the kernel to user space is exactly the same as copying data from user space, except that you pass `kernel_map` to `vm_map_copyin` and pass the user map to `vm_map_copy_overwrite`. In general, however, you should avoid doing this, since you could end up with a task's memory being fragmented into lots of tiny objects, which is undesirable.

Do *not* use `vm_map_copyout` when copying data into an existing user task's address map. The function `vm_map_copyout` is used for filling an unused region in an address map. If the region is allocated, then `vm_map_copyout` does nothing. Because it requires knowledge of the current state of the map, it is primarily used when creating a new address map (for example, if you are manually creating a new process). For most purposes, you do not need to use `vm_map_copyout`.

The functions `vm_map_wire` and `vm_map_unwire` can be used to wire and unwire portions of an address map. If you set the argument `user_wire` to `TRUE`, then the page can be unwired from user space. This should be set to `FALSE` if you are about to use the memory for I/O or for some other operation that cannot tolerate paging. In `vm_map_wire`, the argument `access_type` indicates the types of accesses that should not be allowed to generate a page fault. In general, however, you should be using `vm_wire` to wire memory.

As mentioned earlier, this information is presented strictly for use in the heart of the kernel. You should not use anything in this section from a kernel extension.

Other VM and VM-Related Subsystems

There are two additional VM subsystems: pagers and the working set detection subsystem. In addition, the VM shared memory server subsystem is closely tied to (but is not part of) the VM subsystem. This section describes these three VM and VM-related subsystems.

Pagers

Mac OS X has three basic pagers: the vnode pager, the default pager (or anonymous pager), and the device pager. These are used by the VM system to actually get data into the VM objects that underlie named entries. Pagers are linked into the VM system through a combination of a subset of the old Mach pager interface and UPLs.

The default pager is what most people think of when they think of a VM system. It is responsible for moving normal data into and out of the backing store. In addition, there is a facility known as the dynamic pager that sits on top of the default pager and handles the creation and deletion of backing store files. These pager files are filled with data in clusters (groups of pages).

When the total fullness of the paging file pool reaches a high–water mark, the default pager asks the dynamic pager to allocate a new store file. When the pool drops below its low water mark, the VM system selects a pager file, moves its contents into other pager files, and deletes it from disk.

The vnode pager has a 1:1 (onto) mapping between objects in VM space and open files (vnodes). It is used for memory mapped file I/O. The vnode pager is generally hidden behind calls to BSD file APIs.

The device pager allows you to map non–general-purpose memory with the cache characteristics required for that memory (**WIMG**). Non–general–purpose memory includes physical addresses that are mapped onto hardware other than main memory—for example, PCI memory, frame buffer memory, and so on. The device pager is generally hidden behind calls to various I/O Kit functions.

Working Set Detection Subsystem

To improve performance, Mac OS X has a subsystem known as the working set detection subsystem. This subsystem is called on a VM fault; it keeps a profile of the fault behavior of each task from the time of its inception. In addition, just before a page request, the fault code asks this subsystem which adjacent pages should be brought in, and then makes a single large request to the pager.

Since files on disk tend to have fairly good locality, and since address space locality is largely preserved in the backing store, this provides a substantial performance boost. Also, since it is based upon the application's previous behavior, it tends to pull in pages that would probably have otherwise been needed later. This occurs for all pagers.

The working set code works well once it is established. However, without help, its performance would be the baseline performance until a profile for a given application has been developed. To overcome this, the first time that an application is launched in a given user context, the initial working set required to start the application is captured and stored in a file. From then on, when the application is started, that file is used to seed the working set.

These working set files are established on a per-user basis. They are stored in `/var/vm/app_profile` and are only accessible by the super-user (and the kernel).

VM Shared Memory Server Subsystem

The VM shared memory server subsystem is a BSD service that is closely tied to VM, but is not part of VM. This server provides two submaps that are used for shared library support in Mac OS X. Because shared libraries contain both read-only portions (text segment) and read-write portions (data segment), the two portions are treated separately to maximize efficiency. The read-only portions are completely shared between tasks, including the underlying `pmap` entries. The read-write portions share a common submap, but have different underlying data objects (achieved through copy-on-write).

The three functions exported by the VM shared memory server subsystem should only be called by `dyld`. Do not use them in your programs.

The function `load_shared_file` is used to load a new shared library into the system. Once such a file is loaded, other tasks can then depend on it, so a shared library cannot be unshared. However, a new set of shared regions can be created with `new_system_shared_regions` so that no new tasks will use old libraries.

The function `reset_shared_file` can be used to reset any changes that your task may have made to its private copy of the data section for a file.

Finally, the function `new_system_shared_regions` can be used to create a new set of shared regions for future tasks. New regions can be used when updating prebinding with new shared libraries to cause new tasks to see the latest libraries at their new locations in memory. (Users of old shared libraries will still work, but they will fall off the pre-bound path and will perform less efficiently.) It can also be used when dealing with private libraries that you want to share only with your task's descendents.

Allocating Memory in the Kernel

As with most things in the Mac OS X kernel, there are a number of ways to allocate memory. The choice of routines depends both on the location of the calling routine and on the reason for allocating memory. In general, you should use Mach routines for allocating memory unless you are writing code for use in the I/O Kit, in which case you should use I/O Kit routines.

Allocating Memory Using Mach Routines

Mach routines provide a relatively straightforward interface for allocating and releasing memory. They are the preferred mechanism for allocating memory outside of the I/O Kit. BSD also offers `_MALLOC` and `_FREE`, which may be used in BSD parts of the kernel.

They do not provide for forced mapping of a given physical address to a virtual address. However, if you need such a mapping, you are probably writing a device driver, in which case you should be using I/O Kit routines instead of Mach routines.

These are some of the commonly used Mach routines for allocating memory:

```
kern_return_t kmem_alloc(vm_map_t map, vm_offset_t *addrp, vm_size_t size);
void kmem_free(vm_map_t map, vm_offset_t addr, vm_size_t size);
kern_return_t mem_alloc_aligned(vm_map_t map, vm_offset_t *addrp,
            vm_size_t size);
kern_return_t kmem_alloc_wired(vm_map_t map, vm_offset_t *addrp,
            vm_size_t size);
kern_return_t kmem_alloc_pageable(vm_map_t map, vm_offset_t *addrp,
            vm_size_t size);
kern_return_t kmem_alloc_contig(vm_map_t map, vm_offset_t *addrp,
            vm_size_t size, vm_offset_t mask, int flags);
```

These functions all take a map as the first argument. Unless you need to allocate memory in a different map, you should pass `kernel_map` for this argument.

All of the `kmem_alloc` functions except `kmem_alloc_pageable` allocate wired memory. The function `kmem_alloc_pageable` creates the appropriate VM structures but does not back the region with physical memory. This function could be combined with `vm_map_copyout` when creating a new address map, for example. In practice, it is rarely used.

The function `kmem_alloc_aligned` allocates memory aligned according to the value of the `size` argument, which must be a power of 2.

The function `kmem_alloc_wired` is synonymous with `kmem_alloc` and is appropriate for data structures that cannot be paged out. It is not strictly necessary; however, if you explicitly need certain pieces of data to be wired, using `kmem_alloc_wired` makes it easier to find those portions of your code.

The function `kmem_alloc_contig` attempts to allocate a block of physically contiguous memory. This is not always possible, and requires a full sort of the system free list even for short allocations. After startup, this sort can cause long delays, particularly on systems with lots of RAM. You should generally not use this function.

The function `kmem_free` is used to free an object allocated with one of the `kmem_alloc` functions. Unlike the standard C `free` function, `kmem_free` requires the length of the object. If you are not allocating fixed-size objects (for example, `sizeof struct foo`), you may have to do some additional bookkeeping, since you must free an entire object, not just a portion of one.

Allocating Memory From the I/O Kit

Although the I/O Kit is generally beyond the scope of this document, the I/O Kit memory management routines are presented here for completeness. In general, I/O Kit routines should not be used outside the I/O Kit. Similarly, Mach allocation routines should not be directly used from the I/O Kit because the I/O Kit has abstractions for those routines that fit the I/O Kit development model more closely.

The I/O Kit includes the following routines for kernel memory allocation:

```
void *IOMalloc(vm_size_t size);
void *IOMallocAligned(vm_size_t size, vm_size_t alignment);
void *IOMallocContiguous(vm_size_t size, vm_size_t alignment,
            IOPhysicalAddress *physicalAddress);
void *IOMallocPageable(vm_size_t size, vm_size_t alignment);
void IOFree(void *address, vm_size_t size);
void IOFreeAligned(void *address, vm_size_t size);
void IOFreeContiguous(void *address, vm_size_t size);
void IOFreePageable(void *address, vm_size_t size);
```

Most of these routines are relatively transparent wrappers around the Mach allocation functions. There are two major differences, however. First, the caller does not need to know which memory map is being modified. Second, they have a separate free call for each allocation call for internal bookkeeping reasons.

The functions `IOMallocContiguous` and `IOMallocAligned` differ somewhat from their Mach underpinnings. `IOMallocAligned` uses calls directly to Mach VM to add support for arbitrary (power of 2) data alignment, rather than aligning based on the size of the object. `IOMallocContiguous` adds an additional parameter, `PhysicalAddress`. If this pointer is not `NULL`, the physical address is returned through this pointer. Using Mach functions, obtaining the physical address requires a separate function call.

Important
If your KEXT allocates memory that will be shared, you should create a buffer of type `IOMemoryDescriptor` or `BufferMemoryDescriptor` and specify that the buffer should be sharable. If you are allocating memory in a user application that will be shared with the kernel, you should use `valloc` or `vm_allocate` instead of `malloc` and then call `mach_make_memory_entry_64`.

CHAPTER 8

Mach Scheduling and Thread Interfaces

Mac OS X is based on Mach and BSD. Like Mach and most BSD UNIX systems, it contains an advanced scheduler based on the CMU Mach 3 scheduler. This chapter describes the scheduler from the perspective of both a kernel programmer and an application developer attempting to set scheduling parameters.

This chapter begins with the "Overview of Scheduling" (page 67), which describes the basic concepts behind Mach scheduling at a high level, including real-time priority support.

The second section, "Using Mach Scheduling From User Applications" (page 69), describes how to access certain key Mach scheduler routines from user applications and from other parts of the kernel outside the scheduler.

The third section, "Kernel Thread APIs" (page 75), explains scheduler-related topics including how to create and terminate kernel threads and describes the BSD `spl` macros and their limited usefulness in Mac OS X.

Overview of Scheduling

The Mac OS X scheduler is derived from the scheduler used in OSFMK 7.3. In general, much documentation about prior implementations applies to the scheduler in Mac OS X, although you will find numerous differences. The details of those differences are beyond the scope of this overview.

Mach scheduling is based on a system of run queues at various priorities that are handled in different ways. The priority levels are divided into four bands according to their characteristics, as described in Table 8-1.

Table 8-1 Thread priority bands

Priority Band	Characteristics
Normal	normal application thread priorities
System high priority	threads whose priority has been raised above normal threads
Kernel mode only	reserved for threads created inside the kernel that need to run at a higher priority than all user space threads (I/O Kit workloops, for example)
Real-time threads	threads whose priority is based on getting a well-defined fraction of total clock cycles, regardless of other activity (in an audio player application, for example).

Threads can migrate between priority levels for a number of reasons, largely as an artifact of the time sharing algorithm used. However, this migration is within a given band.

Threads marked as being real-time priority are also special in the eyes of the scheduler. A real-time thread tells the scheduler that it needs to run for A cycles out of the next B cycles. For example, it might need to run for 3000 out of the next 7000 clock cycles in order to keep up. It also tells the scheduler whether those cycles must be contiguous. Using long contiguous quanta is generally frowned upon but is occasionally necessary for specialized real-time applications.

If the real-time thread requests something relatively reasonable, its priority will remain in the real-time band. If it lies blatantly about its requirements and behaves in a compute-bound fashion, it may be demoted to the priority of a normal thread.

Changing a thread's priority to turn it into a real-time priority thread using Mach calls is described in more detail in "Using Mach Scheduling From User Applications" (page 69).

In addition to the raw Mach RPC interfaces, some aspects of a thread's priority can be controlled from user space using the POSIX thread priority API. The POSIX thread API is able to set thread priority only within the lowest priority band (0–63).

Why Did My Thread Priority Change?

There are many reasons that a thread's priority can change. This section attempts to explain the root cause of these thread priority changes.

A real-time thread, as mentioned previously, is penalized (and may even be knocked down to normal thread priority) if it exceeds its time quantum without blocking repeatedly. For this reason, it is very important to make a reasonable guess about your thread's workload if it needs to run in the real-time band.

Threads that are heavily compute-bound are given lower priority to help minimize response time for interactive tasks so that high–priority compute–bound threads cannot monopolize the system and prevent lower–priority I/O-bound threads from running. Even at a lower priority, the compute–bound threads still run frequently, since the higher–priority I/O-bound threads do only a short amount of processing, block on I/O again, then allow the compute-bound threads to execute.

All of these mechanisms are operating continually in the Mach scheduler. This means that threads are frequently moving up or down in priority based upon their behavior and the behavior of other threads in the system.

Using Mach Scheduling From User Applications

There are three basic ways to change how a user thread is scheduled. You can use the BSD `pthreads` API to change basic policy and importance. You can also use Mach RPC calls to change a task's importance. Finally, you can use RPC calls to change the scheduling policy to move a thread into a different scheduling band. This is commonly used when interacting with CoreAudio.

The pthreads API is a user space API, and has limited relevance for kernel programmers. The Mach thread and task APIs are more general and can be used from anywhere in the kernel. The Mach thread and task calls can also be called from user applications.

Using the pthreads API to Influence Scheduling

Mac OS X supports a number of policies at the pthreads API level. If you need real-time behavior, you must use the Mach thread_policy_set call. This is described in "Using the Mach Thread API to Influence Scheduling" (page 71).

The pthreads API adjusts the priority of threads within a given task. It does not necessarily impact performance relative to threads in other tasks. To increase the priority of a task, you can use nice or renice from the command line or call getpriority and setpriority from your application.

The API provides two functions: pthread_attr_getschedparam and pthread_setschedparam. Their prototypes look like this:

```
pthread_setschedparam(pthread_t thread, int policy,
        struct sched_param *param);
pthread_getschedparam(pthread_t thread, int *policy,
        struct sched_param *param)
```

The arguments for pthread_getschedparam are straightforward. The first argument is a thread ID, and the others are pointers to memory where the results will be stored.

The arguments to pthread_setschedparam are not as obvious, however. As with pthread_getschedparam, the first argument is a thread ID.

The second argument to pthread_setschedparam is the desired policy, which can currently be one of SCHED_FIFO (first in, first out), SCHED_RR (round-robin), or SCHED_OTHER. The SCHED_OTHER policy is generally used for extra policies that are specific to a given operating system, and should thus be avoided when writing portable code.

The third argument is a structure that contains various scheduling parameters.

Here is a basic example of using pthreads functions to set a thread's scheduling policy and priority.

```
int set_my_thread_priority(int priority) {
    struct sched_param sp;

    memset(&sp, 0, sizeof(struct sched_param));
    sp.sched_priority=priority;
    if (pthread_setschedparam(pthread_self(), SCHED_RR, &sp) == -1) {
        printf("Failed to change priority.\n");
        return -1;
    }
    return 0;
}
```

This code snippet sets the scheduling policy for the current thread to round-robin scheduling, and sets the thread's relative importance within the task to the value passed in through the `priority` argument.

Using the Mach Thread API to Influence Scheduling

This API is frequently used in multimedia applications to obtain real-time priority. It is also useful in other situations when the `pthread` scheduling API cannot be used or does not provide the needed functionality.

The API consists of two functions, `thread_policy_set` and `thread_policy_get`.

```
kern_return_t thread_policy_set(
                thread_act_t thread,
                thread_policy_flavor_t flavor,
                thread_policy_t policy_info,
                mach_msg_type_number_t count);

kern_return_t thread_policy_get(
                thread_act_t thread,
                thread_policy_flavor_t flavor,
                thread_policy_t policy_info,
                mach_msg_type_number_t *count,
                boolean_t *get_default);
```

The parameters of these functions are roughly the same, except that the `thread_policy_get` function takes pointers for the `count` and the `get_default` arguments. The count is an `inout` parameter, meaning that it is interpreted as the

maximum amount of storage that the calling task has allocated for the return, but it is also overwritten by the scheduler to indicate the amount of data that was actually returned.

These functions get and set several parameters, according to the thread policy chosen. The possible thread policies are listed in Table 8-2.

Table 8-2 Thread policies

Policy	Meaning
THREAD_STANDARD_POLICY	Default value
THREAD_TIME_CONSTRAINT_POLICY	Used to specify real-time behavior.
THREAD_PRECEDENCE_POLICY	Used to indicate the importance of computation relative to other threads in a given task.

The following code snippet shows how to set the priority of a task to tell the scheduler that it needs real-time performance. The example values provided in comments are based on the estimated needs of `esd` (the Esound daemon).

```
#include <mach/mach_init.h>
#include <mach/thread_policy.h>
#include <mach/sched.h>

int set_realtime(int period, int computation, int constraint) {
    struct thread_time_constraint_policy ttcpolicy;
    int ret;

    ttcpolicy.period=period; // HZ/160
    ttcpolicy.computation=computation; // HZ/3300;
    ttcpolicy.constraint=constraint; // HZ/2200;
    ttcpolicy.preemptible=1;

    if ((ret=thread_policy_set(mach_thread_self(),
        THREAD_TIME_CONSTRAINT_POLICY, (int *)&ttcpolicy,
        THREAD_TIME_CONSTRAINT_POLICY_COUNT)) != KERN_SUCCESS) {
            fprintf(stderr, "set_realtime() failed.\n");
```

```
            return 0;
    }
    return 1;
}
```

The time values are in terms of Mach absolute time units. Since these values differ according to the bus speed of your computer, you should generally use numbers relative to HZ (ticks per second). These are discussed in more detail in "Using Kernel Time Abstractions" (page 139).

Say your computer reports 133 million for the value of HZ. If you pass the example values given as arguments to this function, your thread tells the scheduler that it needs approximately 40,000 out of the next 833,333 bus cycles. The `preemptible` value indicates that those 40,000 bus cycles need not be contiguous. However, the `constraint` value tells the scheduler that there can be no more than 60,000 bus cycles between the start of computation and the end of computation.

A straightforward example using this API is code that displays video directly to the framebuffer hardware. It needs to run for a certain number of cycles every frame to get the new data into the frame buffer. It can be interrupted without worry, but if it is interrupted for too long, the video hardware starts displaying an outdated frame before the software writes the updated data, resulting in a nasty glitch. Audio has similar behavior, but since it is usually buffered along the way (in hardware and in software), there is greater tolerance for variations in timing, to a point.

Another policy call is `THREAD_PRECEDENCE_POLICY`. This is used for setting the relative importance of non-real-time threads. Its calling convention is similar, except that its structure is `thread_precedence_policy`, and contains only one field, an `integer_t` called `importance`, which is a signed 32-bit value.

Using the Mach Task API to Influence Scheduling

This relatively simple API is not particularly useful for most developers. However, it may be beneficial if you are developing a graphical user interface for Darwin. It also provides some insight into the prioritization of tasks in Mac OS X. It is presented here for completeness.

The API consists of two functions, `task_policy_set` and `task_policy_get`.

```
kern_return_t task_policy_set(
                task_t task,
```

```
                task_policy_flavor_t flavor,
                task_policy_t policy_info,
                mach_msg_type_number_t count);

kern_return_t task_policy_get(
                task_t task,
                task_policy_flavor_t flavor,
                task_policy_t policy_info,
                mach_msg_type_number_t *count,
                boolean_t *get_default);
```

As with `thread_policy_set` and `thread_policy_get`, the parameters are similar, except that the `task_policy_get` function takes pointers for the `count` and the `get_default` arguments. The `count` argument is an `inout` parameter. It is interpreted as the maximum amount of storage that the calling task has allocated for the return, but it is also overwritten by the scheduler to indicate the amount of data that was actually returned.

These functions get and set a single parameter, that of the role of a given task, which changes the way the task's priority gets altered over time. The possible roles of a task are listed in Table 8-3.

Table 8-3 Task roles

Role	Meaning
`TASK_UNSPECIFIED`	Default value
`TASK_RENICED`	This is set when a process is executed with `nice` or is modified by `renice`.
`TASK_FOREGROUND_APPLICATION`	GUI application in the foreground. There can be more than one foreground application.
`TASK_BACKGROUND_APPLICATION`	GUI application in the background.
`TASK_CONTROL_APPLICATION`	Reserved for the dock or equivalent (assigned FCFS).
`TASK_GRAPHICS_SERVER`	Reserved for `WindowServer` or equivalent (assigned FCFS).

The following code snippet shows how to set the priority of a task to tell the scheduler that it is a foreground application (regardless of whether it really is).

```
#include <mach/mach_init.h>
#include <mach/task_policy.h>
#include <mach/sched.h>

int set_my_task_policy(void) {
    int ret;
    struct task_category_policy tcatpolicy;

    tcatpolicy.role = TASK_FOREGROUND_APPLICATION;

    if ((ret=task_policy_set(mach_task_self(),
        TASK_CATEGORY_POLICY, (int *)&tcatpolicy,
        TASK_CATEGORY_POLICY_COUNT)) != KERN_SUCCESS) {
            fprintf(stderr, "set_my_task_policy() failed.\n");
            return 0;
    }
    return 1;
}
```

Kernel Thread APIs

The Mac OS X scheduler provides a number of public APIs. While many of these APIs should not be used, the APIs to create, destroy, and alter kernel threads are of particular importance. While not technically part of the scheduler itself, they are inextricably tied to it.

The scheduler directly provides certain services that are commonly associated with the use of kernel threads, without which kernel threads would be of limited utility. For example, the scheduler provides support for wait queues, which are used in various synchronization primitives such as mutex locks and semaphores.

Creating and Destroying Kernel Threads

There are two basic interfaces for creating threads within the kernel. The I/O Kit provides `IOCreateThread`, `IOThreadSelf`, and `IOExitThread`, while Mach itself provides `kernel_thread` and `current_thread`. The basic functions for creating and terminating kernel threads are:

```
thread_t kernel_thread(task_t task, void (*start)(void));
thread_t current_thread(void);
IOThread IOCreateThread(IOThreadFunc function, void *argument);
IOThread IOThreadSelf(void);
void IOExitThread(void);
```

With the exception of `IOCreateThread` (which is a bit more complex), the I/O Kit functions are fairly thin wrappers around Mach thread functions. The types involved are also very thin abstractions. `IOThread` is really the same as `thread_t`.

The functions are relatively straightforward. The Mach functions allow you to create a kernel thread in which a function with no arguments executes, while the equivalent I/O Kit functions support a function with a single argument.

One other useful function is `thread_terminate`. This can be used to destroy an arbitrary thread (except, of course, the currently running thread). This can be *extremely* dangerous if not done correctly. Before tearing down a thread with `thread_terminate`, you should lock the thread and disable any outstanding timers against it. If you fail to deactivate a timer, a kernel panic will occur when the timer expires.

With that in mind, you may be able to terminate a thread as follows:

```
thread_terminate(getact_thread(thread_t thread));
```

In general, you can only be assured that you can kill yourself, not other threads in the system. The function `thread_terminate` takes a single parameter of type `thread_act_t` (a thread activation). The function `getact_thread` takes a thread shuttle (`struct thread_shuttle` or `thread_t`) and returns the thread activation associated with it.

SPL and Friends

BSD–based and Mach–based operating systems contain legacy functions designed for basic single-processor synchronization. These include functions such as `splhigh`, `splbio`, `splx`, and other similar functions. Since these functions are not particularly useful for synchronization in an SMP situation, they are not particularly useful as synchronization tools in Mac OS X.

If you are porting legacy code from earlier Mach–based or BSD–based operating systems, you must find an alternate means of providing synchronization. In many cases, this is as simple as taking the kernel or network funnel. In parts of the kernel, the use of `spl` functions does nothing, but causes no harm if you are holding a funnel (and results in a panic if you are not). In other parts of the kernel, `spl` macros are actually used. Because `spl` cannot necessarily be used for its intended purpose, it should not be used in general unless you are writing code it a part of the kernel that already uses it. You should instead use alternate synchronization primitives such as those described in "Synchronization Primitives" (page 129).

Wait Queues and Wait Primitives

The wait queue API is used extensively by the scheduler and is closely tied to the scheduler in its implementation. It is also used extensively in locks, semaphores, and other synchronization primitives. The wait queue API is both powerful and flexible, and as a result is somewhat large. Not all of the API is exported outside the scheduler, and parts are not useful outside the context of the wait queue functions themselves. This section documents only the public API.

The wait queue API includes the following functions:

```
void wait_queue_init(wait_queue_t wq, int policy);
extern wait_queue_t wait_queue_t wait_queue_alloc(int policy);
void wait_queue_free(wait_queue_t wq);
void wait_queue_lock(wait_queue_t wq);
void wait_queue_lock_try(wait_queue_t wq);
void wait_queue_unlock(wait_queue_t wq);
boolean_t wait_queue_member(wait_queue_t wq, wait_queue_sub_t wq_sub);
boolean_t wait_queue_member_locked(wait_queue_t wq, wait_queue_sub_t wq_sub);
kern_return_t wait_queue_link(wait_queue_t wq, wait_queue_sub_t wq_sub);
kern_return_t wait_queue_unlink(wait_queue_t wq, wait_queue_sub_t wq_sub);
kern_return_t wait_queue_unlink_one(wait_queue_t wq,
            wait_queue_sub_t *wq_subp);
```

```
void wait_queue_assert_wait(wait_queue_t wq, event_t event,
            int interruptible);
void wait_queue_assert_wait_locked(wait_queue_t wq, event_t event,
            int interruptible, boolean_t unlocked);
kern_return_t wait_queue_wakeup_all(wait_queue_t wq, event_t event,
            int result);
kern_return_t wait_queue_peek_locked(wait_queue_t wq, event_t event,
            thread_t *tp, wait_queue_t *wqp);
void wait_queue_pull_thread_locked(wait_queue_t wq, thread_t thread,
            boolean_t unlock);
thread_t wait_queue_wakeup_identity_locked(wait_queue_t wq, event_t event,
            int result, boolean_t unlock);
kern_return_t wait_queue_wakeup_one(wait_queue_t wq, event_t event,
            int result);
kern_return_t wait_queue_wakeup_one_locked(wait_queue_t wq, event_t event,
            int result, boolean_t unlock);
kern_return_t wait_queue_wakeup_thread(wait_queue_t wq, event_t event,
            thread_t thread, int result);
kern_return_t wait_queue_wakeup_thread_locked(wait_queue_t wq, event_t event,
            thread_t thread, int result, boolean_t unlock);
kern_return_t wait_queue_remove(thread_t thread);
```

Most of the functions are straightforward and are not presented in detail. However, a few require special attention.

Notice the functions ending in `_locked`. These functions require that your thread be holding a lock on the wait queue before they are called. Functions ending in `_locked` are equivalent to their nonlocked counterparts (where applicable) except that they do not lock the queue on entry and may not unlock the queue on exit (depending on the value of `unlock`). The remainder of this section does not differentiate between locked and unlocked functions.

The `wait_queue_alloc` and `wait_queue_init` functions take a policy parameter, which can be one of the following:

- `SYNC_POLICY_FIFO`—first-in, first-out
- `SYNC_POLICY_FIXED_PRIORITY`—policy based on thread priority
- `SYNC_POLICY_PREPOST`—keep track of number of wakeups where no thread was waiting and allow threads to immediately continue executing without waiting until that count reaches zero. This is frequently used when implementing semaphores.

You should not use the `wait_queue_init` function outside the scheduler. Because a wait queue is an opaque object outside that context, you cannot determine the appropriate size for allocation. Thus, because the size could change in the future, you should always use `wait_queue_alloc` and `wait_queue_free` unless you are writing code within the scheduler itself.

Similarly, the functions `wait_queue_member`, `wait_queue_member_locked`, `wait_queue_link`, `wait_queue_unlink`, and `wait_queue_unlink_one` are operations on subordinate queues, which are not exported outside the scheduler.

The function `wait_queue_member` determines whether a subordinate queue is a member of a queue.

The functions `wait_queue_link` and `wait_queue_unlink` link and unlink a given subordinate queue from its parent queue, respectively.

The function `wait_queue_unlink_one` unlinks the first subordinate queue in a given parent and returns it.

The function `wait_queue_assert_wait` causes the calling thread to wait on the wait queue until it is either interrupted (by a thread timer, for example) or explicitly awakened by another thread. The `interruptible` flag indicates whether this function should allow an asynchronous event to interrupt waiting.

The function `wait_queue_wakeup_all` wakes up all threads waiting on a given queue for a particular event.

The function `wait_queue_peek_locked` returns the first thread from a given wait queue that is waiting on a given event. It does not remove the thread from the queue, nor does it wake the thread. It also returns the wait queue where the thread was found. If the thread is found in a subordinate queue, other subordinate queues are unlocked, as is the parent queue. Only the queue where the thread was found remains locked.

The function `wait_queue_pull_thread_locked` pulls a thread from the wait queue and optionally unlocks the queue. This is generally used with the result of a previous call to `wait_queue_peek_locked`.

The function `wait_queue_wakeup_identity_locked` wakes up the first thread that is waiting for a given event on a given wait queue and starts it running but leaves the thread locked. It then returns a pointer to the thread. This can be used to wake the first thread in a queue and then modify unrelated structures based on which thread was actually awakened before allowing the thread to execute.

The function `wait_queue_wakeup_one` wakes up the first thread that is waiting for a given event on a given wait queue.

The function `wait_queue_wakeup_thread` wakes up a given thread if and only if it is waiting on the specified event and wait queue (or one of its subordinates).

The function `wait_queue_remove` wakes a given thread without regard to the wait queue or event on which it is waiting.

Bootstrap Contexts

In Mac OS X kernel programming, the term context has several meanings that appear similar on the surface, but differ subtly.

First, the term context can refer to a BSD process or Mach task. Switching from one process to another is often referred to as a context switch.

Second, context can refer to the part of the operating system in which your code resides. Examples of this include thread contexts, the interrupt context, the kernel context, an application's context, a Carbon File Manager context, and so on. Even for this use of the term, the exact meaning depends, ironically, on the context in which the term is used.

Finally, context can refer to a **bootstrap context**. In Mach, the bootstrap task is assigned responsibility for looking up requests for Mach ports. As part of this effort, each Mach task is registered in one of two groups—either in the startup context or a user's login context. (In theory, Mach can support any number of independent contexts, however the use of additional contexts is beyond the scope of this document.)

For the purposes of this chapter, the term context refers to a bootstrap context.

When Mac OS X first boots, there is only the top-level context, which is generally referred to as the startup context. All other contexts are subsets of this context. Basic system services that rely on Mach ports must be started in this context in order to work properly.

When a user logs in, the bootstrap task creates a new context called the login context. Programs run by the user are started in the login context. This allows the user to run a program that provides an alternate port lookup mechanism if desired, causing that user's tasks to get a different port when the tasks look up a basic

service. This has the effect of replacing that service with a user-defined version in a way that changes what the user's tasks see, but does not affect any of the rest of the system.

To avoid wasting memory, currently the login context is destroyed when the user logs out (or shortly thereafter). This behavior may change in the future, however. In the current implementation, programs started by the user will no longer be able to look up Mach ports after logout. If a program does not need to do any port lookup, it will not be affected. Other programs will terminate, hang, or behave erratically.

For example, in Mac OS 10.1 and earlier, `sshd` continues to function when started from a user context. However, since it is unable to communicate with `lookupd` or `netinfo`, it stops accepting passwords. This is not a particularly useful behavior.

Other programs such as `esound`, however, continue to work correctly after logout when started from a user context. Other programs behave correctly in their default configuration but fail in other configurations—for example, when authentication support is enabled.

There are no hard and fast rules for which programs will continue to operate after their bootstrap context is destroyed. Only thorough testing can tell you whether any given program will misbehave if started from a user context, since even programs that do not appear to directly use Mach communication may still do so indirectly.

In Mac OS X 10.2, a great deal of effort has gone into making sure that programs that use only standard BSD services and functions do not use Mach lookups in a way that would fail if started from a user context. If you find an application that breaks when started from a Terminal.app window, please file a bug report.

How Contexts Affect Users

From the perspective of a user, contexts are generally unimportant as long as they do not want a program to survive past the end of their login session.

Contexts do become a problem for the administrator, however. For example, if the administrator upgrades `sshd` by killing the old version, starting the new one, and logging out, strange things could happen since the context in which `sshd` was running no longer exists.

Contexts also pose an issue for users running background jobs with `nohup` or users detaching terminal sessions using `screen`. There are times when it is perfectly reasonable for a program to survive past logout, but by default, this does not occur.

There are three basic ways that a user can get around this. In the case of daemons, they can modify the startup scripts to start the application. On restart, the application will be started in the startup context. This is not very practical if the computer in question is in heavy use, however. Fortunately, there are other ways to start services in a startup context.

The second way to run a service in the startup context is to use `ssh` to connect to the computer. Since `sshd` is running in the startup context, programs started from an `ssh` session also register themselves in the startup context. (Note that a user can safely kill the main `sshd` process without being logged out. The user just needs to be careful to kill the right one.)

The third way is to log in as the console user (`>console`), which causes `LoginWindow` to exit and causes `init` to spawn a `getty` process on the console. Since `init` spawns `getty`, which spawns `login`, which spawns the user's shell, any programs started from the text console will be in the startup context.

More generally, any process that is the child of a process in the startup context (other than those inherited by `init` because their parent process exited) is automatically in the startup context. Any process that is the child of a process in the login context is, itself, in the login context. This means that daemons can safely fork children at any time and those children will be in the startup context, as will programs started from the console (not the Console application). This also means that any program started by a user in a terminal window, from Finder, from the Dock, and so on, will be in the currently logged in user's login context, even if that user runs the application using `su` or `sudo`.

How Contexts Affect Developers

If you are writing *only* kernel code, contexts are largely irrelevant (unless you are creating a new context, of course). However, kernel developers frequently need to write a program that registers itself in the startup context in order to provide some

level of driver communication. For example, you could write a user-space daemon that brokers configuration information for a sound driver based on which user is logged in at the time.

In the most general case, the problem of starting an application in the startup context can be solved by creating a startup script for your daemon, which causes it to be run in the startup context after the next reboot. However, users generally do not appreciate having to reboot their computers to install a new driver. Asking the user to connect to his or her own computer with ssh to execute a script is probably not reasonable, either.

The biggest problem with forcing a reboot, of course, is that users often install several programs at once. Rebooting between each install inconveniences the end user, and has no other benefit. For that reason, you should not force the user to restart. Instead, you should offer the user the option, noting that the software may not work correctly until the user restarts. While this does not solve the fundamental problem, it does at least minimize the most common source of complaints.

There are a number of ways to force a program to start in the startup context without rebooting or using ssh. However, these are not robust solutions, and are not recommended. A standard API for starting daemons is under consideration. When an official API becomes available, this chapter will be updated to discuss it.

CHAPTER 10

I/O Kit Overview

Those of you who are already familiar with writing device drivers for Mac OS 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 that supports many categories of devices. This framework is called the **I/O Kit**.

From a programming perspective, the I/O Kit provides an abstract view of the system hardware to the upper layers of Mac OS X. The I/O Kit uses an object-oriented programming model, implemented in a restricted subset of C++ to promote increased code reuse.

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, you need only to fill in the specific code that makes your driver different. For example, all SCSI controllers deliver a fairly standard set of commands to a device, but do so via different low-level mechanisms. By properly using object-oriented programming methodology, a SCSI driver can implement those low-level transport portions without reimplementing the higher level SCSI protocol code. Similar opportunities for code reuse can be found in most types of drivers.

Part of the philosophy of the I/O Kit is to make the design completely open. Rather than hiding parts of the API 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, you 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 9 driver model nor the FreeBSD model offered a feature set rich enough 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 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 (or substantially less pervasive) in previous versions of the Mac OS.

Although FreeBSD supports these features, the BSD driver 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 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 reuse code. In 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 basic developer tools, and code is shared among its various parts.

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 "For More Information" (page 93) 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 that may impact a higher-level view of the device. 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 88)), 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. You can replace functionality and modify the classes yourself if you need to do so.

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 who 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

A **family** defines a collection of high-level abstractions common to all devices of a particular category that takes the form of C code and C++ classes. Families may include headers, libraries, sample code, test harnesses, and documentation. They provide the API, 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 you 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 90) for an illustration.

Drivers

A **driver** is an I/O Kit object that manages a specific device or bus, presenting a more abstract view of that device to other parts of the system. 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 "Connection Example" (page 90) 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**.

Most drivers are in a client-provider relationship, wherein the driver must know about both the family from which it inherits and the family to which it connects. 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 point of connection for a driver. It represents a controllable entity such as a disk or a bus.

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.

The concept of nubs can be more easily visualized by imagining a TV set. There is a wire attached to your wall that provides TV service from somewhere. For all practical purposes, it is permanently associated with that provider, the instantiating class (the cable company who installed the line). It can be attached to the TV to provide a service (cable TV). That wire is a nub.

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. (In this example, imagine one wire for every home serviced by the cable company.)

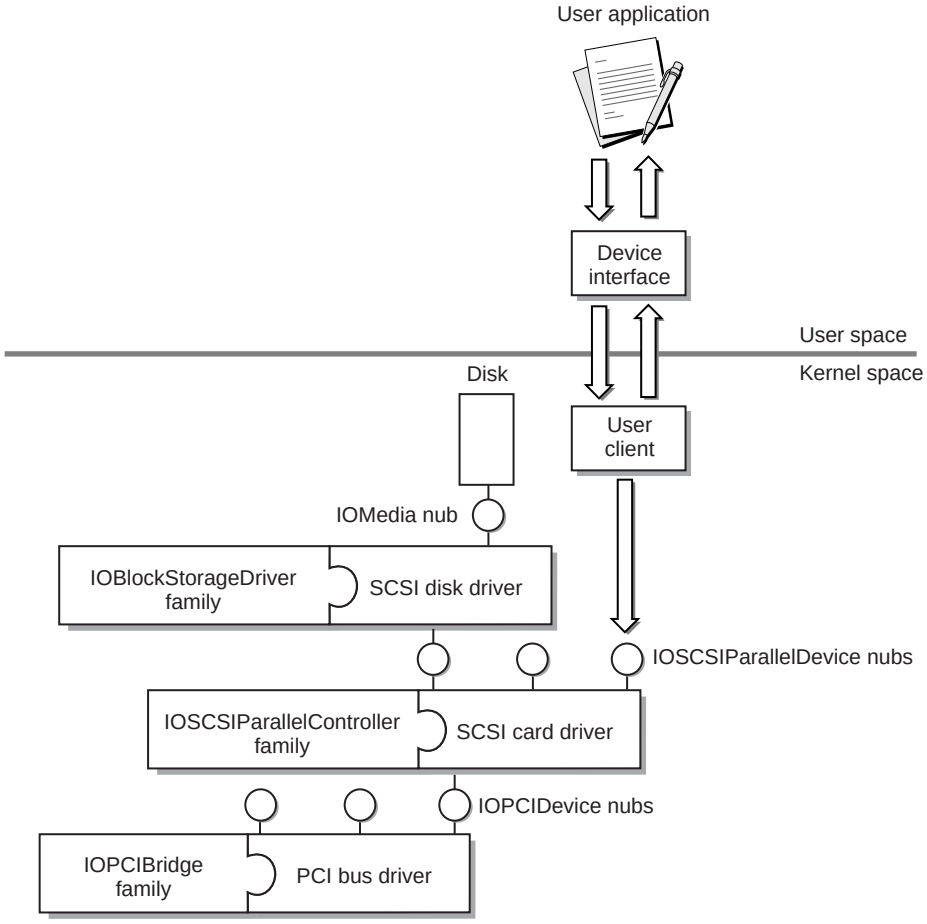
It is also possible for a driver that controls only a single device or service to act as its own nub. (Imagine the antenna on the back of your TV that has a built-in wire.) See the "Connection Example" (page 90) for an illustration of the relationship between nubs and drivers.

Connection Example

Figure 10-1 illustrates the I/O Kit architecture, using several example drivers and their corresponding nubs. Note that many different driver combinations are possible; this diagram shows only one possibility.

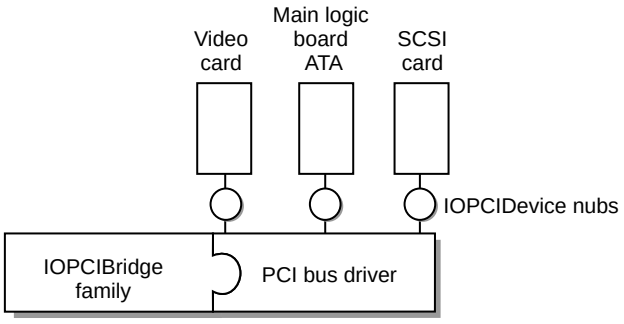
In this case, a SCSI stack is shown, with a PCI controller, a disk, and a SCSI scanner. The SCSI disk is controlled by a kernel-resident driver. The SCSI scanner is controlled by a driver that is part of a user application.

Figure 10-1 I/O Kit architecture

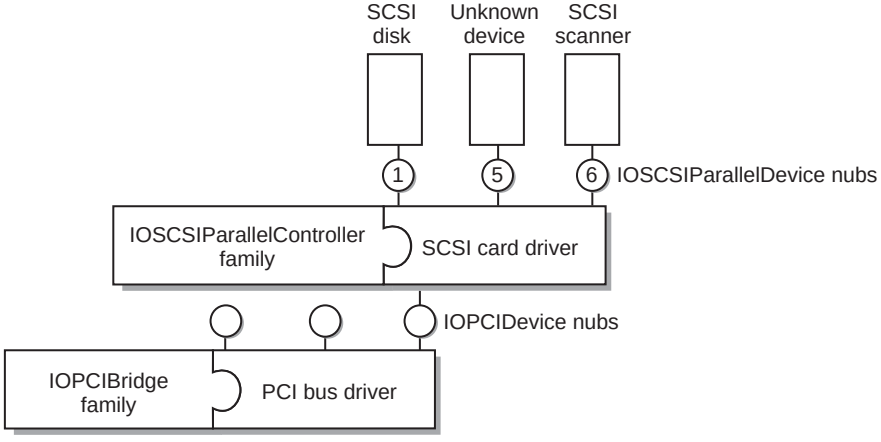


This example illustrates how a SCSI disk driver (Storage family) is connected to the PCI bus. The connection is made in several steps.

1. The PCI bus driver discovers a PCI device and announces its presence by creating a nub (`IOPCIDevice`). The nub's class is defined by the PCI family.

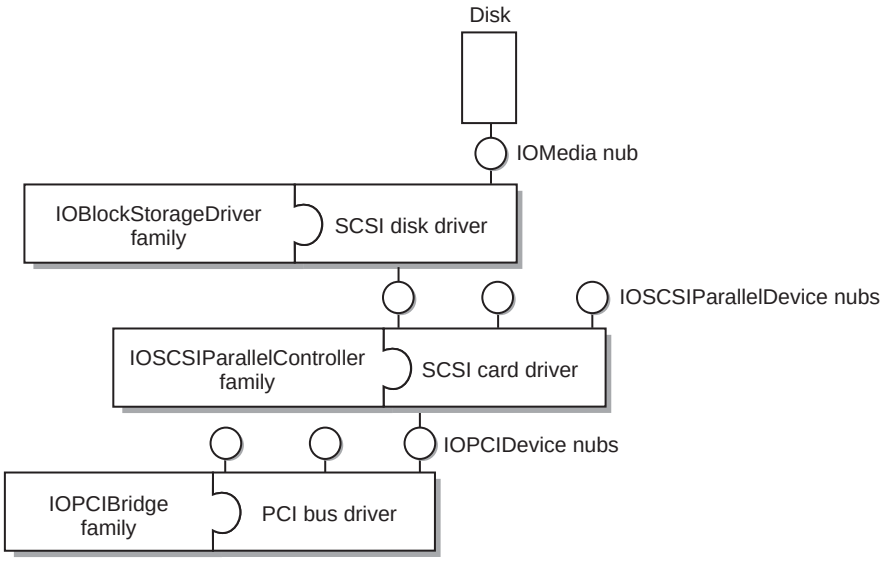


2. The bus driver identifies (matches) the correct device driver and requests that the driver be loaded. At the end of this matching process, a SCSI controller driver has been found and loaded. Loading the controller driver causes all required families to be loaded as well. In this case, the SCSI family 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 controller driver 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.



For More Information

For more information on the I/O Kit, you should read the document *Inside Mac OS X: I/O Kit Fundamentals*, available from Apple's technical publications website, http://developer.apple.com/techpubs. It provides a good general overview of the I/O Kit.

In addition to *Inside Mac OS X: I/O Kit Fundamentals*, the website contains a number of HOWTO documents and topic-specific documents that describe issues specific to particular technology areas such as FireWire and USB.

BSD Overview

The BSD portion of the Mac OS X kernel is derived primarily from FreeBSD, a version of 4.4BSD that offers advanced networking, performance, security, and compatibility features. BSD variants in general are derived (sometimes indirectly) from 4.4BSD-Lite Release 2 from the Computer Systems Research Group (CSRG) at the University of California at Berkeley. BSD provides many advanced features, including the following:

- 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, and NFS. 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 computers 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 and system facilities implemented by 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
- POSIX synchronization primitives
- POSIX shared memory
- signals
  - signal types
  - signal handlers
  - sending signals
- timing and statistics
  - real time
  - interval time
- descriptors
  - files
  - pipes
  - sockets
- 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 is deprecated. Its use is not recommended 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()` function. (Graphic device support and other subsystems provide similar functionality, but using different APIs.) In Mac OS X, this interface should be done through user clients. See the Apple I/O Kit documents for additional information.
- 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.

- Mach provides IPC primitives that differ from the System V primitives traditionally found in UNIX. See "Boundary Crossings" (page 107) for more information on Mach IPC. Some System V primitives are emulated on top of Mach primitives, but their use is discouraged.
- The `dlopen`/`dlsym` API for loading shared code dynamically is not present. Mac OS X provides different mechanisms for supporting shared libraries. For more information, see the man pages for `dyld` and `libtool` and *Inside Mac OS X: UNIX Porting Guide*.
- Several changes have been made to the BSD security model to support single-user and multiple-administrator configurations, including the ability to disable ownership and permissions on a volume-by-volume basis.

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
- enhancements to file-system support
  - implementation of Apple extensions for ISO-9660 file systems
  - multithreaded asynchronous I/O for NFS
  - addition of system calls to support semantics of Mac OS Extended (HFS+) 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.

CHAPTER 11

BSD Overview

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.

File Systems Overview

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), ISO 9660 (used for CD-ROM), MS-DOS, SMB (Windows file sharing standard), AFP (Mac OS file sharing), and UDF.

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.

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.

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).

Mac OS X currently can boot and "root" from an HFS+, UFS, ISO, NFS, or UDF volume. That is, Mac OS X can boot from and mount a volume of any of these types and use it as the primary, or root, file system.

Other file systems can also be mounted, allowing users to gain access to additional volume formats and features.

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.

VFS Transition

The details of the VFS subsystem in Mac OS X are in the process of changing in order to make the VFS interface sustainable.

If you are writing a leaf file system, these changes will still affect you in many ways. please contact Apple Developer Support for more information.

Network Architecture

Network kernel extensions (NKEs) are a specific type of 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 document.

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 "4.4 BSD network architecture" (page 104) 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.

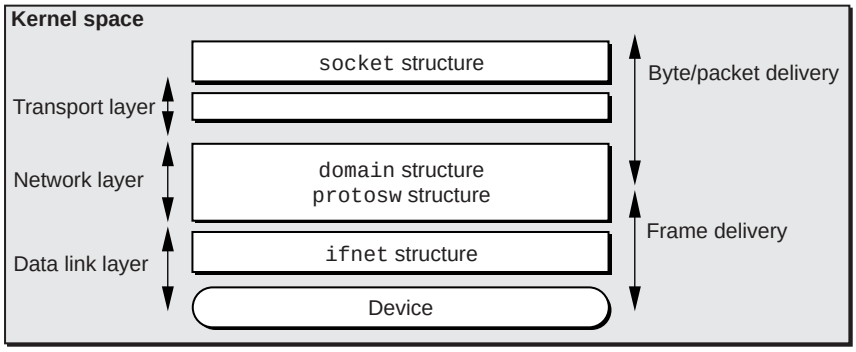
Review of 4.4BSD Network Architecture

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 throughout the 4.4BSD networking infrastructure. Instead, each structure is used at a specific level, as shown in Figure 13-1.

Figure 13-1 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 their destination or source device. 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. These include:

- 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. These are primarily useful as a basis for constructing more complex NKEs such as those used by the Classic environment's networking (`SharedIP`).
- 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 encryption or address translation).

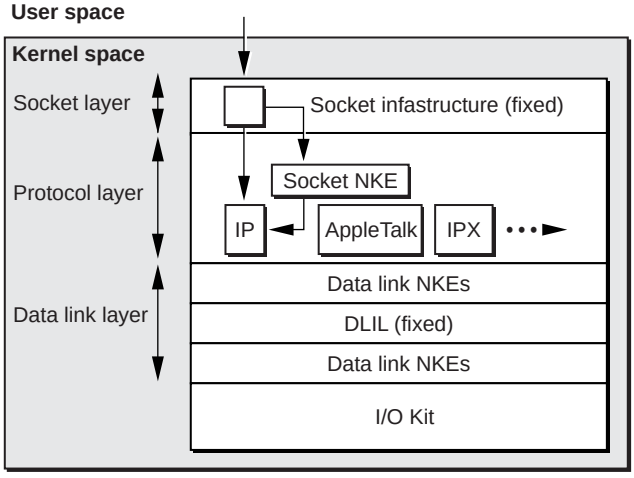
Socket NKEs operate in one of two modes: programmatic or global. Data link NKEs operate only in global mode.

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 **namespace** 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.

Figure 13-2 shows the basic architecture of NKEs.

Figure 13-2 NKE architecture



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 domain structure field `protosw` is now a linked list, thereby removing the array's upper bound. The new `max_protohdr` field defines the maximum protocol header size for the domain. The new `dom_refs` field 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.
- Because the `protosw` structure is no longer an array, the `pr_next` field 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` field is an unsigned integer instead of a short. NKE hooks have been added to link NKE descriptors together.

Boundary Crossings

Two applications can communicate in a number of ways—for example, by using pipes or sockets. The applications themselves are unaware of the underlying mechanisms that provide this communication. However this communication occurs by sending data from one program into the kernel, which then sends the data to the second program.

As a kernel programmer, it is your job to create the underlying mechanisms responsible for communication between your kernel code and applications. This communication is known as crossing the user-kernel boundary. This chapter explains various ways of crossing that boundary.

In a protected memory environment, each process is given its own address space. This means that no program can modify another program's data unless that data also resides in its own memory space (shared memory). The same applies to the kernel. It resides in its own address space. When a program communicates with the kernel, data cannot simply be passed from one address space to the other as you might between threads (or between programs in environments like Mac OS 9 and most real-time operating systems, which do not have protected memory).

We refer to the kernel's address space as **kernel space**, and collectively refer to applications' address spaces as **user space**. For this reason, applications are also commonly referred to as **user-space programs**, or **user programs** for short.

When the kernel needs a small amount of data from an application, the kernel cannot just dereference a pointer passed in from that application, since that pointer is relative to the application's address space. Instead, the kernel generally copies that information into storage within its own address space. When a large region of data needs to be moved, it may map entire pages into kernel space for efficiency. The same behavior can be seen in reverse when moving data from the kernel to an application.

Because it is difficult to move data back and forth between the kernel and an application, this separation is called a **boundary**. It is inherently time consuming to copy data, even if that data is just the user-space address of a shared region. Thus, there is a performance penalty whenever a data exchange occurs. If this penalty is a serious problem, it may affect which method you choose for crossing the user-kernel boundary. Also, by trying to minimize the number of boundary crossings, you may find ways to improve the overall design of your code. This is particularly significant if your code is involved in communication between two applications, since the user-kernel boundary must be crossed twice in that case.

There are a number of ways to cross the user-kernel boundary. Some of them are covered in this chapter in the following sections:

- "Mach Messaging and Mach Interprocess Communication (IPC)" (page 111)
- "BSD syscall API" (page 116)
- "BSD ioctl API" (page 117)
- "BSD sysctl API" (page 118)
- "Memory Mapping and Block Copying" (page 126)

In addition, the I/O Kit uses the user-client/device-interface API for most communication. Because that API is specific to the I/O Kit, it is not covered in this chapter. The user client API is covered in *Inside Mac OS X: I/O Kit Fundamentals*, *Inside Mac OS X: Accessing Hardware From Applications*, *Inside Mac OS X: Making Hardware Accessible to Applications*, and *Inside Mac OS X: Writing I/O Kit Drivers*.

The `ioctl` API is also specific to the construction of device drivers, and is largely beyond the scope of this document. However, since `ioctl` is a BSD API, it is covered at a glance for your convenience.

This chapter covers one subset of Mach IPC—the Mach remote procedure call (RPC) API. It also covers the `syscall`, `sysctl`, memory mapping, and block copying APIs.

Security Considerations

Crossing the user-kernel boundary represents a security risk if the kernel code operates on the data in any substantial way (beyond writing it to disk or passing it to another application). You must carefully perform bounds checking on any data passed in, and you must also make sure your code does not dereference memory that no longer belongs to the client application. Also, under *no circumstances* should you run unverified program code passed in from user space within the kernel. See "Security Considerations" (page 9) for further information.

Choosing a Boundary Crossing Method

The first step in setting up user-kernel data exchange is choosing a means to do that exchange. First, you must consider the purpose for the communication. Some crucial factors are latency, bandwidth, and the kernel subsystem involved. Before choosing a method of communication, however, you should first understand at a high-level each of these forms of communication.

Mach messaging and Mach interprocess communication (IPC) are relatively low-level ways of communicating between two Mach tasks (processes), as well as between a Mach task and the kernel. These form the basis for most communication outside of BSD and the I/O Kit. The Mach remote procedure call (RPC) API is a high level procedural abstraction built on top of Mach IPC. Mach RPC is the most common use of IPC.

The BSD `syscall` API is an API for calling kernel functions from user space. It is used extensively when writing file systems and networking protocols, in ways that are very subsystem-dependent. Developers are strongly discouraged from using the `syscall` API outside of file-system and network extensions, as no plug-in API exists for registering a new system call with the `syscall` mechanism.

The BSD `sysctl` API (in its revised form) supersedes the `syscall` API and also provides a relatively painless way to change individual kernel variables from user space. It has a straightforward plug-in architecture, making it a good choice where possible.

Memory mapping and block copying are used in conjunction with one of the other APIs mentioned, and provide ways of moving large amounts of data (more than a few bytes) or variably sized data to and from kernel space.

Kernel Subsystems

The choice of boundary crossing methods depends largely on the part of the kernel into which you are adding code. In particular, the boundary crossing method preferred for the I/O Kit is different from that preferred for BSD, which is different from that preferred for Mach.

If you are writing a device driver or other related code, you are probably dealing with the I/O Kit. In that case, you should instead read appropriate sections in *Inside Mac OS X: I/O Kit Fundamentals*, *Inside Mac OS X: Accessing Hardware From Applications*, *Inside Mac OS X: Making Hardware Accessible to Applications*, and *Inside Mac OS X: Writing I/O Kit Drivers*.

If you are writing code that resides in the BSD subsystem (for example, a file system), you should generally use BSD APIs such as `syscall` or `sysctl` unless you require high bandwidth or exceptionally low latency.

If you are writing code that resides anywhere else, you will probably have to use Mach messaging.

Bandwidth and Latency

The guidelines in the previous section apply to most communication between applications and kernel code. The methods mentioned, however, are somewhat lacking where high bandwidth or low latency are concerns.

If you require high bandwidth, but latency is not an issue, you should probably consider doing memory-mapped communication. For large messages this is handled somewhat transparently by Mach RPC, making it a reasonable choice. For

BSD, however, you must explicitly pass pointers and use `copyin` and `copyout` to move large quantities of data. This is discussed in more detail in "Memory Mapping and Block Copying" (page 126).

If you require low latency but bandwidth is not an issue, `sysctl` and `syscall` are not good choices. Mach RPC, however, may be an acceptable solution. Another possibility is to actually wire a page of memory (see "Memory Mapping and Block Copying" (page 126) for details), start an asynchronous Mach RPC `simpleroutine` (to process the data), and use either locks or high/low water marks (buffer fullness) to determine when to read and write data. This can work for high-bandwidth communication as well.

If you require both high bandwidth and low latency, you should also look at the user client/device interface model used in the I/O Kit, since that model has similar requirements.

Mach Messaging and Mach Interprocess Communication (IPC)

Mach IPC and Mach messaging are the basis for much of the communication in Mac OS X. In many cases, however, these facilities are used indirectly by services implemented on top of one of them. Mach messaging and IPC are fundamentally similar except that Mach messaging is stateless, which prevents certain types of error recovery, as explained later. Except where explicitly stated, this section treats the two as equivalent.

The fundamental unit of Mach IPC is the **port**. The concept of Mach ports can be difficult to explain in isolation, so instead this section assumes a passing knowledge of a similar concept, that of ports in TCP/IP.

In TCP/IP, a server listens for incoming connections over a network on a particular port. Multiple clients can connect to the port and send and receive data in word-sized or multiple-word–sized blocks. However, only one server process can be bound to the port at a time.

In Mach IPC, the concept is the same, but the players are different. Instead of multiple hosts connecting to a TCP/IP port, you have multiple Mach tasks on the same computer connecting to a Mach port. Instead of firewall rules on a port, you have **port rights** that specify what tasks can send data to a particular Mach port.

Also, TCP/IP ports are bidirectional, while Mach ports are unidirectional, much like UNIX pipes. This means that when a Mach task connects to a port, it generally allocates a reply port and sends a message containing send rights to that reply port so that the receiving task can send messages back to the sending task.

As with TCP/IP, multiple client tasks can open connections to a Mach port, but only one task can be listening on that port at a time. Unlike TCP/IP, however, the IPC mechanism itself provides an easy means for one task to hand off the right to listen to an arbitrary task. The term **receive rights** refers to a task's ability to listen on a given port. Receive rights can be sent from task to task in a Mach message. In the case of Mach IPC (but *not* Mach messaging), receive rights can even be configured to automatically return to the original task if the new task crashes or becomes unreachable (for example, if the new task is running on another computer and a router crashes).

In addition to specifying receive rights, Mach ports can specify which tasks have the right to send data. A task with send rights may be able to send once, or may be able to arbitrarily send data to a given port, depending on the nature of the rights.

Using Well-Defined Ports

Before you can use Mach IPC for task communication, the sending task must be able to obtain send rights on the receiving task's **task port**. Historically, there are several ways of doing this, not all of which are supported by Mac OS X. For example, in Mac OS X, unlike most other Mach derivatives, there is no service server or name server. Instead, the bootstrap task and **mach_init** subsume this functionality.

When a task is created, it is given send rights to a bootstrap port for sending messages to the bootstrap task. Normally a task would use this port to send a message that gives the bootstrap task send rights on another port so that the bootstrap task can then return data to the calling task. Various routines exist in `bootstrap.h` that abstract this process. Indeed, most users of Mach IPC or Mach messaging actually use Mach remote procedure calls (RPC), which are implemented on top of Mach IPC.

Since direct use of IPC is rarely desirable (because it is not easy to do correctly), and because the underlying IPC implementation has historically changed on a regular basis, the details are not covered here. You can find more information on using Mach IPC directly in the *Mach 3 Server Writer's Guide* from Silicomp (formerly the Open Group, formerly the Open Software Foundation Research Institute), which can be obtained from the developer section of Apple's website. While much of the information contained in that book is not fully up-to-date with respect to Mac OS X, it should still be a relatively good resource on using Mach IPC.

Remote Procedure Calls (RPC)

Mach RPC is the most common use for Mach IPC. It is frequently used for user-kernel communication, but can also be used for task to task or even computer-to-computer communication. Programmers frequently use Mach RPC for setting certain kernel parameters such as a given thread's scheduling policy.

RPC is convenient because it is relatively transparent to the programmer. Instead of writing long, complex functions that handle ports directly, you have only to write the function to be called and a small **RPC definition** to describe how to export the function as an RPC interface. After that, any application with appropriate permissions can call those functions as if they were local functions, and the compiler will convert them to RPC calls.

In the directory `osfmk/mach` (relative to your checkout of the xnu module from CVS), there are a number of files ending in `.defs`; these files contain the RPC definitions. When the kernel (or a kernel module) is compiled, the **Mach Interface Generator (MIG)** uses these definitions to create IPC code to support the functions exported via RPC. Normally, if you want to add a new remote procedure call, you should do so by adding a definition to one of these existing files. (See "Building and Debugging Kernels" (page 153) for more information on obtaining kernel sources.)

What follows is an example of the definition for a **routine**, one of the more common uses of RPC.

```
routine thread_policy_get(
                    thread      : thread_act_t;
                    flavor      : thread_policy_flavor_t;
            out     policy_info : thread_policy_t, CountInOut;
            inout   get_default : boolean_t);
```

Notice the C-like syntax of the definition. Each parameter in the routine roughly maps onto a parameter in the C function. The C prototype for this function follows.

```
kern_return_t thread_policy_get(
    thread_act_t             act,
    thread_policy_flavor_t   flavor,
    thread_policy_t          policy_info,
    mach_msg_type_number_t   *count,
    boolean_t                get_default);
```

The first two parameters are integers, and are passed as call-by-value. The third is a `struct` containing integers. It is an outgoing parameter, which means that the values stored in that variable will *not* be received by the function, but *will* be overwritten on return.

Note: The parameters are all word-sized or multiples of the word size. Smaller data are impossible because of limitations inherent to the underlying Mach IPC mechanisms.

From there it becomes more interesting. The fourth parameter in the C prototype is a representation of the size of the third. In the definition file, this is represented by an added option, `CountInOut`.

The MIG option `CountInOut` specifies that there is to be an `inout` parameter called `count`. An `inout` parameter is one in which the original value can be read by the function being called, and its value is replaced on return from that function. Unlike a separate `inout` parameter, however, the value initially passed through this parameter is not directly set by the calling function. Instead, it is tied to the `policy_info` parameter so that the number of integers in `policy_info` is transparently passed in through this parameter.

In the function itself, the function checks the count parameter to verify that the buffer size is at least the size of the data to be returned to prevent exceeding array bounds. The function changes the value stored in count to be the desired size and returns an error if the buffer is not large enough (unless the buffer pointer is null, in

which case it returns success). Otherwise, it dereferences the various fields of the `policy_info` parameter and in so doing, stores appropriate values into it, then returns.

Note: Since Mach RPC is done via message passing, `inout` parameters are technically call-by-value-return and *not* call-by-reference. For more realistic call-by-reference, you need to pass a pointer. The distinction is not particularly significant except when aliasing occurs. (Aliasing means having a single variable visible in the same scope under two or more different names.)

In addition to the `routine`, Mach RPC also has a **simpleroutine**. A `simpleroutine` is a routine that is, by definition, asynchronous. It can have no `out` or `inout` parameters and no return value. The caller does not wait for the function to return. One possible use for this might be to tell an I/O device to send data as soon as it is ready. In that use, the `simpleroutine` might simply wait for data, then send a message to the calling task to indicate the availability of data.

Another important feature of MIG is that of the **subsystem**. In MIG, a subsystem is a group of `routines` and `simpleroutines` that are related in some way. For example, the semaphore subsystem contains related routines that operate on semaphores. There are also subsystems for various timers, parts of the virtual memory (VM) system, and dozens of others in various places throughout the kernel.

Most of the time, if you need to use RPC, you will be doing it within an existing subsystem. The details of creating a new subsystem are beyond the scope of this document. Developers needing to add a new Mach subsystem should consult the *Mach 3 Server Writer's Guide* from The Open Group (TOG), which can be obtained from various locations on the internet.

Another feature of MIG is the type. A type in MIG is exactly the same thing as it is in programming languages. However, the construction of aggregate types differs somewhat.

```
type clock_flavor_t      = int;
type clock_attr_t        = array[*:1] of int;
type mach_timespec_t     = struct[2] of int;
```

Data of type `array` is passed as the user-space address of what is assumed to be a contiguous array of the base type, while a struct is passed by copying all of the individual values of an array of the base type. Otherwise, these are treated similarly. A "struct" is not like a C struct, as elements of a MIG struct must all be of the same base type.

The declaration syntax is similar to Pascal, where `*:1` and `2` represent sizes for the array or structure, respectively. The `*:1` construct indicates a variable-sized array, where the size can be up to 1, inclusive, but no larger.

Calling RPC From User Applications

RPC, as mentioned previously, is virtually transparent to the client. The procedure call looks like any other C function call, and no additional library linkage is needed. You need only to bring the appropriate headers in with a `#include` directive. The compiler automatically recognizes the call as a remote procedure call and handles the underlying MIG aspects for you.

BSD `syscall` API

The `syscall` API is the traditional UNIX way of calling kernel functions from user space. Its implementation varies from one part of the kernel to the next, however, and it is completely unsupported for loadable modules. For this reason, it is not a recommended way of getting data into or out of the kernel in Mac OS X unless you are writing a file system.

File systems have to support a number of standard system calls (for example, `mount`), but do so by means of generic file system routines that call the appropriate file-system functions. Thus, if you are writing a file system, you need to implement those functions, but you do not need to write the code that handles the system calls directly. For more information on implementing `syscall` support in file systems, see the chapter "File Systems Overview" (page 100).

BSD ioctl API

The ioctl interface provides a way for an application to send certain commands or information to a device driver. These can be used for parameter tuning (though this is more commonly done with sysctl), but can also be used for sending instructions for the driver to perform a particular task (for example, rewinding a tape drive).

The use of the ioctl interface is essentially the same under Mac OS X as it is in other BSD-derived operating systems, except in the way that device drivers register themselves with the system. In Mac OS X, unlike most BSDs, the contents of the /dev directory are created dynamically by the kernel. This file system mounted on /dev is referred to as **devfs**. You can, of course, still manually create device nodes with mknod, because devfs is union mounted over the root file system.

The I/O Kit automatically registers some types of devices with devfs, creating a node in /dev. If your device family does not do that, you can manually register yourself in devfs using cdevsw_add or bdevsw_add (for character and block devices, respectively).

When registering a device manually with devfs, you create a struct cdevsw or struct bdevsw yourself. In that device structure, one of the function pointers is to an ioctl function. You must define the particular values passed to the ioctl function in a header file accessible to the person compiling the application.

A user application can also look up the device using the I/O Kit function call IOServiceGetMatchingServices and then use various I/O Kit calls to tune parameter instead. For more information on looking up a device driver from an application, see the document *Inside Mac OS X: Accessing Hardware From Applications* and *Inside Mac OS X: Making Hardware Accessible to Applications.*

You can also find additional information about writing an ioctl in *The Design and Implementation of the 4.4 BSD Operating System*. See the bibliography at the end of this document for more information.

BSD sysctl API

The sysctl API is specifically designed for kernel parameter tuning. This functionality supersedes the syscall API, and also provides an easy way to tune simple kernel parameters without actually needing to write a handler routine in the kernel. The sysctl namespace is divided into several broad categories corresponding to the purpose of the parameters in it. Some of these areas include

- kern—general kernel parameters
- vm—virtual memory options
- fs—filesystem options
- machdep—machine dependent settings
- net—network stack settings
- debug—debugging settings
- hw—hardware parameters (generally read-only)
- user—parameters affecting user programs
- ddb—kernel debugger

Most of the time, programs use the sysctl call to retrieve the current value of a kernel parameter. For example, in Mac OS X, the hw sysctl group includes the option ncpu, which returns the number of processors in the current computer (or the maximum number of processors supported by the kernel on that particular computer, whichever is less).

The sysctl API can also be used to modify parameters (though most parameters can only be changed by the root). For example, in the net hierarchy, net.inet.ip.forwarding can be set to 1 or 0, to indicate whether the computer should forward packets between multiple interfaces (basic routing).

General Information on Adding a sysctl

When adding a sysctl, you must do all of the following *first*:

- add the following includes:

```
#include <mach/mach_types.h>

#include <sys/systm.h>

#include <sys/types.h>

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

- add `-no-cpp-precomp` to your compiler options in Project Builder (or to `CFLAGS` in your makefile if building by hand).

Adding a `sysctl` Procedure Call

Adding a `sysctl` was once a daunting task requiring changes to dozens of files. With the current implementation, a `sysctl` can be added simply by writing the appropriate handler functions and then registering the handler with the system at runtime. The old-style `sysctl`, which used fixed numbers for each `sysctl`, is deprecated.

Note: Because this is largely a construct of the BSD subsystem, all path names in this section can be assumed to be from /path/to/xnu-version/bsd/.

Also, you may safely assume that all program code snippets should go into the main source file for your subsystem or module unless otherwise noted, and that in the case of modules, function calls should be made from your init or unload routines unless otherwise noted.

The preferred way of adding a `sysctl` looks something like the following:

```
SYSCTL_PROC(_hw, OID_AUTO, l2cr, CTLTYPE_INT|CTLFLAG_RW,
    &L2CR, 0, &sysctl_l2cr, "I", "L2 Cache Register");
```

The `_PROC` part indicates that you are registering a procedure to provide the value (as opposed to simply reading from a static address in kernel memory). `_hw` is the top level category (in this case, hardware), and `OID_AUTO` indicates that you should

be assigned the next available sysctl ID in that category (as opposed to the old-style, fixed ID sysctls). l2cr is the name of your sysctl, which will be used by applications to look up the number of your sysctl.

Note: Not all top level categories will necessarily accept the addition of a user-specified new-style sysctl. If you run into problems, you should try a different top-level category.

CTLTYPE_INT indicates that the value being changed is an integer. Other legal values are CTLTYPE_NODE, CTLTYPE_STRING, CTLTYPE_QUAD, and CTLTYPE_OPAQUE (also known as CTLTYPE_STRUCT). CTLTYPE_NODE is the only one that isn't somewhat obvious. It refers to a node in the sysctl hierarchy that isn't directly usable, but instead is a parent to other entries. Two examples of nodes are hw and kern.

CTLFLAG_RW indicates that the value can be read and written. Other legal values are CTLFLAG_RD, CTLFLAG_WR, CTLFLAG_ANYBODY, and CTLFLAG_SECURE. CTLFLAG_ANYBODY means that the value should be modifiable by anybody. (The default is for variables to be changeable only by root.) CTLFLAG_SECURE means that the variable can be changed only when running at securelevel <= 0 (effectively, in single-user mode).

L2CR is the location where the sysctl will store its data. Since the address is set at compile time, however, this must be a global variable or a static local variable. In this case, L2CR is a global of type unsigned int.

The number 0 is a second argument that is passed to your function. This can be used, for example, to identify which sysctl was used to call your handler function if the same handler function is used for more than one sysctl. In the case of strings, this is used to store the maximum allowable length for incoming values.

sysctl_l2cr is the handler function for this sysctl. The prototype for these functions is of the form

```
static int sysctl_l2cr SYSCTL_HANDLER_ARGS;
```

If the sysctl is writable, the function may either use sysctl_handle_int to obtain the value passed in from user space and store it in the default location or use the SYSCTL_IN macro to store it into an alternate buffer. This function must also use the SYSCTL_OUT macro to return a value to user space.

"I" indicates that the argument should refer to a variable of type integer (or a constant, pointer, or other piece of data of equivalent width), as opposed to "L" for a long, "A" for a string, "N" for a node (a sysctl that is the parent of a sysctl category or subcategory), or "S" for a struct. "L2 Cache Register" is a human-readable description of your sysctl.

In order for a sysctl to be accessible from an application, it must be registered. To do this, you do the following:

```
sysctl_register_oid(&sysctl__hw_l2cr);
```

You should generally do this in an init routine for a loadable module. If your code is not part of a loadable module, you should add your sysctl to the list of built-in OIDs in the file kern/sysctl_init.c.

If you study the SYSCTL_PROC constructor macro, you will notice that sysctl__hw_l2cr is the name of a variable created by that macro. This means that the SYSCTL_PROC line must be before sysctl_register_oid in the file, and must be in the same (or broader) scope. This name is in the form of sysctl_ followed by the name of it's parent node, followed by another underscore (_) followed by the name of your sysctl.

A similar function, sysctl_unregister_oid exists to remove a sysctl from the registry. If you are writing a loadable module, you should be certain to do this when your module is unloaded.

In addition to registering your handler function, you also have to write the function. The following is a typical example

```
static int myhandler SYSCTL_HANDLER_ARGS
{

    int error, retval;

    error = sysctl_handle_int(oidp, oidp->oid_arg1, oidp->oid_arg2, req);
    if (!error && req->newptr) {
        /* We have a new value stored in the standard location.*/
        /* Do with it as you see fit here. */
        printf("sysctl_test: stored %d\n", SCTEST);
    } else if (req->newptr) {
        /* Something was wrong with the write request */
        /* Do something here if you feel like it.... */
```

```
    } else {
        /* Read request. Always return 763, just for grins. */
        printf("sysctl_test: read %d\n", SCTEST);
        retval=763;
        error=SYSCTL_OUT(req, &retval, sizeof retval);
    }
    /* In any case, return success or return the reason for failure */
    return error;
}
```

This demonstrates the use of `SYSCTL_OUT` to send an arbitrary value out to user space from the `sysctl` handler. The "phantom" `req` argument is part of the function prototype when the `SYSCTL_HANDLER_ARGS` macro is expanded, as is the `oidp` variable used elsewhere. The remaining arguments are a pointer (type indifferent) and the length of data to copy (in bytes).

This code sample also introduces a new function, `sysctl_handle_int`, which takes the arguments passed to the `sysctl`, and writes the integer into the usual storage area (`L2CR` in the earlier example, `SCTEST` in this one). If you want to see the new value without storing it (to do a sanity check, for example), you should instead use the `SYSCTL_IN` macro, whose arguments are the same as `SYSCTL_OUT`.

Registering a New Top Level `sysctl`

In addition to adding new `sysctl` options, you can also add a new category or subcategory of `sysctl`. The macro `SYSCTL_DECL` can be used to declare a node that can have children. This requires modifying one additional file to create the child list. For example, if your main C file does this:

```
SYSCTL_DECL(_net_newcat);
SYSCTL_NODE(_net, OID_AUTO, newcat, CTLFLAG_RW, handler, "new category");
```

then this is basically the same thing as declaring `extern sysctl_oid_list sysctl__net_newcat_children` in your program. In order for the kernel to compile, or the module to link, you must then add this line:

```
struct sysctl_oid_list sysctl__net_newcat_children;
```

If you are not writing a module, this should go in the file kern/kern_newsysctl.c. Otherwise, it should go in one of the files of your module. Once you have created this variable, you can use _net_newcat as the parent when creating a new sysctl. As with any sysctl, the node (sysctl__net_newcat) must be registered with sysctl_register_oid and can be unregistered with sysctl_unregister_oid.

Note: When creating a top level sysctl, parent is simply left blank, for example,

```
SYSCTL_NODE( , OID_AUTO, _topname, flags, handler_fn, "desc");
```

Adding a Simple sysctl

If your sysctl only needs to read a value out of a variable, then you do not need to write a function to provide access to that variable using sysctl. Instead, you can use one of the following macros:

- SYSCTL_INT(parent, nbr, name, access, ptr, val, descr)
- SYSCTL_LONG(parent, nbr, name, access, ptr, descr)
- SYSCTL_STRING(parent, nbr, name, access, arg, len, descr)
- SYSCTL_OPAQUE(parent, nbr, name, access, ptr, len, descr)
- SYSCTL_STRUCT(parent, nbr, name, access, arg, type, descr)

The first four parameters for each macro are the same as for SYSCTL_PROC (described in the previous section) as is the last parameter. The len parameter (where applicable) gives a length of the string or opaque object in bytes.

The arg parameters are pointers just like the ptr parameters. However, the parameters named ptr are explicitly described as pointers because you must explicitly use the "address of" (&) operator unless you are already working with a pointer. Parameters called arg either operate on base types that are implicitly pointers or add the & operator in the appropriate place during macro expansion. In both cases, the argument should refer to the integer, character, or other object that the sysctl will use to store the current value.

The type parameter is the name of the type minus the "struct". For example, if you have an object of type struct scsipi, then you would use scsipi as that argument. The SYSCTL_STRUCT macro is functionally equivalent to SYSCTL_OPAQUE, except that it hides the use of sizeof.

Finally, the val parameter for SYSCTL_INT is a default value. If the value passed in ptr is NULL, this value is returned when the sysctl is used. You can use this, for example, when adding a sysctl option that is specific to certain hardware or certain compile options. One possible example of this might be a special value for feature.version that means "not present." If that feature became available (for example, if a module were loaded by some user action), it could then update that pointer. If that module were subsequently unloaded, it could set the pointer back to NULL.

Calling a sysctl From User Space

Unlike RPC, sysctl requires explicit intervention on the part of the programmer. To complicate things further, there are two different ways of calling sysctl functions, and neither one works for every sysctl. The old-style sysctl call can only invoke a sysctl if it is listed in a static OID table in the kernel. The new-style sysctlbyname call will work for any user-added sysctl, but not for those listed in the static table. Occasionally, you will even find a sysctl that is registered in both ways, and thus available to both calls. In order to understand the distinction, you must first consider the functions used.

The sysctlbyname System Call

If you are calling a sysctl that was added using the new sysctl method (including any sysctl that you may have added), then your sysctl does not have a fixed number that identifies it, since it was added dynamically to the system. Since there is no approved way to get this number from user space, and since the underlying implementation is not guaranteed to remain the same in future releases, you cannot call a dynamically added sysctl using the sysctl function. Instead, you must use sysctlbyname.

```
sysctlbyname(char *name, void *oldp, size_t *oldlenp,
        void *newp, u_int newlen)
```

The parameter name is the name of the sysctl, encoded as a standard C string.

The parameter oldp is a pointer to a buffer where the old value will be stored. The oldlenp parameter is a pointer to an integer-sized buffer that holds the current size of the oldp buffer. If the oldp buffer is not large enough to hold the returned data, the call will fail with errno set to ENOMEM, and the value pointed to by oldlenp will be changed to indicate the buffer size needed for a future call to succeed.

Here is an example for reading an integer, in this case a buffer size.

```
int get_debug_bufsize()
{
char *name=”debug.bpf_bufsize”;
int bufsize, retval;
size_t len;
len=4;
retval=sysctlbyname(name, &bufsize, &len, NULL, 0);
/* Check retval here */
return bufsize;
}
```

The `sysctl` System Call

The `sysctlbyname` system call is the recommended way to call system calls. However, not every built-in `sysctl` is registered in the kernel in such a way that it can be called with `sysctlbyname`. For this reason, you should also be aware of the `sysctl` system call.

Note: If you are adding a `sysctl`, it will be accessible using `sysctlbyname`. You should use this system call only if the `sysctl` you need cannot be retrieved using `sysctlbyname`. In particular, you should not assume that future versions of `sysctl` will be backed by traditional numeric OIDs except for the existing legacy OIDs, which will be retained for compatibility reasons.

The `sysctl` system call is part of the original historical BSD implementation of `sysctl`. You should not depend on its use for any `sysctl` that you might add to the system. The usage of `sysctl` looks like the following

```
sysctl(int *name, u_int namelen, void *oldp, size_t *oldlenp,
        void *newp, u_int newlen)
```

`Sysctl`, in this form, is based on the **MIB**, or Management Information Base. A MIB is a list of objects and identifiers for those objects. Each object identifier, or OID, is a list of integers that represent a tokenization of a path through the `sysctl` tree. For example, if the hw class of `sysctl` is number 3, the first integer in the OID would be the number 3. If the `l2cr` option is built into the system and assigned the number 75, then the second integer in the OID would be 75. To put it another way, each number in the OID is an index into a node’s list of children.

Here is a short example of a call to get the bus speed of the current computer:

```
int get_bus_speed()
{
int mib[2], busspeed, retval;
unsigned int miblen;
size_t len;
mib[0]=CTL_HW;
mib[1]=HW_BUS_FREQ;
miblen=2;
len=4;
retval=sysctl(mib, miblen, &busspeed, &len, NULL, 0);
/* Check retval here */
return busspeed;
}
```

For more information on the `sysctl` system call, see the manual page `sysctl(2)`

Memory Mapping and Block Copying

Memory mapping is one of the more common means of communicating between two applications or between an application and the kernel. While occasionally used by itself, it is usually used in conjunction with one of the other means of boundary crossing.

One way of using memory mapping is known as **shared memory**. In this form, one or more pages of memory are mapped into the address space of two processes. Either process can then access or modify the data stored in those shared pages. This is useful when moving large quantities of data between processes, as it allows direct communication without multiple user-kernel boundary crossings. Thus, when moving large amounts of data between processes, this is preferable to traditional message passing.

The same holds true with memory mapping between an application and the kernel. The BSD `sysctl` and `syscall` interfaces (and to an extent, Mach IPC) were designed to transfer small units of data of known size, such as an array of four integers. In this regard, they are much like a traditional C function call. If you need to pass a large

amount of data to a function in C, you should pass a pointer. This is also true when passing data between an application and the kernel, with the addition of memory mapping or copying to allow that pointer to be dereferenced in the kernel.

There are a number of limitations to the way that memory mapping can be used to exchange data between an application and the kernel. For one, memory allocated in the kernel cannot be written to by applications, including those running as root (unless the kernel is running in an insecure mode, such as single user mode). For this reason, if a buffer must be modified by an application, the buffer must be allocated by that program, *not* by the kernel.

When you use memory mapping for passing data to the kernel, the application allocates a block of memory and fills it with data. It then performs a system call that passes the address to the appropriate function in kernel space. It should be noted, however, that the address being passed is a virtual address, not a physical address, and more importantly, it is relative to the address space of the program, which is *not* the same as the address space of the kernel.

Since the address is a user-space virtual address, the kernel must call special functions to copy the block of memory into a kernel buffer or to map the block of memory into the kernel's address space.

In the Mac OS X kernel, data is most easily copied into kernel space with the BSD `copyin` function, and back out to user space with the `copyout` function. For large blocks of data, entire pages will be memory mapped using copy-on-write. For this reason, it is generally not useful to do memory mapping by hand.

Getting data from the kernel *to* an application can be done in a number of ways. The most common method is the reverse of the above, in which the application passes in a buffer pointer, the kernel scribbles on a chunk of data, uses `copyout` to copy the buffer data into the address space of the application, and returns `KERN_SUCCESS`. Note that this is really using the buffer allocated in the application, even though the physical memory may have actutally been allocated by the kernel. Assuming the kernel frees its reference to the buffer, no memory is wasted.

A special case of memory mapping occurs when doing I/O to a device from user space. Since I/O operations can, in some cases, be performed by DMA hardware that operates based on physical addressing, it is vital that the memory associated with I/O buffers not be paged out while the hardware is copying data to or from the buffer.

For this reason, when a driver or other kernel entity needs a buffer for I/O, it must take steps to mark it as not pageable. This step is referred to as **wiring** the pages in memory.

Wiring pages into memory can also be helpful where high bandwidth, low latency communication is desired, as it prevents shared buffers from being paged out to disk. In general, however, this sort of workaround should be unnecessary, and is considered to be bad programming practice.

Pages can be wired in two ways. When a memory region is allocated, it may be allocated in a nonpageable fashion. The details of allocating memory for I/O differ, depending on what part of the kernel you are modifying. This is described in more detail in the appropriate sections of this document, or in the case of the I/O Kit, in the API reference documentation (available from the developer section of Apple's web site). Alternately, individual pages may be wired after allocation.

The recommended way to do this is through a call to `vm_wire` in BSD parts of the kernel, with `mlock` from applications (but only by processes running as root), or with `IOMemoryDescriptor::prepare` in the I/O Kit. Because this can fail for a number of reasons, it is particularly crucial to check return values when wiring memory. The `vm_wire` call and other virtual memory topics are discussed in more detail in "Memory and Virtual Memory" (page 53). The `IOMemoryDescriptor` class is described in more detail in the I/O Kit API reference available from the developer section of Apple's web site.

Summary

Crossing the user-kernel boundary is not a trivial task. Many mechanisms exist for this communication, and each one has specific advantages and disadvantages, depending on the environment and bandwidth requirements. Security is a constant concern to prevent inadvertently allowing one program to access data or files from another program or user. It is every kernel programmer's personal responsibility to take security into account any time that data crosses the user-kernel boundary.

Synchronization Primitives

This chapter is not intended as an introduction to synchronization. It is assumed that you have some understanding of the basic concepts of locks and semaphores already. If you need additional background reading, synchronization is covered in most introductory operating systems texts. However, since synchronization in the kernel is somewhat different from locking in an application this chapter does provide a brief overview to help ease the transition, or for experienced kernel developers, to refresh your memory.

As a Mac OS X kernel programmer, you have many choices of synchronization mechanisms at your disposal. The kernel itself provides two such mechanisms: locks and semaphores.

A **lock** is used for basic protection of shared resources. Multiple threads can attempt to acquire a lock, but only one thread can actually hold it at any given time (at least for traditional locks—more on this later). While that thread holds the lock, the other threads must wait. There are several different types of locks, differing mainly in what threads do while waiting to acquire them.

A **semaphore** is much like a lock, except that a finite number of threads can hold it simultaneously. Semaphores can be thought of as being much like piles of tokens. Multiple threads can take these tokens, but when there are none left, a thread must wait until another thread returns one. It is important to note that semaphores can be implemented in many different ways, so Mach semaphores may not behave in the same way as semaphores on other platforms.

In addition to locks and semaphores, certain low-level synchronization primitives like test and set are also available, along with a number of other atomic operations. These additional operations are described in `libkern/gen/OSAtomicOperations.c` in the kernel sources. Such atomic operations may be helpful if you do not need something as robust as a full-fledged lock or semaphore. Since they are not general synchronization mechanisms, however, they are beyond the scope of this chapter.

Semaphores

Semaphores and locks are similar, except that with semaphores, more than one thread can be doing a given operation at once. Semaphores are commonly used when protecting multiple indistinct resources. For example, you might use a semaphore to prevent a queue from overflowing its bounds.

Mac OS X uses traditional counting semaphores rather than binary semaphores (which are essentially locks). Mach semaphores obey Mesa semantics—that is, when a thread is awakened by a semaphore becoming available, it is not executed immediately. This presents the potential for starvation in multiprocessor situations when the system is under low overall load because other threads could keep downing the semaphore before the just-woken thread gets a chance to run. This is something that you should consider carefully when writing applications with semaphores.

Semaphores can be used any place where mutexes can occur. This precludes their use in interrupt handlers or within the context of the scheduler, and makes it strongly discouraged in the VM system. The public API for semaphores is divided between the MIG–generated `task.h` file (located in your build output directory, included with `#include <mach/task.h>`) and `osfmk/mach/semaphore.h` (included with `#include <mach/semaphore.h>`).

The public semaphore API includes the following functions:

```
kern_return_t semaphore_create(task_t task, semaphore_t *semaphore,
    int policy, int value)
kern_return_t semaphore_signal(semaphore_t semaphore)
kern_return_t semaphore_signal_all(semaphore_t semaphore)
kern_return_t semaphore_wait(semaphore_t semaphore)
kern_return_t semaphore_destroy(task_t task, semaphore_t semaphore)
kern_return_t semaphore_signal_thread(semaphore_t semaphore,
    thread_act_t thread_act)
```

which are described in `<mach/semaphore.h>` or `xnu/osfmk/mach/semaphore.h` (except for create and destroy, which are described in `<mach/task.h>`.

The use of these functions is relatively straightforward with the exception of the `semaphore_create`, `semaphore_destroy`, and `semaphore_signal_thread` calls.

The `value` and `semaphore` parameters for `semaphore_create` are exactly what you would expect—a pointer to the semaphore structure to be filled out and the initial value for the semaphore, respectively.

The `task` parameter refers to the primary Mach task that will "own" the lock. This task should be the one that is ultimately responsible for the subsequent destruction of the semaphore. The `task` parameter used when calling `semaphore_destroy` must match the one used when it was created.

For communication within the kernel, the `task` parameter should be the result of a call to `current_task`. For synchronization with a user process, you need to determine the underlying Mach task for that process. The details of such user-kernel synchronization are beyond the scope of this document.

The `policy` parameter is passed as the policy for the wait queue contained within the semaphore. The possible values are defined in `osfmk/mach/sync_policy.h`. Current possible values are

- `SYNC_POLICY_FIFO`
- `SYNC_POLICY_FIXED_PRIORITY`
- `SYNC_POLICY_PREPOST`

The FIFO policy is, as the name suggests, first-in-first-out. The fixed priority policy causes wait queue reordering based on fixed thread priority policies. The prepost policy causes the `semaphore_signal` function to not increment the counter if no threads are waiting on the queue. This policy is needed for creating condition variables (where a thread is expected to always wait until signalled). See the section "Wait Queues and Wait Primitives" (page 77) for more information.

The `semaphore_signal_thread` call takes a particular thread from the wait queue and places it back into one of the scheduler's wait-queues, thus making that thread available to be scheduled for execution. If `thread_act` is `NULL`, the first thread in the queue is similarly made runnable.

With the exception of `semaphore_create` and `semaphore_destroy`, these functions can also be called from user space via RPC. See "Calling RPC From User Applications" (page 116) for more information.

Condition Variables

The BSD portion of Mac OS X provides `tsleep`, `wakeup`, and `wakeup_one`, which are equivalent to condition variables with the addition of an optional time-out. Since they are more commonly used for waiting a predetermined period of time, these calls are discussed in more detail in the section "Using tsleep" (page 142) as part of the "Miscellaneous Kernel Services" (page 139) chapter.

Outside the BSD portion of the kernel, condition variables may be implemented using semaphores.

Locks

Mac OS X (and Mach in general) has three basic types of locks: spinlocks, mutexes, and read-write locks. Each of these has different uses and different problems. There are also many other types of locks that are not implemented in Mac OS X, such as spin-sleep locks, some of which may be useful to implement for performance comparison purposes.

Spinlocks

A spinlock is the simplest type of lock. In a system with a test-and-set instruction or the equivalent, the code looks something like this:

```
while (test_and_set(bit) != 0);
```

In other words, until the lock is available, it simply "spins" in a tight loop that keeps checking the lock until the thread's **time quantum** expires and the next thread begins to execute. Since the entire time quantum for the first thread must complete before the next thread can execute and (possibly) release the lock, a spinlock is very wasteful of CPU time, and should be used *only* in places where a mutex cannot be used, such as in a hardware exception handler or low-level interrupt handler.

Note that a thread may not block while holding a spinlock, because that could cause deadlock. Further, preemption is disabled on a given processor while a spinlock is held.

There are three basic types of spinlocks available in Mac OS X: `simple_lock_t`, `usimple_lock_t`, and `hw_lock_t`. You are strongly encouraged to not use `hw_lock_t`; it is only mentioned for the sake of completeness.

The `u` in `usimple` stands for uniprocessor, because they are the only spinlocks that provide actual locking on uniprocessor systems. Traditional simple locks, by contrast, disable preemption but do not spin on uniprocessor systems. Note that in most contexts, it is not useful to spin on a uniprocessor system, and thus you usually only need simple locks. Use of usimple locks is permissible for synchronization between thread context and interrupt context or between a uniprocessor and an intelligent device. However, in most cases, a mutex is a better choice.

Important
Simple locks that could potentially be shared between interrupt context and thread context must have their use coordinated with **spl** (see glossary). The IPL (interrupt priority level) must always be the same when acquiring the lock, otherwise deadlock may result.

The simple and usimple lock functions consist of the following:

```
void simple_lock_init(simple_lock_t, etap_event_t)
void simple_lock(simple_lock_t)
void simple_unlock(simple_lock_t)
unsigned int simple_lock_try(simple_lock_t)

void usimple_lock_init(usimple_lock_t, etap_event_t)
void usimple_lock(usimple_lock_t)
void usimple_unlock(usimple_lock_t)
unsigned int usimple_lock_try(usimple_lock_t)
void usimple_lock_held(usimple_lock_t)
void usimple_lock_none_held(void)
```

Prototypes for these locks can be found in `<kern/simple_lock.h>` or `xnu/osfmk/kern/simple_lock.h`.

The second argument to the lock initializer, of type `etap_event_t`, is a special feature of Mac OS X. It is part of the Event Trace Analysis Package, which can be used for lock contention profiling. The list of values for this parameter is found in

xnu/osfmk/kern/etap.c. Unless you are planning to use this facility, you should probably use the value ETAP_NO_TRACE. Tracing is not enabled in the default kernel build, so you must enable the appropriate options in osfmk/conf/MASTER and recompile the kernel. The details of lock tracing are beyond the scope of this document.

Mutexes

A mutex, mutex lock, or sleep lock, is similar to a spinlock, except that instead of constantly polling, it places itself on a queue of threads waiting for the lock, then yields the remainder of its time quantum. It does not execute again until the thread holding the lock wakes it (or in some user space variations, until an asynchronous signal arrives).

Mutexes are more efficient than spinlocks for most purposes. However, they are less efficient in multiprocessing environments where the expected lock-holding time is relatively short. If the average time is relatively short but occasionally long, spin/sleep locks may be a better choice. Although Mac OS X does not support spin/sleep locks in the kernel, they can be easily implemented on top of existing locking primitives. If your code performance improves as a result of using such locks, however, you should probably look for ways to restructure your code, such as using more than one lock or moving to read-write locks, depending on the nature of the code in question. See "Spin/Sleep Locks" (page 137) for more information.

Because mutexes are based on blocking, they can only be used in places where blocking is allowed. For this reason, mutexes cannot be used in the context of interrupt handlers. Interrupt handlers are not allowed to block because interrupts are disabled for the duration of an interrupt handler, and thus, if an interrupt handler blocked, it would prevent the scheduler from receiving timer interrupts, which would prevent any other thread from executing, resulting in deadlock.

For a similar reason, it is not reasonable to block within the scheduler. Also, blocking within the VM system can easily lead to deadlock if the lock you are waiting for is held by a task that is paged out.

However, unlike simple locks, it is permissible to block while holding a mutex. This would occur, for example, if you took one lock, then tried to take another, but the second lock was being held by another thread. However, this is generally not recommended unless you carefully scrutinize all uses of that mutex for possible circular waits, as it can result in deadlock. You can avoid this by always taking locks in a certain order.

In general, blocking while holding a mutex specific to your code is fine as long as you wrote your code correctly, but blocking while holding a more global mutex is probably not, since you may not be able to guarantee that other developers' code obeys the same ordering rules.

A Mach mutex is of type `mutex_t`. The functions that operate on mutexes include:

```
mutex_t      *mutex_alloc(etap_event_t)
void         mutex_free(mutex_t *)
void         mutex_lock(mutex_t *)
void         mutex_unlock(mutex_t *)
boolean_t    mutex_try(mutex_t *)
void         mutex_pause(void)
```

as described in `<kern/lock.h>` or `xnu/osfmk/kern/lock.h`.

The argument to `mutex_alloc`, of type `etap_event_t`, is part of the Event Trace Analysis Package, used for lock contention profiling. For a list of possible values, see `xnu/osfmk/kern/etap.c`. As with spinlocks, unless you are planning to use this facility, you should probably use the value `ETAP_NO_TRACE`. Tracing is not enabled in the default kernel build, so you must enable the appropriate options in `osfmk/conf/MASTER` and recompile the kernel. The details of lock tracing are beyond the scope of this document.

The `mutex_pause` function puts the current thread to sleep for 1 ms. You might use this function if, for example, you have a thread that needs to do something while waiting for a lock to become available. In such a design, your code first calls `mutex_try` on the lock. If that fails, a block of code executes, ending with a `mutex_pause`, followed by another `mutex_try`, and so on. The use of `mutex_pause` prevents the code from resembling a spinlock (which could potentially cause the thread's priority to be lowered).

There is a second mutex API in the kernel called a sync lock. The sync lock API is exported to user space, and is described in `osfmk/kern/sync_lock.c`. Sync locks should generally be avoided in the kernel in favor of the `mutex_` functions unless you are synchronizing communication between the kernel and user space or you need lock sets. Sync locks and lock sets are beyond the scope of this document.

Read-Write Locks

Read-write locks (also called shared-exclusive locks) are somewhat different from traditional locks in that they are not always exclusive locks. A read-write lock is useful when shared data can be reasonably read concurrently by multiple threads except while a thread is modifying the data. Read-write locks can dramatically improve performance if the majority of operations on the shared data are in the form of reads (since it allows concurrency), while having negligible impact in the case of multiple writes.

A read-write lock allows this sharing by enforcing the following constraints:

- Multiple readers can hold the lock at any time.
- Only one writer can hold the lock at any given time.
- A writer must block until all readers have released the lock before obtaining the lock for writing.
- Readers arriving while a writer is waiting to acquire the lock will block until after the writer has obtained and released the lock.

The first constraint allows read sharing. The second constraint prevents write sharing. The third prevents read-write sharing, and the fourth prevents starvation of the writer by a steady stream of incoming readers.

Mach read-write locks also provide the ability for a reader to become a writer and vice-versa. In locking terminology, an upgrade is when a reader becomes a writer, and a downgrade is when a writer becomes a reader. To prevent deadlock, some additional constraints must be added for upgrades and downgrades:

- Upgrades are favored over writers.
- The second and subsequent concurrent upgrades will fail, causing that thread's read lock to be released.

The first constraint is necessary because the reader requesting an upgrade is holding a read lock, and the writer would not be able to obtain a write lock until the reader releases its read lock. In this case, the reader and writer would wait for each other forever. The second constraint is necessary to prevents the deadlock that would occur if two readers wait for the other to release its read lock so that an upgrade can occur.

The functions that operate on read-write locks are:

```
void        lock_init(lock_t *l, boolean_t can_sleep, etap_event_t event,
            etap_event_t i_event)
lock_t *    lock_alloc(boolean_t can_sleep, etap_event_t event,
            etap_event_t i_event)
void        lock_free(lock_t *)
void        lock_read(lock_t *)
void        lock_write(lock_t *)
boolean_t   lock_read_to_write(lock_t *)
void        lock_write_to_read(lock_t *)
void        lock_done(lock_t *)
void        lock_read_done(lock_t *)
void        lock_write_done(lock_t *)
```

This is a more complex interface than that of the other locking mechanisms, and actually is the interface upon which the other locks are built. Most of the functions are straightforward except for `lock_init` and `lock_done`.

The first argument to the function `lock_init` is, of course, a pointer to the lock structure to be locked. The second argument, `can_sleep`, is a boolean value that indicates whether the lock is allowed to block (and consequently, whether a thread is allowed to block while holding the lock). If this argument is true, the lock is fundamentally a mutex. Otherwise, it behaves like a spinlock. The third and fourth arguments are for gathering trace statistics using the Event Trace Analysis Package. As before, unless you plan to use the trace facility, you should simply pass in `ETAP_NO_TRACE`.

The function `lock_done` releases a lock. It does not matter whether the lock is held for reading or writing. However, most parts of the kernel create macros with names like `lock_read_done` and `lock_write_done` to make the code more understandable.

Spin / Sleep Locks

Spin / sleep locks are not implemented in the Mac OS X kernel. However, they can be easily implemented on top of existing locks if desired.

For short waits on multiprocessor systems, the amount of time spent in the context switch can be greater than the amount of time spent spinning. When the time spent spinning while waiting for the lock becomes greater than the context switch overhead, however, mutexes become more efficient. For this reason, if there is a large degree of variation in wait time on a highly contended lock, spin / sleep locks may be more efficient than traditional spinlocks or mutexes.

Ideally, a program should be written in such a way that the time spent holding a lock is always about the same, and the choice of locking is clear. However, in some cases, this is not practical for a highly contended lock. In those cases, you may consider using spin/sleep locks.

The basic principle of spin/sleep locks is simple. A thread takes the lock if it is available. If the lock is not available, the thread may enter a spin cycle. After a certain period of time (usually a fraction of a time quantum or a small number of time quanta), the spin routine's time-out is reached, and it returns failure. At that point, the lock places the waiting thread on a queue and puts it to sleep.

In other variations on this design, spin/sleep locks determine whether to spin or sleep according to whether the lock-holding thread is currently on another processor (or is about to be).

For short wait periods on multiprocessor computers, the spin/sleep lock is more efficient than a mutex, and roughly as efficient as a standard spinlock. For longer wait periods, the spin/sleep lock is significantly more efficient than the spinlock and only slightly less efficient than a mutex. There is a period near the transition between spinning and sleeping in which the spin/sleep lock may behave significantly worse than either of the basic lock types, however. Thus, spin/sleep locks should not be used unless a lock is heavily contended and has widely varying hold times. When possible, you should rewrite the code to avoid such designs.

CHAPTER 16

Miscellaneous Kernel Services

This chapter contains information about miscellaneous services provided by the Mac OS X kernel. For most projects, you will probably never need to use most of these services, but if you do, you will find it hard to do without them.

This chapter contains these sections: "Using Kernel Time Abstractions" (page 139), "Boot Option Handling" (page 143), "Queues" (page 144), and "Installing Shutdown Hooks" (page 145).

Using Kernel Time Abstractions

There are two basic groups of time abstractions in the kernel. One group includes functions that provide delays and timed wake-ups. The other group includes functions and variables that provide the current wall clock time, the time used by a given process, and other similar information. This section describes both aspects of time from the perspective of the kernel.

Obtaining Time Information

There are a number of ways to get basic time information from within the kernel. The officially approved methods are those that Mach exports in `kern/clock.h`. These include the following:

```
void clock_get_uptime(uint64_t *result);
mach_timespec_t clock_get_system_value(void);
mach_timespec_t clock_get_calendar_value(void);
mach_timespec_t clock_get_calendar_offset(void);
```

The function `clock_get_uptime` returns a value in `AbsoluteTime` units. For more information on using `AbsoluteTime`, see "Using Mach Absolute Time Functions" (page 141).

The remaining functions return type `mach_timespec_t`, which is similar to the traditional BSD `struct timespec`, except that fractions of a second are measured in nanoseconds instead of microseconds:

```
struct mach_timespec {
    unsigned int tv_sec;
    clock_res_t tv_nsec;
};
typedef struct mach_timespec *mach_timespec_t;
```

The function `clock_get_calendar_value` returns time as the number of seconds and nanoseconds since January 1, 1970 (the traditional UNIX time measure). The function `clock_get_system_value` returns time as the number of seconds and nanoseconds that the computer has been turned on. Finally, the function `clock_get_calendar_offset` returns the difference between the two.

In addition to the traditional Mach functions, if you are writing code in BSD portions of the kernel you can also get the current calendar (wall clock) time as a BSD `timeval`, as well as find out the calendar time when the system was booted by doing the following:

```
#include <sys/kernel.h>
struct timeval tv=time; /* calendar time */
struct timeval tv_boot=boottime; /* calendar time when booting occurred */
```

For other information, you should use the Mach functions listed previously.

Event and Timer Waits

Each part of the Mac OS X kernel has a distinct API for waiting a certain period of time. In most cases, you can call these functions from other parts of the kernel. The I/O Kit provides `IODelay` and `IOSleep`. Mach provides functions based on `AbsoluteTime`, as well as a few based on microseconds. BSD provides `tsleep`.

Using IODelay and IOSleep

IODelay, provided by the I/O Kit, abstracts a timed spin. If you are delaying for a short period of time, and if you need to be guaranteed that your wait will not be stopped prematurely by delivery of asynchronous events, this is probably the best choice. If you need to delay for several seconds, however, this is a bad choice, because the CPU that executes the wait will spin until the time has elapsed, unable to handle any other processing.

IOSleep puts the currently executing thread to sleep for a certain period of time. There is no guarantee that your thread will execute after that period of time, nor is there a guarantee that your thread will not be awakened by some other event before the time has expired. It is roughly equivalent to the sleep call from user space in this regard.

The use of IODelay and IOSleep are straightforward. Their prototypes are:

```
IODelay(unsigned microseconds);
IOSleep(unsigned milliseconds);
```

Note the differing units. It is not practical to put a thread to sleep for periods measured in microseconds, and spinning for several milliseconds is also inappropriate.

Using Mach Absolute Time Functions

The following Mach time functions are commonly used. Several others are described in osfmk/kern/clock.h.

```
void delay(uint64_t microseconds);
void clock_delay_for_interval(uint32_t interval, uint32_t scale_factor);
void clock_delay_until(uint64_t deadline);
void clock_absolutetime_interval_to_deadline(uint64_t abstime,
            uint64_t *result);
void nanoseconds_to_absolutetime(uint64_t nanoseconds, uint64_t *result);
void absolutetime_to_nanoseconds(uint64_t abstime, uint64_t *result);
```

These functions are generally straightforward. However, a few points deserve explanation. Unless specifically stated, all times, deadlines, and so on, are measured in abstime units. The abstime unit is equal to the length of one bus cycle, so the duration is dependent on the bus speed of the computer. For this reason, Mach provides conversion routines between abstime units and nanoseconds.

The function `clock_delay_for_interval`, however, is an unusual function in that its unit is based in nanoseconds, but is variable according to a scaling factor. The most common way to use this function is something like the following:

```
clock_delay_for_interval(32, NSEC_PER_USEC);
```

That statement causes a delay of 32 microseconds since `NSEC_PER_USEC = 1000`. Similarly, a scaling factor of 1 million would delay for a millisecond, and so on. To put it another way, this function waits `interval * scale_factor` nanoseconds.

Using `tsleep`

In addition to Mach and I/O Kit routines, BSD provides `tsleep`, which is the recommended way to delay while holding the kernel or network funnel. It will *not* work if you are not holding a funnel. In other parts of the kernel, you should with either use `wait_queue` functions or use `assert_wait` and `thread_wakeup` functions, both of which are closely tied to the Mach scheduler, and are described in "Kernel Thread APIs" (page 75).

The `tsleep` call is similar to a condition variable. It puts a thread to sleep until `wakeup` or `wakeup_one` is called on that channel. Unlike a condition variable, however, you can set a timeout measured in clock ticks. This means that it is both a synchronization call and a delay. The prototypes follow:

```
sleep(void *channel, int priority);
tsleep0(void *channel, int priority, char *subsystem, int timeout,
    int (*continuation)(void));
tsleep(void *channel, int priority, char *subsystem, int timeout);
wakeup(void *channel);
wakeup_one(void *channel);
```

The three sleep calls are similar except in the amount of debugging support (`subsystem`) and in whether they allow timeouts or calling a function on `wakeup`.

In these functions, `channel` is a unique identifier representing a single condition upon which you are waiting. Normally, when `tsleep` is used, you are waiting for a change to occur in a data structure. In such cases, it is common to use the address of that data structure as the value for `channel`, as this ensures that no code elsewhere in the system will be using the same value.

The priority argument has two effects. First, when wakeup is called, threads are inserted in the scheduling queue at this priority. Second, the value of priority modifies signal delivery behavior. If the value of priority is negative, signal delivery cannot wake the thread early. If the bit (priority & PCATCH) is set, tsleep0 does not call the continuation function upon waking up from sleep and returns a value of 1.

The subsystem argument is a short text string that represents the subsystem that is waiting on this channel. This is used solely for debugging purposes.

The timeout argument is used to set a maximum wait time. The thread may wake sooner, however, if wakeup or wakeup_one is called on the appropriate channel. It may also wake sooner if a signal is received, depending on the value of priority.

Finally, the continuation argument is a function that is called when the thread is woken up by a signal or a call to wakeup or wakeup_one. If the wake occurred programmatically, this function is called (if it is not NULL). If the wake occurred as a result of a signal, it is called only if the PCATCH bit is not set in the priority value.

Boot Option Handling

Mac OS X provides a simple parse routine, PE_parse_boot_arg, for basic boot argument passing. It supports both flags and numerical value assignment. For obtaining values, you write code similar to the following:

```
unsigned int argval;

if (PE_parse_boot_arg("argflag", &argval)) {
    /* check for reasonable value */
    if (argval < 10 || argval > 37)
        argval = 37;
} else {
    /* use default value */
    argval = 37;
}
```

Since PE_parse_boot_arg returns a nonzero value if the flag exists, you can check for the presence of a flag by using a flag that starts with a dash (-) and ignoring the value stored in argvalue.

The PE_parse_boot_arg function can also be used to get a string argument. To do this, you must pass in the address of an array of type char as the second argument. The behavior of PE_parse_boot_arg is undefined if a string is passed in for a numeric variable or vice versa. Its behavior is also undefined if a string exceeds the storage space allocated. Be sure to allow enough space for the largest reasonable string including a null delimiter. No attempt is made at bounds checking, since an overflow is generally a fatal error and should reasonably prevent booting.

Queues

As part of its BSD infrastructure, the Mac OS X kernel provides a number of basic support macros to simplify handling of linked lists and queues. These are implemented as C macros, and assume a standard C struct. As such, they are probably not suited for writing code in C++.

The basic types of lists and queues included are

- SLIST, a singly linked list
- STAILQ, a singly linked tail queue
- LIST, a doubly linked list
- TAILQ, a doubly linked tail queue

SLIST is ideal for creating stacks or for handling large sets of data with few or no removals. Arbitrary removal, however, requires an O(n) traversal of the list.

STAILQ is similar to SLIST except that it maintains pointers to both ends of the queue. This makes it ideal for simple FIFO queues by adding entries at the tail and fetching entries from the head. Like SLIST, it is inefficient to remove arbitrary elements.

LIST is a doubly linked version of SLIST. The extra pointers require additional space, but allow O(1) (constant time) removal of arbitrary elements and bidirectional traversal.

TAILQ is a doubly linked version of STAILQ. Like LIST, the extra pointers require additional space, but allow O(1) (constant time) removal of arbitrary elements and bidirectional traversal.

Because their functionality is relatively simple, their use is equally straightforward. These macros can be found in xnu/bsd/sys/queue.h.

Installing Shutdown Hooks

Although Mac OS X does not have traditional BSD-style shutdown hooks, the I/O Kit provides equivalent functionality in recent versions. Since the I/O Kit provides this functionality, you must call it from C++ code.

To register for notification, you call registerPrioritySleepWakeInterest (described in IOKit/RootDomain.h) and register for sleep notification. If the system is about to be shut down, your handler is called with the message kIOMessageSystemWillPowerOff. If the system is about to reboot, your handler gets the message kIOMessageSystemWillRestart.

If you no longer need to receive notification (for example, if your KEXT gets unloaded), be certain to release the notifier with IONofitier::release to avoid a kernel panic on shutdown.

Kernel Extension Overview

As discussed in the chapter "Kernel Architecture Overview" (page 10), 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 cause as much trouble as a badly behaved application or extension could in Mac OS 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 kernel **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 threading in applications is just as efficient as threading inside the kernel, so efficiency should not be an issue. Unless your application requires low-level access to kernel interfaces, 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*. Even if your code was a system extension in Mac OS 9, that does not necessarily mean that it should 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 (built-in) hardware needs to interrupt the CPU and execute a handler.
- The primary client of your code is inside the kernel—for example, a block device whose primary client is a file system.
- Your code needs to access kernel interfaces that are not exported to user space.
- Your code has other special requirements that cannot be satisfied in a user space application.

If your code does not meet any of the above criteria (and possibly even if it does), 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 150) 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.

Implementation of a Kernel Extension (KEXT)

Kernel extensions 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**—a 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.
- **KEXT binary**—a file in Mach-O format, containing the actual binary code used by the KEXT. A KEXT binary (also known as a kernel module or **KMOD**) represents the minimum unit of code that can be loaded into the kernel. A KEXT usually contains one KEXT binary. If no KEXT binaries are included, the information property list file must contain a reference to another KEXT and change its default settings.
- **Resources**—for example, icons or localization dictionaries. Resources are optional; they may be useful for a KEXT that needs to display a dialog or menu. At present, no resources are explicitly defined for use with KEXTs.
- **KEXT bundles**—a kext can contain other KEXTs. This can be used for plug-ins that augment features of a KEXT.

Kernel Extension Dependencies

Any KEXT can declare that it is dependent upon any other KEXT. The developer lists these dependencies in the "Requires" field of the module's property list file.

Before a KEXT is loaded, all of its requirements are checked. Those required extensions (and their requirements) are loaded first, iterating back through the lists until there are no more required extensions to load. Only after all requirements are met, is the requested KEXT 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 "I/O Kit Overview" (page 85), for an explanation of drivers and families.

It is important to list all dependencies for each KEXT. If your KEXT fails to do so, your KEXT may not load due to unrecognized symbols, thus rendering the KEXT useless. Dependencies in KEXTs 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 KEXT requirements.

Building and Testing Your Extension

After creating the necessary property list and C or C++ source files, you use Project Builder to build your KEXT. 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 or in the Console application..You also need to log in to the **root** account (or use the `su` or `sudo` 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. A manual page for `kextload` is included in Mac OS X. (On Mac OS X prior to 10.2, you must use the `kmodload` command instead.)

Note that this command is useful only 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 151) 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. (You can use `sudo` to gain root privileges if you don't want to enable a root password.)

Kernel debugging is performed using two Mac OS X computers, called the development or debug host and the debug target. These computers 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 loaded and run) on the target. The debugger is launched and run on the debug host. You can also rebuild your KEXT on the debug host, after you fix any errors you find.

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

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.

Developers generally debug KEXTs 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`.

Some features of `gdb` are unavailable when debugging KEXTs because of implementation limitations. For example:

- You can't use `gdb` to call a function or method in a KEXT.

- You should not use `gdb` to debug interrupt routines.

The former is largely a barrier introduced by the C++ language. The latter may work in some cases but is not recommended due to the potential for `gdb` to interrupt something upon which **kdp** (the kernel shim used by `gdb`) depends in order to function properly.

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.

Additional information about debugging can be found in "When Things Go Wrong: Debugging the Kernel" (page 159).

Installed KEXTs

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

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

KEXTs are usually installed in the folder `/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 KEXT 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 KEXTs 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 KEXTs (and their dependencies).

CHAPTER 18

Building and Debugging Kernels

This chapter is not about building kernel extensions (KEXTs). There are a number of good KEXT tutorials on Apple's developer documentation site (http://developer.apple.com/techpubs). This chapter is about adding new in-kernel modules (optional parts of the kernel), building kernels, and debugging kernel and kernel extension builds.

The discussion is divided into three sections. The first, "Adding New Files or Modules" (page 153), describes how to add new functionality into the kernel itself. You should only add files into the kernel when the use of a KEXT is not possible (for example, when adding certain low-level motherboard hardware support).

The second section, "Building Your First Kernel" (page 156), describes how to build a kernel, including how to build a kernel with debugger support, how to add new options, and how to obtain sources that are of similar vintage to those in a particular version of Mac OS X or Darwin.

The third section, "When Things Go Wrong: Debugging the Kernel" (page 159), tells how to debug a kernel or kernel module using `ddb` and `gdb`. This is a must-read for anyone doing kernel development.

Adding New Files or Modules

In this context, the term module is used loosely to refer to a collection of related files in the kernel that are controlled by a single `config` option at compile time. It does not refer to loadable modules (KEXTs). This section describes how to add additional files that will be compiled into the kernel, including how to add a new `config` option for an additional module.

Modifying the Configuration Files

The details of adding a new file or module into the kernel differ according to what portion of the kernel contains the file. If you are adding a new file or module into the Mach portion of the kernel, you need to list it in various files in `xnu/osfmk/conf`. For the BSD portion of the kernel, you should list it in various files in `xnu/bsd/conf`. In either case, the procedure is basically the same, just in a different directory.

This section is divided into two subsections. The first describes adding the module itself and the second describes enabling the module.

Adding the Files or Modules

In the appropriate `conf` directory, you need to add your files or modules into various files. The files `MASTER`, `MASTER.ppc`, and `MASTER.i386` contain the list of configuration options that should be built into the kernel for all architectures, PowerPC, and i386, respectively.

These are supplemented by `files`, `files.ppc`, and `files.i386`, which contain associations between compile options and the files that are related to them for their respective architectures.

The format for these two files is relatively straightforward. If you are adding a new module, you should first choose a name for that module. For example, if your module is called `mach_foo`, you should then add a new option line near the top of `files` that is whitespace (space or tab) delimited and looks like this:

```
OPTIONS/mach_foo      optional mach_foo
```

The first part defines the name of the module as it will be used in `#if` statements in the code. (See "Modifying the Source Code Files" (page 155) for more information.) The second part is always the word optional. The third part tells the name of the option as used to turn it on or off in a `MASTER` file. Any line with `mach_foo` in the last field will be enabled only if there is an appropriate line in a `MASTER` file.

Then, later in the file, you add

```
osfmk/foo/foo_main.c              optional mach_foo
osfmk/foo/foo_bar.c               optional mach_foo
```

and so on, for each new file associated with that module. This also applies if you are adding a file to an existing module. If you are adding a file that is not associated with any module at all, you add a line that looks like the following to specify that this file should always be included:

```
osfmk/crud/mandatory_file.c       standard
```

If you are not adding any modules, then you're done. Otherwise, you also need to enable your option in one of the `MASTER` files.

Enabling Module Options

To enable a module option (as described in the `files` files), you must add an entry for that option into one of the `MASTER` files. If your code is not a BSD pseudo-device, you should add something like the following:

```
options  MACH_FOO
```

Otherwise, you should add something like this:

```
pseudo-device    mach_foo
```

In the case of a pseudo-device (for example, `/dev/random`), you can also add a number. When your code checks to see if it should be included, it can also check that number and allocate resources for more than one pseudo-device. The meaning of multiple pseudo-devices is device-dependent. An example of this is `ppp`, which allocates resources for two simultaneous PPP connections. Thus, in the `MASTER.ppc` file, it has the line:

```
pseudo-device    ppp 2
```

Modifying the Source Code Files

In the Mac OS X kernel, all source code files are automatically compiled. It is the responsibility of the C file itself to determine whether its contents need to be included in the build or not.

In the example above, you created a module called mach_foo. Assume that you want this file to compile only on PowerPC-based computers. In that case, you should have included the option only in MASTER.ppc and not in MASTER.i386. However, by default, merely specifying the file foo_main.c in files causes it to be compiled, regardless of compile options specified.

To make the code compile only when the option mach_foo is included in the configuration, you should begin each C source file with the lines

```
#include <mach_foo.h>
#if (MACH_FOO > 0)
```

and end it with

```
#endif /* MACH_FOO */
```

If mach_foo is a pseudo-device and you need to check the number of mach_foo pseudo-devices included, you can do further tests of the value of MACH_FOO.

Note that the file <mach_foo.h> is not something you create. It is created by the makefiles themselves. You must run make exporthdrs before make all to generate these files.

Building Your First Kernel

Before you can build a kernel, you must first obtain source code. Source code for the Mac OS X kernel can be found in the Darwin xnu project on http://www.opensource.apple.com. To find out your current kernel version, use the command uname -a. If you run into trouble, search the archives of the darwin-kernel and darwin-development mailing lists for information. If that doesn't help, ask for assistance on either list. The list archives and subscription information can be found at http://www.lists.apple.com.

After you have obtained and extracted the sources, you will need to compile several support tools. Get the bootstrap_cmds, Libstreams, and cctools packages from http://www.opensource.apple.com. Extract the files from these .tar packages, then do the following:

```
cd bootstrap_cmds-version/relpath.tproj
make
sudo make install
cd ../../Libstreams-version
make
make install
cd ../cctools
```

In the `cctools` package, modify the `Makefile`, and change the `COMMON_SUBDIRS` line to read:

```
COMMON_SUBDIRS = libstuff libmacho misc
```

Finally, issue the following commands:

```
make RC_OS=macos
sudo cp misc/seg_hack.NEW /usr/local/bin/seg_hack
cd ld
make RC_OS=macos kld_build
sudo cp static_kld/libkld.a /usr/local/lib
```

Congratulations. You now have all the necessary tools, libraries, and header files to build a kernel.

The next step is to compile the kernel itself. First, change directories into the `xnu` directory. Next, you need to set a few environment variables appropriately. For your convenience, the kernel sources contain shell scripts to do this for you. If you are using sh, bash, zsh, or some other Bourne-compatible shell, issue the following command:

```
source SETUP/setup.sh
```

If you are using csh, tcsh, or a similar shell, use the following command:

```
source SETUP/setup.csh
```

Then, you should be able to type

```
make exporthdrs
make all
```

and get a working kernel in `BUILD/obj/RELEASE_PPC/mach_kernel` (assuming you are building a `RELEASE` kernel for PowerPC, of course).

Building an Alternate Kernel Configuration

When building a kernel, you may want to build a configuration other than the RELEASE configuration (the default shipping configuration). Additional configurations are RELEASE_TRACE, DEBUG, DEBUG_TRACE, and PROFILE. These configurations add various additional options (except PROFILE, which is reserved for future expansion, and currently maps onto RELEASE).

The most useful and interesting configurations are RELEASE and DEBUG. The release configuration should be the same as a stock Apple-released kernel, so this is interesting only if you are building source that differs from that which was used to build the kernel you are already running. Compiling a kernel without specifying a configuration results in the RELEASE configuration being built.

The DEBUG configuration enables ddb, the in-kernel serial debugger. The ddb debugger is helpful to debug panics that occur early in boot or within certain parts of the Ethernet driver. It is also useful for debugging low-level interrupt handler routines that cannot be debugged by using the more traditional gdb.

To compile an alternate kernel configuration, you should follow the same basic procedure as outlined previously, changing the final make statement slightly. For example, to build the DEBUG configuration, instead of typing

```
make all
```

you type

```
make KERNEL_CONFIGS=DEBUG all
```

and wait.

To turn on additional compile options, you must modify one of the MASTER files. For information on modifying these files, see the section "Enabling Module Options" (page 155).

When Things Go Wrong: Debugging the Kernel

No matter how careful your programming habits, sometimes things don't work right the first time. Kernel panics are simply a fact of life during development of kernel extensions or other in-kernel code.

There are a number of ways to track down problems in kernel code. In many cases, you can find the problem through careful use of `printf` or `IOLog` statements. Some people swear by this method, and indeed, given sufficient time and effort, any bug can be found and fixed without using a debugger.

Of course, the key words in that statement are "given sufficient time and effort." For the rest of us, there are debuggers: `gdb` and `ddb`.

Setting Debug Flags in Open Firmware

With the exception of kernel panics or calls to `PE_enter_debugger`, it is not possible to do remote kernel debugging without setting debug flags in Open Firmware. These flags are relevant to both `gdb` and `ddb` debugging and are important enough to warrant their own section.

To set these flags, you can either use the `nvram` program (from the Mac OS X command line) or access your computer's Open Firmware. You can access Open Firmware this by holding down Command-Option-O-F at boot time. For most computers, the default is for Open Firmware to present a command–line prompt on your monitor and accept input from your keyboard. For some older computers you must use a serial line at 38400, 8N1. (Technically, such computers are not supported by Mac OS X, but some are usable under Darwin, and thus they are mentioned here for completeness.)

From an Open Firmware prompt, you can set the flags with the `setenv` command. From the Mac OS X command line, you would use the `nvram` command. Note that when modifying these flags you should always look at the old value for the appropriate Open Firmware variables and add the `debug` flags.

For example, if you want to set the debug flags to `0x4`, you use one of the following commands. For computers with recent versions of Open Firmware, you would type

```
printenv boot-args
setenv boot-args original_contents debug=0x4
```

from Open Firmware or

```
nvram boot-args
nvram boot-args="original_contents debug=0x4"
```

from the command line (as root).

For older firmware versions, the interesting variable is `boot-command`. Thus, you might do something like

```
printenv boot-command
setenv boot-command 0 bootr debug=0x4
```

from Open Firmware or

```
nvram boot-command
nvram boot-command="0 bootr debug=0x4"
```

from the command line (as root).

Of course, the more important issue is what value to choose for the debug flags. Table 18-1 lists the debugging flags that are supported in Mac OS X.

Table 18-1 Debugging flags

Symbolic name	Flag	Meaning
`DB_HALT`	0x01	Halt at boot-time and wait for debugger attach (`gdb`).
`DB_PRT`	0x02	Send kernel debugging `printf` output to console.
`DB_NMI`	0x04	Drop into debugger on NMI (Command–Power or interrupt switch).
`DB_KPRT`	0x08	Send kernel debugging `kprintf` output to serial port.
`DB_KDB`	0x10	Make `ddb` (`kdb`) the default debugger (requires a custom kernel).
`DB_SLOG`	0x20	Output certain diagnostic info to the system log.

Table 18-1 Debugging flags

Symbolic name	Flag	Meaning
DB_ARP	0x40	Allow debugger to ARP and route (allows debugging across routers and removes the need for a permanent ARP entry, but is a potential security hole)—not available in all kernels.
DB_KDP_BP_DIS	0x80	Support old versions of gdb on newer systems.
DB_LOG_PI_SCRN	0x100	Disable graphical panic dialog.

The option DB_KDP_BP_DIS is not available on all systems, and should not be important if your target and host systems are running the same or similar versions of Mac OS X with matching developer tools. The last option is only available in Mac OS 10.2 and later.

Note: If your target machine is running Mac OS X Server, your system will automatically reboot within seconds after a crash. This is caused by a watchdog timer in hardware. You can disable the automatic reboot on crash feature in the server administration tool.

Choosing a Debugger

There are two basic debugging environments supported by Mac OS X: ddb and gdb. ddb is a built-in debugger that works over a serial line. By contrast, gdb is supported using a debugging shim built into the kernel, which allows a remote computer on the same physical network to attach after a panic (or sooner if you pass certain options to the kernel).

For problems involving network extensions or low-level operating system bringups, ddb is the only way to do debugging. For other bugs, gdb is generally easier to use. For completeness, this chapter describes how to use both ddb and gdb to do basic debugging. Since gdb itself is well documented and is commonly used for application programming, this chapter assumes at least a passing knowledge of the basics of using gdb and focuses on the areas where remote (kernel) gdb differs.

Using gdb for Kernel Debugging

gdb, short for the GNU Debugger, is a piece of software commonly used for debugging software on UNIX and Linux systems. This section assumes that you have used gdb before, and does not attempt to explain basic usage.

In standard Mac OS X builds (and in your builds unless you compile with ddb support), gdb support is built into the system but is turned off except in the case of a kernel panic.

Of course, many software failures in the kernel do not result in a kernel panic but still cause aberrant behavior. For these reasons, you can pass additional flags to the kernel to allow you to attach to a remote computer early in boot or after a nonmaskable interrupt (NMI), or you can programmatically drop into the debugger in your code.

You can cause the test computer (the debug target) to drop into the debugger in the following ways:

- debug on panic
- debug on NMI
- debug on boot
- programmatically drop into the default debugger

  The function PE_enter_debugger can be called from anywhere in the kernel, although if gdb is your default debugger, a crash will result if the network hardware is not initialized or if gdb cannot be used in that particular context. This call is described in the header pexpert/pexpert.h.

After you have decided what method to use for dropping into the debugger on the target, you must configure your debug host (the computer that will actually be running gdb). Your debug host should be running a version of Mac OS X that is comparable to the version running on your target host. However, it should not be running a customized kernel, since a debug host crash would be problematic, to say the least.

Note: It is possible to use a non-Mac OS X system as your debug host. This is not a trivial exercise, however, and a description of building a cross-gdb is beyond the scope of this document.

When using gdb, the best results can be obtained when the source code for the customized kernel is present on your debug host. This not only makes debugging easier by allowing you to see the lines of code when you stop execution, it also makes it easier to modify those lines of code. Thus, the ideal situation is for your debug host to also be your build computer. This is not required, but it makes things easier. If you are debugging a kernel extension, it generally suffices to have the source for the kernel extension itself on your debug host. However, if you need to see kernel-specific structures, having the kernel sources on your debug host may also be helpful.

Once you have built a kernel using your debug host, you must then copy it to your target computer and reboot the target computer. At this point, if you are doing panic-only debugging, you should trigger the panic. Otherwise, you should tell your target computer to drop into the debugger by issuing an NMI (or by merely booting, in the case of debug=0x1).

Next, unless your kernel supports ARP while debugging (and unless you enabled it with the appropriate debug flag), you need to add a permanent ARP entry for the target. It will be unable to answer ARP requests while waiting for the debugger. This ensures that your connection won't suddenly disappear. The following example assumes that your target is target.foo.com with an IP number of 10.0.0.69:

```
$ ping -c 1 target_host_name
ping results: ....
$ arp -an
target.foo.com (10.0.0.69): 00:a0:13:12:65:31
$ arp -s target.foo.com 00:a0:13:12:65:31
$ arp -an
target.foo.com (10.0.0.69) at00:a0:13:12:65:31 permanent
```

Now, you can begin debugging by doing the following:

```
gdb /path/to/mach_kernel
source /path/to/xnu/osfmk/.gdbinit
p proc0
source /path/to/xnu/osfmk/.gdbinit
target remote-kdp
attach 10.0.0.69
```

Note that the mach kernel passed as an argument to `gdb` should be the symbol–laden kernel file located in `BUILD/obj/DEBUG_PPC/mach_kernel.sys` (for debug kernel builds, `RELEASE_PPC` for non-debug builds), not the bootable kernel that you copied onto the debug target. Otherwise most of the `gdb` macros will fail. The correct kernel should be several times as large as a normal kernel.

You must do the `p proc0` command and source the `.gdbinit` file (from the appropriate kernel sources) twice to work around a bug in `gdb`. Of course, if you do not need any of the macros in `.gdbinit`, you can skip those two instructions. The macros are mostly of interest to people debugging aspects of Mach, though they also provide ways of obtaining information about currently loaded KEXTs.

WARNING
It may not be possible to detach in a way that the target computer's kernel continues to run. If you detach, the target hangs until you reattach. It is not always possible to reattach, though the situation is improving in this area. Do not detach from the remote kernel!

If you are debugging a kernel module, you need to do some additional work to get debugging symbol information about the module. First, you need to know the load address for the module. You can get this information by running `kextstat` (`kmodstat` on systems running Mac OS X 10.1 or earlier) as root on the target.

If you are already in the debugger, then assuming the target did not panic, you should be able to use the `continue` function in `gdb` to revive the target, get this information, then trigger another NMI to drop back into the debugger.

If the target is no longer functional, and if you have a fully symbol–laden kernel file on your debug host that matches the kernel on your debug target, you can use the `showallkmods` macro to obtain this information. Obtaining a fully symbol–laden kernel generally requires compiling the kernel yourself.

Once you have the load address of the module in question, you need to create a symbol file for the module. You do this in different ways on different versions of Mac OS X.

For versions 10.1 and earlier, you use the `kmodsyms` program to create a symbol file for the module. If your KEXT is called `mykext` and it is loaded at address 0xf7a4000, for example, you change directories to `mykext.kext/Contents/MacOS` and type:

```
kmodsyms -k path/to/mach_kernel -o mykext.sym mykext@0xf7a4000
```

Be sure to specify the correct path for the mach kernel that is running on your target (assuming it is not the same as the kernel running on your debug host).

For versions after 10.1, you have two options. If your KEXT does not crash the computer when it loads, you can ask `kextload` to generate the symbols at load time by passing it the following options:

```
kextload -s symboldir mykext.kext
```

It will then write the symbols for your kernel extension and its dependencies into files within the directory you specified. Of course, this only works if your target doesn't crash at or shortly after load time.

Alternately, if you are debugging an existing panic, or if your KEXT can't be loaded without causing a panic, you can generate the debugging symbols on your debug host. You do this by typing:

```
kextload -n -s symboldir mykext.kext
```

If will then prompt you for the load address of the kernel extension and the addresses of all its dependencies. As mentioned previously, you can find the addresses with `kextstat` (or `kmodstat`) or by typing `showallkmods` inside `gdb`.

You should now have a file or files containing symbolic information that `gdb` can use to determine address–to–name mappings within the KEXT. To add the symbols from that KEXT, within `gdb` on your debug host, type the command

```
add-symbol-file mykext.sym
```

for each symbol file. You should now be able to see a human-readable representation of the addresses of functions, variables, and so on.

Using ddb for Kernel Debugging

When doing typical debugging, `gdb` is probably the best solution. However, there are times when `gdb` cannot be used or where `gdb` can easily run into problems. Some of these include

- drivers for built-in Ethernet hardware
- interrupt handlers (the hardware variety, not handler threads)
- early bootstrap before the network hardware is initialized

When gdb is not practical (or if you're curious), there is a second debug mechanism that can be compiled into Mac OS X. This mechanism is called ddb, and is similar to the kdb debugger in most BSD UNIX systems. It is not quite as easy to use as gdb, mainly because of the hardware needed to use it.

Unlike gdb (which uses Ethernet for communication with a kernel stub), ddb is built into the kernel itself, and interacts directly with the user over a serial line. Also unlike gdb, using ddb requires building a custom kernel using the DEBUG configuration. For more information on building this kernel, see "Building Your First Kernel" (page 156).

Note: ddb requires an actual *built-in hardware* serial line on the debug target. Neither PCI nor USB serial adapters will work. In order to work reliably for interrupt-level debugging, ddb controls the serial ports directly with a polled-mode driver without the use of the I/O Kit.

If your debug target does not have a factory serial port, third-party adapter boards may be available that replace your internal modem with a serial port. Since these devices use the built-in serial controller, they should work for ddb.

If your target computer has two serial ports, ddb uses the modem port (SCC port 0). However, if your target has only one serial port, that port is probably attached to port 1 of the SCC cell, which means that you have to change the default port if you want to use ddb. To use this port (SCC port 1), change the line:

```
const int console_unit=0;
```

in osfmk/ppc/serial_console.c to read:

```
const int console_unit=1;
```

and recompile the kernel.

Once you have a kernel with ddb support, it is relatively easy to use. First, you need to set up a terminal emulator program on your debug host. If your debug host is running Mac OS 9, you might use ZTerm, for example. For Mac OS X computers, or for computers running Linux or UNIX, minicom provides a good environment. Setting up these programs is beyond the scope of this document.

Important
Serial port settings for communicating with ddb must be 57600 8N1. Hardware handshaking may be on, but is not necessary.

Note: For targets whose Open Firmware uses the serial ports, remember that the baud rate for communicating with Open Firmware is 38400 and that hardware handshaking must be *off*.

Once you boot a kernel with ddb support, a panic will allow you to drop into the debugger, as will a call to PE_enter_debugger. If the DB_KDB flag is not set, you will have to press the D key on the keyboard to use ddb. Alternately, if both DB_KDB and DB_NMI are set, you should be able to drop into ddb by generating a nonmaskable interrupt (NMI). See "Setting Debug Flags in Open Firmware" (page 159) for more information on debug flags.

To generate a nonmaskable interrupt, hold down the command key while pressing the power key on your keyboard or press the interrupt button on your target computer. At this point, the system should hang, and you should see ddb output on the serial terminal. If you do not, check your configuration and verify that you have specified the correct serial port on both computers.

Commands and Syntax of ddb

The ddb debugger is much more gdb-like than previous versions, but it still has a syntax that is very much its own (shared only with other ddb and kdb debuggers). Because ddb is substantially different from what most developers are used to using, this section outlines the basic commands and syntax.

The commands in ddb are generally in this form:

```
command[/switch] address[,count]
```

The switches can be one of those shown in Table 18-2.

Table 18-2 Switch options in `ddb`

Switch	Description
`/A`	Print the location with line number if possible
`/I`	Display as instruction with possible alternate machine-dependent format
`/a`	Print the location being displayed
`/b`	Display or process by bytes
`/c`	Display low 8 bits as a character (nonprinting characters as octal) *or* count instructions while executing (depends on instruction)
`/d`	Display as signed decimal
`/h`	Display or process by half word (16 bits)
`/i`	Display as an instruction
`/l`	Display or process by long word (32 bits)
`/m`	Display as unsigned hex with character dump for each line
`/o`	Display in unsigned octal
`/p`	Print cumulative instruction count and call tree depth at each call or return statement
`/r`	Display in current radix, signed
`/s`	Display the null-terminated string at address (nonprinting as octal).
`/u`	Display in unsigned decimal *or* set breakpoint at a user space address (depending on command).
`/x`	Display in unsigned hex
`/z`	Display in signed hex

The ddb debugger has a rich command set that has grown over its lifetime. Its command set is similar to that of ddb and kdb on other BSD systems, and their manual pages provide a fairly good reference for the various commands. The command set for ddb includes the following commands:

`break[/u] addr`
: Set a breakpoint at the address specified by `addr`. Execution will stop when the breakpoint is reached. The `/u` switch means to set a breakpoint in user space.

`c` *or* `continue[/c]`
: Continue execution after reaching a breakpoint. The `/c` switch means to count instructions while executing.

`call`
: Call a function.

`cond`
: Set condition breakpoints. This command is not supported on PowerPC.

`cpu cpunum`
: Causes `ddb` to switch to run on a different CPU.

`d` *or* `delete [addr|#]`
: Delete a breakpoint. This takes a single argument that can be either an address or a breakpoint number.

`dk`
: Equivalent to running `kextstat` while the target computer is running. This lists loaded KEXTs, their load addresses, and various related information.

`dl vaddr`
: Dumps a range of memory starting from the address given. The parameter `vaddr` is a kernel virtual address. If `vaddr` is not specified, the last accessed address is used. See also `dr`, `dv`.

`dm`
: Displays mapping information for the last address accessed.

`dmacro name`
: Delete the macro called `name`. See `macro`.

`dp`
: Displays the currently active page table.

`dr addr`

Dumps a range of memory starting from the address given. The parameter `address` is a physical address. If `addr` is not specified, the last accessed address is used. See also `dl`, `dv`.

`ds`

Dumps save areas of all Mach tasks.

`dv [addr [vsid]]`

Dumps a range of memory starting from the address given. The parameter `addr` is a virtual address in the address space indicated by `vsid`. If `addr` is not specified, the last accessed address is used. Similarly, if `vsid` is not specified, the last `vsid` is used. See also `dl`, `dr`.

`dwatch addr`

Delete a watchpoint. See `watch`.

`dx`

Displays CPU registers.

`examine`

See `print`.

`gdb`

Switches to `gdb` mode, allowing `gdb` to attach to the computer.

`lt`

On PowerPC only: Dumps the PowerPC exception trace table.

`macro name command [ ; command .. ]`

Create a macro called `name` that executes the listed commands. You can show a macro with the command `show macro name` or delete it with `dmacro name`.

`match[/p]`

Stop at the matching return instruction. If the `/p` switch is not specified, summary information is printed only at the final return.

`print[/AIabcdhilmorsuxz] addr1 [addr2 ...]`

Print the values at the addresses given in the format specified by the switch. If no switch is given, the last used switch is assumed. Synonymous with `examine` and `x`. Note that some of the listed switches may work for `examine` and not for `print`.

`reboot`

Reboots the computer. Immediately. Without doing any file-system unmounts or other cleanup. Do not do this except after a panic.

`s` *or* `step`

Single step through instructions.

`search[/bhl] addr value [mask[,count]]`

Search memory for `value` starting at `addr`. If the value is not found, this command can wreak havoc. This command may take other formatting values in addition to those listed.

`set $name [=] expr`

Sets the value of the variable or register named by `name` to the value indicated by `expr`.

`show`

Display system data. For a list of information that can be shown, type the `show` command by itself. Some additional options are available for certain options, particularly `show all`. For those suboptions, type `show all` by itself.

`trace[/u]`

Prints a stack backtrace. If the `/u` flag is specified, the stack trace extends to user space if supported by architecture-dependent code.

`until[/p]`

Stop at the next call or return.

`w` *or* `write[/bhl] addr expr1 [expr2 ... ]`

Writes the value of `expr1` to the memory location stored at `addr` in increments of a byte, half word, or long word. If additional expressions are specified, they are written to consecutive bytes, half words, or long words.

`watch addr[,size]`

Sets a watchpoint on a particular address. Execution stops when the value stored at that address is modified. Watch points are not supported on PowerPC.

WARNING

Watching addresses in wired kernel memory may cause unrecoverable errors on i386.

`x`

Short for `examine`. See `print`.

`xb`

Examine backward. Execute the last examine command, but use the address previous to the last one used (jumping backward by increments of the last width displayed).

`xf`

Examine forward. Execute the last `examine` command, but use the address following the last one used (jumping by increments of the last width displayed).

The `ddb` debugger should seem relatively familiar to users of `gdb`, and its syntax was changed radically from its predecessor, `kdb`, to be more `gdb`-like. However, it is still sufficiently different that you should take some time to familiarize yourself with its use before attempting to debug something with it. It is far easier to use `ddb` on a system whose memory hasn't been scribbled upon by an errant DMA request, for example.

Document Revision History

Table A-1 describes the revisions to *Inside Mac OS X: Kernel Programming*.

Table A-1 Document revision history

Date	Notes
Feb, 2003	Minor update release. Added index and tweaked wording throughout. Fixed minor errata in debugging chapter. Added a few missing details in the security chapter and cleaned up the equations presented. Corrected a few very minor Mac OS X 10.2-specific details that weren't caught during the first revision.
Aug. 2002	Mac OS X 10.2 update release. Changed information on KEXT management, various small corrections (mainly wording improvements).
June 2002	Full web release to coincide with WWDC. Corrected a few minor errata from the previous release.
Jan. 2002	Initial partial web release.

APPENDIX A

Document Revision History

Bibliography

Apple Mac OS X Publications

The following Apple publications have information that could be of interest to you if you are programming in the kernel:

Hello Debugger Debugging a Device Driver with GDB (tutorial).

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

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

Inside Mac OS X: Accessing Hardware From Applications

Inside Mac OS X: I/O Kit Fundamentals

Inside Mac OS X: Making Hardware Accessible to Applications

Inside Mac OS X: Network Kernel Extensions

Inside Mac OS X: System Overview

Inside Mac OS X: UNIX Porting Guide

Inside Mac OS X: Writing I/O Kit Drivers

Packaging Your KEXT for Distribution and Installation (tutorial).

Bibliography

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 and 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 (paperback), ISBN 0-13-937699-2 (hardback).

BSD and UNIX Internals

Advanced Topics in UNIX: Processes, Files, and 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 Operating System. Marshall Kirk McKusick, et al. Addison-Wesley, 1996. ISBN 0-201-54979-4.

Bibliography

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

Linux Kernel Internals 2nd edition. Michael Beck, et al. Addison-Wesley, 1997. ISBN 0-201-33143-8.

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, Implementation for the Mach Microkernel. Dejan S. Milojicic. Vieweg Verlag, 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), now Silicomp:

Bibliography

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

Bibliography

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.

Bibliography

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 Eigth Edition for GDB version 5.0. Richard Stallman et al. Cygnus Support. http://developer.apple.com/techpubs/macosx/DeveloperTools/gdb/gdb/gdb_toc.html.

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

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/docs/manuals/gnupro/

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/

The Single UNIX Specification Version 2

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

Stackable File Systems

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

The USENIX Association; USENIX Proceedings

http://www.usenix.org

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

Security and Cryptography

Applied Cryptography: Protocols, Algorithms, and Source Code in C. Bruce Schneier. John Wiley & Sons, 1994. ISBN 0-471-59756-2.

comp.security newsgroup (news:comp.security).

comp.security.unix newsgroup (news:comp.security.unix).

Computer Security. Dieter Gollmann. John Wiley and Son Ltd, 1999. ISBN 0-471-97844-2.

Foundations of Cryptography. Oded Goldreich. Cambridge University Press, 2001. ISBN 0-521-79172-3.

Secrets and Lies: Digital Security in a Networked World. Bruce Schneier. John Wiley & Sons, 2000. ISBN 0-471-25311-1.

Glossary

abstraction (v) The process of separating the interface to some functionality from the underlying implementation in such a way that the implementation can be changed without changing the way that piece of code is used. (n) The API (interface) for some piece of functionality that has been separated in this way.

address space The virtual address ranges available to a given task (note: the task may be the kernel). In Mac OS X, processes do not share the same address space. The address spaces of multiple processes can, however, point to the same physical address ranges. This is referred to as shared memory.

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 convention) by which an application program accesses a service. This service may be provided by the operating system, by libraries, or by other parts of the application.

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.

bundle 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 presented by the Finder as if they were 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. Carbon applications can run in Mac OS X, Mac OS 9, and all versions of Mac OS 8 later than Mac OS 8.1 (with appropriate libraries).

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.

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 Essentially a wait queue with additional locking semantics. When a thread sleeps waiting for some event to occur, it releases a related lock so that another thread can cause that event to occur. When the second thread posts the event, the first thread wakes up, and, depending on the condition variable semantics used, either takes the lock immediately or begins waiting for the lock to become available.

console (1) A text-based login environment that also displays system log messages, kernel panics, and other information. (2) A special window in Mac OS X that displays messages that would be printed to the text console if the GUI were not in use. This window also displays output written to the standard error and standard output streams by applications launched from the Finder. (3) An application by the same name that displays the console window.

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 9 is a cooperative multitasking environment. See also **preemptive multitasking**.

copy-on-write A delayed copy optimization used in Mach. The object to be copied is marked temporarily read-only. When a thread attempts to write to any page in that object, a trap occurs, and the kernel copies only the page or pages that are actually being modified. See also **thread**.

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. They may also be spawned by other daemons.

Darwin The core of Mac OS X, Darwin is 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.

default pager In Mach, one of the 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 requiring the host processor to move the data itself. This reduces processor overhead for I/O operations and may reduce contention on the processor bus.

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 family of high-speed local area network technologies in common use. Some common variants include 802.3 and 802.11 (Airport).

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 a backing store. External pagers are clients of Mach's EMMI. The pager API is currently not exported to user space. The built-in pagers in Mac OS X are the default pager, the device pager, and the vnode pager. See also **EMMI (External Memory Management Interface)**.

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 Executable 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 (1) 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. (2) In BSD, `fork` is a system call that creates a new process.

framework A bundle containing a dynamic shared library and associated resources, including image files, header files, and documentation. Frameworks are often used to provide an abstraction for manipulating device driver families from applications.

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.

HFS (hierarchical file system) The Mac OS Standard file system 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+ The Mac OS Extended file system format. This file system 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 larger disks.

host (1) The computer that is running (is host to) a particular program or service. The term is usually used to refer to a computer on a network. (2) In debugging, the computer that is running the debugger itself. In this context, the target is the machine running the application, kernel, or driver being debugged.

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.

IDE (integrated development environment) An application or set of tools that allows a programmer to write, compile, edit, and in some cases 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 pages in the parent process's address space. A memory page can be inherited as copy-on-write, shared, or not at all.

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

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**.

interrupt service thread A thread running in kernel space for handling I/O that is triggered by an interrupt, but does not run in an interrupt context. Also called an I/O service thread.

I/O (input/output) The exchange of data between two parts of a computer system, usually between system memory and 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**.

I/O service thread See **interrupt service thread**.

IPC (interprocess communication) The transfer of information between processes or between the kernel and a process.

IPL (interrupt priority level) A means of basic synchronization on uniprocessor systems in traditional BSD systems, set using the `spl` macro. Interrupts with lower priority than the current IPL will not be acted upon until the IPL is lowered. In many parts of the kernel, changing the IPL in Mac OS X is not useful as a means of synchronization. New use of `spl` macros is discouraged. See also **spl (set priority level)**.

KDP The kernel shim used for communication with a remote debugger (`gdb`).

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 crash An unrecoverable system failure in the kernel caused by an illegal instruction, memory access exception, or other failure rather than explicitly triggered as in a panic. See also **panic**.

kernel extension See **KEXT (kernel extension)**.

kernel mode See **supervisor mode**.

kernel panic See **panic**.

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

KEXT (kernel extension) A bundle that extends 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.

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

key signing In public key cryptography, to (electronically) state your trust that a public key really belongs to the person who claims to own it, and potentially that the person who claims to own it really is who he or she claims to be.

KMOD (kernel module) See KEXT binary.

lock A basic means of synchronizing multiple threads. Generally only one thread can "hold" a lock at any given time. While a thread is holding the lock, any other thread that tries to take it will wait, either by blocking or by spinning, depending on the nature of the lock. Some lock variants such as read-write locks allow multiple threads to hold a single lock under certain conditions.

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

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.

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 the memory, file, or other storage that backs a VM object. 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, usually through the use of separate address spaces.

message A unit of data sent by one task or thread that is guaranteed to be delivered atomically to another task or thread. In Mach, a message consists of a header and a variable-length body. Some 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. Mac OS X is a hybrid between microkernel and monolithic kernel architectures. See also **monolithic kernel**.

MIG (Mach interface generator) (1) A family of software that generates and supports the use of a procedure call interface to Mach's system of interprocess communication. (2) The interface description language supported by MIG.

monolithic kernel A kernel architecture in which all pieces of the kernel are closely intertwined. A monolithic kernel provides substantial performance improvements. It is difficult to evolve the individual components independently, however. 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, usually on different networks. 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 The concurrent execution of multiple programs. Mac OS X uses preemptive multitasking. Mac OS 9 uses cooperative multitasking.

mutex See **mutex lock (mutual exclusion lock)**.

mutex lock (mutual exclusion lock) A type of lock characterized by putting waiting threads to sleep until the lock is available.

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

name port In Mach, access to the name port allows non-priviledged operations against an object (for example, obtaining information about the object). In effect, it provides a name

for the object without providing any significant access to the object. See also **port**; **control port**.

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

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

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 communicate with each other.

NFS (network file system) A commonly used file server protocol often found in UNIX and UNIX-based environments.

NKE (network kernel extension) A type of KEXT that provides a way to extend and modify the networking infrastructure of Mac OS X dynamically without recompiling or relinking the kernel.

NMI (nonmaskable interrupt) An interrupt produced by a particular keyboard sequence or button that cannot be blocked in software. It can be used to interrupt a hung system, for example to drop into a debugger.

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

notify port A special Mach port that is part of a task. A task's notify port receives messages from the kernel advising the task of changes in port access rights and of the status of messages it has sent.

nub An I/O Kit object that represents a point of connection for 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 (random-access memory)**.

object (1) A collection of data. (2) In Mach, a collection of data, with permissions and ownership. (3) In object-oriented programming, an instance of a class.

OHCI (Open Host Controller Interface) The register-level standards that are used by most USB and Firewire controller chips.

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

Open Transport A communications architecture for implementing network protocols and other communication features on computers running classic Mac OS. Open

Transport provides a set of programming interfaces that supports, among other things, both the AppleTalk and TCP/IP protocols.

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

packet An individual piece of information sent on a network.

page (n) (1) The largest block of virtual address space for which the underlying physical address space is guaranteed contiguous—in other words, the unit of mapping between virtual and physical addresses. (2) logical page size: The minimum unit of information that an anonymous pager transfers between system memory and the backing store. (3) physical page size: The unit of information treated as a unit by a hardware MMU. The logical page size must be at least as large as the physical page size for hardware-based memory protection to be possible. (v) To move data between memory and a backing store.

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 explicitly triggered by the kernel with a call to `panic`. See also **kernel crash**.

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

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.

pmap Part of Mach VM that provides an abstract way to set and fetch virtual to physical mappings from hardware. The pmap system is the machine-dependent layer of the VM system.

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 receiving service for an incoming packet, or to specify the sender of an outgoing packet.

port name In Mach, an integer index into a port namespace; a port right is specified with respect to 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()`.

POSIX (Portable Operating System Interface) A standard that defines a set of operating-system services. It is supported by ISO/IEC, IEEE, and The Open Group.

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

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**.

priority In scheduling, a number 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's resources include an 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 (PID), A number that uniquely identifies a process. Also called a process ID.

programmed I/O I/O in which the CPU accomplishes data transfer with explicit load and store instructions to device registers, rather than DMA, and without the use of interrupts. This data transfer is often done in a byte-by-byte, or word-by-word fashion. Also known as direct or polled I/O. See also **DMA (direct memory access)**.

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**.

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).

pthreads The POSIX threads implementation. See also **POSIX (Portable Operating System Interface)**; **thread**.

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 from or write to.

real-time performance Performance characterized by guaranteed worst-case response times. 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**.

remote procedure call See **RPC (remote procedure call)**.

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

ROM (read-only memory) Memory that cannot be written to.

root (1) An administrative account with special privileges. For example, only the root account can load kernel extensions.(2) In graph theory, the base of a tree. (3) root

directory: The base of a file system tree. (4) root file system: The primary file system off which a computer boots, so named because it includes the root node of the file system tree.

routine In Mach, a remote procedure call that returns a value. This can be used for synchronous or asynchronous operations. See also **simpleroutine**.

RPC (remote procedure call) An interface to IPC that appears (to the caller) as an ordinary function call. In Mach, RPCs are implemented using MIG-generated interface libraries and Mach messages.

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

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

SCSI (Small Computer Systems Interface) A standard communications protocol used for connecting devices such as disk drives to computers. Also, a family of physical bus designs and connectors commonly used to carry SCSI communication.

semaphore Similar to a lock, except that a finite number of threads can be holding a semaphore at the same time. See also **lock**.

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**.

session key In cryptography, a temporary key that is only used for one message, one connection session, or similar. Session keys are generally treated as shared secrets, and are frequently exchanged over a channel encrypted using public key cryptography.

shadow object In Mach VM, a memory object that holds modified pages that originally belonged to another memory object. This is used when an object that was duplicated in a copy-on-write fashion is modified. If a page is not found in this shadow object, the original object is referenced.

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

simpleroutine In Mach, a remote procedure call that does not return a value, and has no `out` or `inout` parameters.. This can be used for asynchronous operations. See also **routine**.

SMP (symmetric multiprocessing) A system architecture in which two or more processors are managed by one kernel, share the same memory, have equal access to I/O devices, and in which any task, including kernel tasks, can run on any processor.

spinlock Any of a family of lock types characterized by continuously polling to see if a lock is available, rather than putting the waiting thread to sleep.

spin/sleep lock Any of a family of lock types characterized by some combination of the behaviors of spinlocks and mutex (sleep) locks.

spl (set priority level) A macro that sets the current IPL. Interrupts with lower priority than the current IPL will not be acted upon until the IPL is lowered. The `spl` macros have no effect in many parts of Mac OS X, so their use is discouraged as a means of synchronization in new programming except when modifying code thtat already uses `spl` macros.. See also **IPL (interrupt priority level)**.

socket (1) In a user process, a file descriptor that has been allocated using `socket(2)`. (2) In the kernel, the data structure allocated when the kernel's implementation of the `socket(2)` call is made. (3) In AppleTalk protocols, a socket serves the same purpose as a port in IP transport protocols.

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

submap A collection of mappings in the VM system that is shared among multiple Mach tasks.

supervisor mode Also known as kernel mode, the processor mode in which certain privileged instructions can be executed, including those related to page table management, cache management, clock setting, and so on.

symmetric multiprocessing See **SMP (symmetric multiprocessing)**.

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

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 The unit of program execution. 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 code Code that can be executed safely by multiple 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.

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

UFS (UNIX file system) An industry standard file system format used in UNIX and similar operating systems such as BSD. UFS in Mac OS X is a derivative of 4.4BSD UFS.

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.

UPL (universal page list) A data structure used when communicating with the virtual memory system. UPLs can be used to change the behavior of pages with respect to caching, permissions, mapping, and so on.

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.

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.

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 into the kernel.

virtual address An address as viewed from the perspective of an application. Each task has its own range of virtual addresses, beginning at address zero. The Mach VM system makes the CPU hardware map these addresses onto physical memory. See also **physical address**.

virtual memory A system in which addresses as seen by software are not the same as addresses seen by the hardware. This provides support for memory protection, reduces the need for code relocatability, and allows the operating system to provide the illusion to each application that it has resources much larger than those that could actually be backed by RAM.

VM See **virtual memory**.

vnode An in-memory data structure containing information about a file.

vnode pager In Mach, one of the 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 KEXT that waits repeatedly for incoming events and dispatches them.

XML (Extensible Markup Language) A 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.

Index

INDEX