Kernel and Operating System Services

Preliminary

Developer Press

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

Simultaneously published in the United States and Canada.

Contents

Figures, Tables, and Listings

Preface

This book provides the conceptual background to understand how and when to use services provided by the Mac OS kernel and operating system. The kernel controls and coordinates access to the hardware, which includes the processor, memory, and I/O devices. All services are built on top of kernel services. Operating system services, as described in this manual, are services built directly from kernel services and are not part of a larger service, such as the file system and Toolbox, which also use kernel services and may use operating system services as well.

If you are an application developer or an OpenDoc part developer, you should read this manual to gain an understanding of the services provided by the Mac OS. If you write only single-task applications or parts (such as those created for System 7), you probably won't need to use most of the services described herein. However, this manual explains how you can write more efficient yet simpler applications and parts using services provided by the kernel and operating system.

If you are writing extensions, device drivers, or other system software, you will need to use kernel and operating system services. Read this chapter to understand the services that are provided and to gain an appreciation of the choices you'll need to make when implementing your software.

Book Organization

IMPORTANT

This book is a work in progress. Its contents are subject to change without notice. Your comments are welcome. ▲

This book is divided into seven chapters:

- Chapter 1, "Introduction to Kernel and Operating System Services," introduces the fundamental concepts that you need to make decisions about how to implement software. It also identifies the services described in the rest of the manual.

- Chapter 2, “Software Structure,” shows how you can package your software for execution and how software is scheduled for execution.
- Chapter 3, “Memory Management,” identifies the kinds of memory that you can use and how memory is addressed, accessed, and shared.
- Chapter 4, “Synchronization Services,” describes the services available to synchronize access to resources, such as memory data structures.
- Chapter 5, “Messaging Service,” describes the primary kernel service for interprocess and intraprocess communication.
- Chapter 6, “Other Services,” describes other kernel and operating system services, including the system registry, timing services, notification services and interspace block copy.
- Chapter 7, “System 7 Compatibility,” briefly identifies issues that affect the compatibility of System 7 applications.

Conventions Used in This Book

This book uses various conventions to present certain types of information.

Special Fonts

All code listings, reserved words, and the names of data structures, constants, fields, parameters, and functions are shown in a monospaced font (`this is monospaced`).

When new terms are introduced, they are in **boldface.** These terms are also defined in the glossary.

Types of Notes

There are several types of notes used in this book.

Note
A note like this contains information that is interesting but possibly not essential to an understanding of the main text. ◆

IMPORTANT
A note like this contains information that is especially important. ▲

Numerical Formats

Hexadecimal numbers are shown in this format: #x0008.

The numerical values of constants are shown in decimal format, unless the constants are flag or mask elements with bitwise interpretations, in which case they are shown in hexadecimal format.

CHAPTER 1

Introduction to Kernel and Operating System Services

Contents

This chapter introduces the kernel and operating system services. You should read this chapter if you need to obtain an overview of the services provided by the kernel and operating system. These services are described more fully in the chapters that follow.

The following sections briefly identify the major services provided by the kernel and operating system. These services include

- runtime and memory management services
- synchronization services
- interprocess communications services
- other services

Runtime Services and Memory Management

Runtime and memory management services include the mapping of processes into address spaces, code sharing, and the execution of tasks and other software. The organization of an address space and memory management issues are discussed in the chapter "Memory Management," beginning on page 43. The following sections briefly introduce two important runtime concepts:

- execution environments
- scheduling

Execution Environments

Execution environments specify the rules under which software can execute. For example, execution environments control the kinds of memory access allowed and the kernel services that can be executed. The execution environments are

- task (user mode and supervisor mode) environment
- hardware interrupt environment
- secondary interrupt environment

Of these environments, only user mode software can execute in the task environment; only supervisor mode (privileged) software can execute in all environments. For more information about execution environments, see the section "Execution Environments," beginning on page 36.

Scheduling

The kernel uses a preemptive multitask scheduling algorithm, in which the highest priority task runs until it blocks, until its time-slice is used up, or until a higher priority task becomes unblocked. For example, it might block so that an I/O operation can complete or because it is waiting for an event to occur. When a task is blocked, the next highest priority task is allowed to run.

Special rules apply to primary tasks in the cooperative process address space. While they obey the same rules as other tasks, the Process Manager ensures that only one primary task is eligible for execution; others are blocked until the current one relinquishes its eligibility.

For more information about scheduling, see the section "Scheduling Algorithm," beginning on page 38.

Synchronization Services

Synchronization services provide serialized access to shared resources. Often a shared resource is a data structure in memory, such as a data structure that can be updated by two tasks in the same address space or a data structure in shared memory that can be updated by software in different address spaces. By serializing access to this data, operations that need to be performed on the data atomically, that is, from start to finish as a single undivided operation, can be performed.

The major kinds of synchronization services provided by the kernel and operating system are

- Synchronization primitives and locking provide processor-supported atomic instructions and locks for implementing critical sections.
- Event groups let you wait for a condition to occur so that execution of a critical section will not start until the condition is satisfied.

- Kernel queues also let you wait for a condition to occur; however, they maintain more information about changes in condition and can also be used for explicit but limited communication.
- Interrupts can also be used to synchronize operations.

These services are described in the chapter "Synchronization Services," beginning on page 63.

Interprocess Communication Services

Interprocess communications services allow data to be transferred between different address spaces and between different execution environments. Some of these services are built directly into the kernel, while others are considered to be application services. All the interprocess communications services that are available are introduced in this section; however, application services such as Apple events, which are used by most applications to communicate between processes, are discussed in other *Inside Macintosh* books.

As a general principle, you should never expose the details of the communication mechanism or its protocol and you should try to minimize the volume of the interprocess communication.

In a client-server implementation, for example, you most likely will package the communication between client and server in a subroutine (function) library. There are advantages to using a subroutine library:

- The library hides the actual communication mechanism and protocol being used. In the future, you may decide to change the communications mechanism or you may decide to change the sequence or packaging of requests to or responses from the server.
- The library protects the client from changes in the underlying architecture; for example, you may find a way to reduce interprocess communication by shifting some of the software being executed on the server to the client. The code being moved can be treated as simply an implementation detail of a function in the library as opposed to a change to the client software itself.

The following communications services are supported by the operating system:

- Apple events
- messaging

- kernel queues
- system notification
- shared memory
- interspace block copy

Note
High-level events can also be used, but they are provided only for compatibility. For further information, see "High-Level Events," beginning on page 101. ◆

Transfer of data requires synchronization to succeed; for example, you can't allow data to be read while it is potentially inconsistent, such as when it is being updated. Therefore, most interprocess communications services either guarantee synchronization or support protocols that can be used to implement synchronization.

Apple Events

Apple events are the primary communications service for interapplication communication (IAC). Apple events are especially useful for IAC because the semantics of the data can be represented at a high level; for example, you can specify the paragraphs in a word-processing document or cells in a spreadsheet as the target of an Apple event.

You can use Apple events to build complex data structures and handle dynamic typing. The data itself, along with the semantics for its interpretation, can be flattened into a structure that can be transported with a physical communications service that supports variable size data. This structure is unflattened automatically at its destination. For more information about Apple events and their data model, see *Inside Macintosh: Interapplication Communication*.

Messaging

The messaging service provides a transaction-based mechanism for exchanging data between software in the same or different address spaces. The transaction consists of a message that is sent by the sender and received and replied to by the receiver. The contents of the message are not interpreted in any way by the

operating system. For more information about the messaging service, see "About the Messaging Service," beginning on page 81.

Kernel Queue Messages

Kernel queues are primarily used for synchronization; however, kernel queues can be used as a fast communications service (generally faster than messaging) if you can package the message in three words or less. For further information about kernel queues and the typical form of kernel queue messages, see the section "Kernel Queues," beginning on page 73.

Shared Memory

Global memory can be shared by tasks in different address spaces. The operating system supports other mechanisms as well. They provide more protection than using global memory. For more information about shared memory used for interprocess communication, see the section "Shared Data," beginning on page 56.

System Notification

System notification is an operating system service that allows a piece of software to broadcast a notification about a change in the state of the system. A broadcast is a notification that is sent to a potentially wide audience. The audience is software that can then respond to the notification. For more information about system notification, see "System Notification," beginning on page 96.

Interspace Block Copy

Interspace block copy is a service that allows the contents of memory to be copied between address spaces. It does not provide synchronization. For more information about the interspace block copy function, see "Interspace Block Copy" on page 97.

Other Services

Other services include

- the system registry, which is used to look up well-known names. For more information, see the section “System Registry” on page 93.
- timing services, which allow you to precisely measure elapsed time. For more information about timing services, see “Timing Services” on page 94.
- notification services, which include system notification as described above and three techniques for asynchronous notification. For more information about notification services, see “Notification Services” on page 95.

CHAPTER 2

Software Structure

Contents

This chapter describes the possible ways to structure your software, whether an application, part, extension, or driver. Closely related to software structure are execution environments. Execution environments control the addressability of memory and, indirectly, they control memory access rights and affect synchronization. Finally, the kernel handles scheduling for all tasks. The rest of this chapter expands on these topics.

Software Structure

Most software, especially applications, is structured as tasks in processes. In addition to using processes and tasks, software can also be structured to use accept functions and interrupt handlers. An accept function is a special-purpose subroutine that replaces the need for a task that receives messages and replies to them. An interrupt handler is a subroutine that responds to an asynchronous event, such as a change in the hardware.

The following sections describe processes and the ways that software can be structured as

- tasks
- interrupt handlers
- accept functions
- exception handlers

Finally, issues related to performance and software structure are discussed.

About Processes

A **process** is a collection of one or more tasks and other resources associated with an address space. Resources include data in files and data in resources.

When a process is created, an address space is associated with the process. Once the process has been created, its address space cannot change. When you create tasks, you must specify the process to which they are associated. Once a task has been created, its process cannot be changed. (The same code, however, can be executed by tasks in different processes.) The process, therefore, controls the logical addresses that can be addressed by these tasks. For information

about tasks, see "About Tasks," beginning on page 24. For information about address spaces, see "Address Spaces," beginning on page 43.

In general, each process should be in a separate address space, as this allows resources in one process to be protected from tasks in another process. An exception is that if a task needs to use cooperative services, the task's process must be associated with the cooperative process address space. **Cooperative services** are non-reentrant operating system services, such as most parts of the Toolbox, the classic Memory Manager, and other services commonly associated with applications. A **cooperative process** is a Process Manager process whose (primary) task can use cooperative services. For information about the Process Manager, see *Inside Macintosh: Processes*. The **cooperative process address space** is an address space provided by the operating system.

Typically, a cooperative process is an application with a user interface component. They are called cooperative processes because their primary tasks cooperate to ensure that they periodically relinquish the processor to tasks in other cooperative processes—this is not automatic—and that they do not inadvertently affect memory associated with the tasks of other processes in the cooperative process address space.

The key issue in determining the kind of process to use is whether or not your software uses cooperative services. Ask the question, "Does the software require cooperative services such as the Toolbox?" If the answer is yes, you need to use a cooperative process. If the answer is no, you can use a process in a separate address space because it provides memory protection and is the most suitable structure for future enhancements to the operating system.

About Tasks

A task represents code in a state of execution. Typically, this code is shared so that a task does not actually "own" the code—instead, the task maintains state information, such as local variables, the next instruction to execute and register values.

There are several ways to identify the kind of task:

- A task can be either a primary task or a secondary task. A **primary task** is a task associated with a cooperative process and is created by the Process Manager when the application is launched. Primary tasks are the only tasks allowed to use cooperative services. Other tasks created from within cooperative processes are **secondary tasks;** they cannot use cooperative services.

- A task can be either a user mode task or a supervisor mode task. The kind of task determines which instructions can be executed and which addresses in the address space can be referenced. A **user mode task** is one that cannot execute privileged instructions. A **supervisor mode task**, also called a **privileged task,** can execute any instruction and can have different, potentially greater, memory access rights. All tasks in the same process run in the same mode.
- A task can be a **parent task**, which owns one or more **child tasks**, or a task can be an **orphan task**, which is a task that has no parent. These hierarchial relationships are useful when specifying actions that affect multiple tasks.

These ways of identifying tasks are not mutually exclusive. For example, a secondary task may or may not be a privileged task; it could be a parent, child, or orphan task. The following sections describe each of these kinds of tasks further.

Primary and Secondary Tasks

Primary and secondary tasks are relevant only for cooperative processes—these processes are typically applications and have a user interface component. When an application is launched, the Process Manager creates a primary task. This task can use either reentrant or non-reentrant services, including cooperative services. You can create other tasks, called secondary tasks, from your application to achieve increased efficiency overall by dividing the work to be performed into multiple tasks. Secondary tasks cannot use cooperative services.

Note
Tasks associated with processes that are not cooperative processes have the same restriction as secondary tasks associated with a cooperative process. None of these tasks can use cooperative services. ◆

If your application must perform computationally intensive execution, you should use a secondary task. For example, you can improve the responsiveness and productivity of your application by having the primary task perform user interface tasks and using secondary tasks to process data and perform time-consuming I/O and computation-intensive operations.

Figure 2-1 shows an example of primary and secondary tasks using system services.

Figure 2-1 Primary and secondary tasks within an application

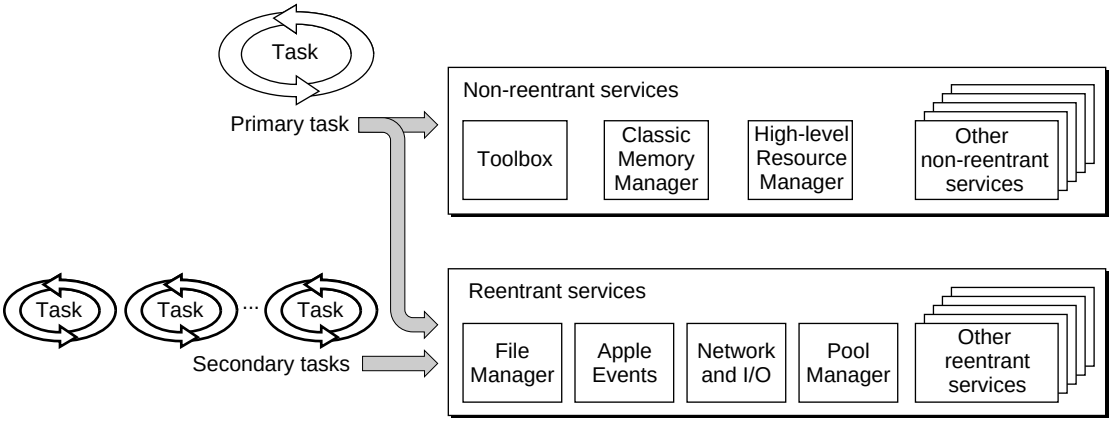


Table 2-1 shows the services that can be used with any task because they are reentrant. All other services are cooperative services and can be used only by primary tasks.

Table 2-1 Reentrant services

Reentrant service	Reentrant service
Modern file system APIs	Messaging
Area (memory) services	Virtual memory backing provider
Kernel services	Pool Manager
Synchronization services	Driver services library
Tasking services	System notification
System registry	International text object (parts of) and string comparison services
Apple events	Open Transport services
I/O families (SCSI, ADB, etc.)	Code Fragment Manager
Standard C library	

User Mode and Supervisor Mode Tasks

User mode and supervisor mode control instruction execution privileges (which specify the instructions that can be executed) and memory access rights (which determine the ability of a task to access different locations in its process's address space). A task can be either a user mode task or a supervisor mode task.

Supervisor mode tasks can execute any instruction. User mode tasks cannot execute privileged processor instructions or invoke services that handle interrupts or set up accept functions. For more information about instruction privileges, see the section "Performance and Software Structure," beginning on page 34.

Memory access rights are specified when ranges of logical addresses are defined within the address space. These ranges are called areas. For more information about areas, see the section "Areas," beginning on page 45.

A user mode task has read/write access to private memory within an address space and read-only access to most global memory. A user mode task can be granted read/write access to areas in global memory, as needed. Some global memory is completely inaccessible to user mode tasks; for example, I/O device control registers. A supervisor mode task can only address locations in global memory.

All applications should be written as user mode tasks. Primary tasks in cooperative address spaces are always user mode tasks. Supervisor mode tasks should only be used when access to hardware-related memory locations are required.

Parent, Child, and Orphan Tasks

When you create a task from another task, by default the newly created task becomes a child of the task that created it. You can use this relationship to specify the scope while changing task priorities or terminating a task. For example, you can use the scope to terminate tasks in a single call—you could specify that the parent task and child tasks be terminated, or just the parent task.

If you create a task from another task but wish the newly created task to be in a hierarchy by itself, you can specify the newly created task as an orphan task. If the orphan task creates another task in the default manner, it becomes a parent task.

A child task executes in the same mode as its parent. A user mode task can only create other user mode tasks and a supervisor mode task can only create other supervisor mode tasks.

About Accept Functions

Accept functions are privileged code whose purpose is to receive messages. An accept function serves the same purpose as a task-based receiver; however, it is actually a subroutine that is automatically invoked when a message is received. A message is part of an interprocess communication mechanism that allows data to be transferred from a sender to a receiver (with a reply sent back), whether or not the sender and receiver are in the same address space. A sender is a task (either user mode or a supervisor mode) or a software interrupt handler. For information about messages and senders and receivers, see "About the Messaging Service," beginning on page 81. For information about software interrupt handlers, see "Software Interrupt Handlers," beginning on page 29.

An accept function executes in the sender's context. An accept function always runs in supervisor mode, even if the sender is running in user mode. Because an accept function is implemented as a subroutine call, no task switch is required. The sending task cannot continue its execution while the accept function executes.

There are several advantages to using accept functions instead of tasks to handle receipt of messages:

- The receiver of a message can access the sender's memory because the accept function is executing in the sender's address space. Thus the data being transferred can be passed by reference rather than being copied.
- Execution of an accept function is fast. By being implemented as a subroutine associated with a task rather than being implemented as a task itself, an accept function can be executed without a context switch from the sending task to the receiver. It is slower than making a subroutine call; however, it is much faster than sending a message to another task.
- An accept function requires resources (stack space) only when it is actually running. To receive messages with tasks, you would need to have a task created and waiting, and it would be consuming resources while waiting.
- An accept function can be executed concurrently, with one execution in progress per sender. To achieve the same level of concurrency with tasks,

you would need to create a task to handle each message as soon as it was received or you would need to have a task ready to receive each message.

About Interrupt Handlers

Interrupt handlers respond to hardware and software events. When an interrupt occurs, task execution is suspended on the processor and execution of the interrupt handler begins. Task execution does not resume on that processor until the interrupt has been handled.

You can implement an interrupt handler for three kinds of interrupts:

- software interrupts are interrupts that are sent or signaled by software. They are the only kind of interrupt that can be handled within an application.
- hardware interrupts are interrupts that are caused by a signal from an external device.
- secondary interrupts are interrupts that are sent or signaled from a hardware interrupt handler or a supervisor mode task.

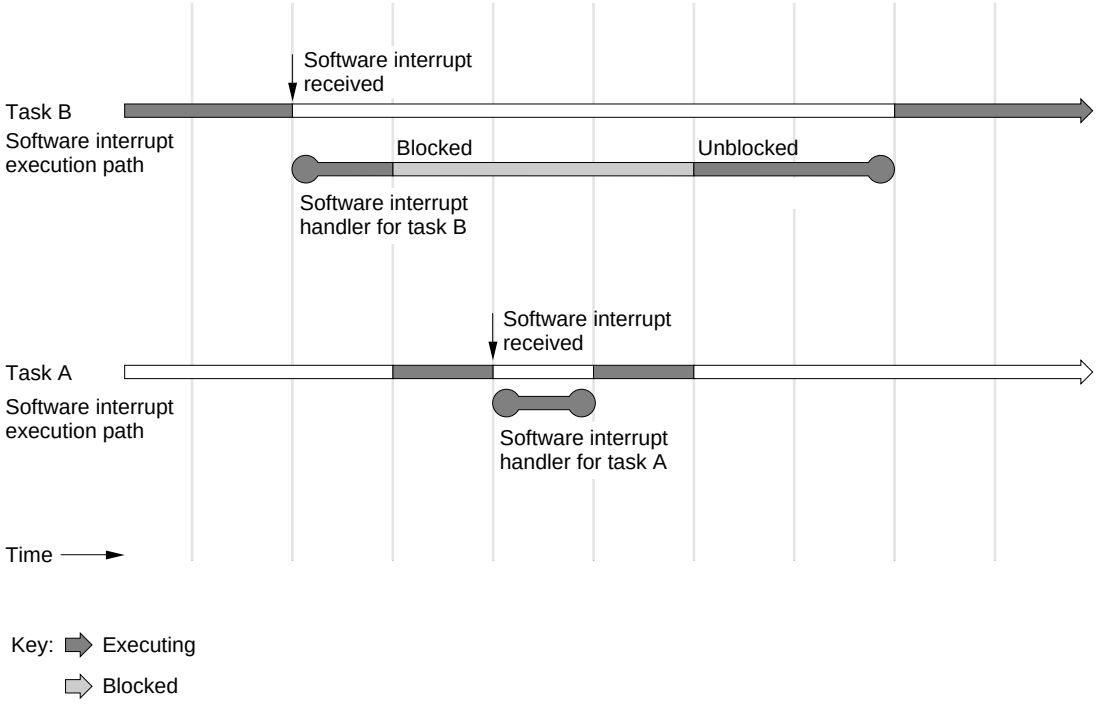
The following sections discuss handlers for these kinds of interrupts.

Software Interrupt Handlers

A **software interrupt** is a signal or event that is sent to a task. A **software interrupt handler** can respond to this kind of interrupt. The sender of an interrupt can be software in the same process (and address space) or software in a different process and, most likely, in a different address space as the task to which the interrupt is sent. The kernel saves the interrupted task's state, executes the specified interrupt handler, then restores the task's state and allows it to resume execution.

A task responds to multiple software interrupts sequentially. If several interrupts are received, they are guaranteed to be executed in the order in which they are sent. Figure 2-2 shows the relationship between task execution and software interrupt execution.

Figure 2-2 Effect of task and software interrupt execution



Note
Figure 2-2 implies that the tasks are running on a single processor. When implementing your software, you should always assume that tasks can execute concurrently, as if each task were executing on its own processor. ◆

Software interrupts are delivered to a task, even if the task is blocked. If a software interrupt handler blocks, no further interrupts are delivered to its task; the interrupts are queued for delivery.

Software interrupt handlers do not change execution priority. A software interrupt handler is not run until the interrupted task is scheduled to execute. The interrupt handler runs at the same priority as the task.

Interrupt handlers can be temporary, meaning that they only execute (at most) once, or they can be permanent, meaning that they are set up once and can respond to an interrupt each time it is sent.

If a permanent interrupt handler cannot handle its interrupt before receiving the next one, it executes only once, regardless of the number of times the interrupt occurred. After the handler executes, the next interrupt causes the handler to be scheduled again.

The primary reason to use software interrupts is to handle asynchronous events within the same task. You might consider a permanent software interrupt if you don't care whether each interrupt is handled.

For other uses of software interrupts, you should carefully consider the benefit versus complexity of handling asynchronous events within a single task. It may be easier to use several tasks, each handling a single kind of event. You may also find it easier to use kernel queues to synchronize events. For example, instead of issuing an asynchronous I/O request and handling its completion as a software interrupt, you should consider using a separate task to handle just the I/O or using a kernel queue to notify your task of the completion. For information about kernel queues, see the section "Kernel Queues," beginning on page 73.

Hardware Interrupt Handlers

A **hardware interrupt** is a signal from an external device. A **hardware interrupt handler** is privileged code that responds to hardware interrupts. On processors with only one hardware interrupt priority level, such as the Power Macintosh, a hardware interrupt handler cannot be interrupted.

Note
In a multiprocessor system, only one hardware interrupt handler can execute at a time, regardless of the number of processors. ◆

A hardware interrupt causes all software execution on the interrupted processor to stop until the handler executes. Interrupts are effectively disabled during a hardware interrupt handler's execution because nothing else can execute and the handler cannot be interrupted. Thus a hardware interrupt handler should take only minimal actions, such as

- resetting the interrupt condition on the device

- notifying some other piece of software that an interrupt occurred. (This software is typically a supervisor mode task or a secondary interrupt handler.)

The following additional points are relevant to implementors of hardware interrupt handlers:

- A hardware interrupt handler cannot be allowed to block or cause a page fault; thus all resources that the handler uses must be available, and the memory needed for code and data should be resident and locked down.
- Hardware interrupt handlers are given a stack, called the interrupt stack, to run on.

Secondary Interrupt Handlers

A **secondary interrupt** is a signal from a hardware interrupt handler or a supervisor mode task. A **secondary interrupt handler** is privileged code that can be interrupted only by hardware interrupts. All task execution remains blocked on that processor until a secondary interrupt handler finishes its execution on that processor.

Note
In a multiprocessor system, only one secondary interrupt handler can execute at a time, regardless of the number of processors. ◆

Secondary interrupt handlers associated with hardware interrupt handlers are queued, then executed in a first-in, first-out order. Secondary interrupt handlers for privileged tasks are called and executed immediately. All secondary interrupt handlers must complete execution before task execution can resume.

You might need to use a secondary interrupt handler

- when work started by a hardware interrupt handler is so time consuming that the work cannot be completed before allowing another hardware interrupt to occur
- when atomic instructions are not sufficient for synchronizing a secondary interrupt handler scheduled by a hardware interrupt handler with a supervisor mode task

In the first case, you must exit the hardware interrupt handler as soon as possible so that additional hardware interrupts are not lost. Secondary

interrupts could be used for real-time processing before task execution is allowed to resume.

In the second case, a hardware interrupt handler could queue a secondary interrupt handler and a secondary interrupt handler could be called from a task as soon as its execution resumes. Because secondary interrupt handlers are guaranteed to be executed serially, they cannot be interrupted while in the process of changing shared data. For an example of how this works, see the section "Interrupts and Synchronization," beginning on page 74.

About Exception Handlers

The microprocessor detects **exceptions**—that is, errors or other special conditions such as addressing errors, arithmetic overflows, and illegal instructions—in the course of program execution. When one of these exceptions occurs, the kernel tries to call an **exception handler**, whose job it is to handle the exception in the most graceful way possible. The handler performs its action, then the kernel resumes execution from where the exception occurred or transfers control as indicated by the exception handler.

Note
These exception handlers are not the same as the high-level exception mechanism available in the C++ language. ◆

Each task, accept function, secondary interrupt handler, and hardware interrupt handler should have its own exception handler for kernel and hardware detected exceptions. The default exception handler returns an error.

If the exception handler returns an error, the kernel's actions depend on the execution environment. A debugger is called if one's installed. If not, and the exception occurred in a task or a software interrupt handler, the task is terminated.

If no debugger is installed and the exception occurred in a secondary interrupt handler or hardware interrupt handler, the exception is fatal to the system. Therefore, secondary interrupt handlers and hardware interrupt handlers should always have exception handlers if they might conceivably get an exception—even if the handlers only jump to a safe exit point.

An unhandled exception in an accept function is less catastrophic; the sending task receives an error. If the accept function held any locks or other resources,

those resources will never be released and the system will partially or fully hang. Accept functions should always have exception handlers to unlock and deallocate resources if a failure occurs. The exception handler can clean up and return a failure to the sending software, leaving the system in a fully functional state.

Performance and Software Structure

There are several performance-related issues that are affected by the choice of software structure:

- To a certain extent, memory space can be traded off for increased speed; however, this is not an absolute maxim. Any such tradeoff needs to be carefully considered.
- Boundaries such as execution environments and address spaces provide protection for executing software. Boundaries are set as the result of decisions about software structure, which are often made with consideration of robustness, flexibility and reuse potential as well. Communication across a protection boundary; however, is one of the more expensive services performed by the operating system, thus, tradeoffs exist between speed and desired level of protection, robustness, and other measures of software quality.

The following sections examine these issues.

Speed Versus Space

Often one thinks of *speed* as the major criteria and *space* as the resource to be traded for speed. You should avoid this inclination or your software may become too unwieldy for its purpose. For example, a space-versus-speed issue can arise when you are determining how many tasks to use to implement your software.

It is simpler to implement a task that handles one event (synchronous processing of an event by a single task) than it is to implement a task that handles several events. This fact suggests that the software should be structured as multiple tasks. In general, this conclusion is a good one if the software is structured such that

- there are a fixed number of tasks each performing a specific job
- there are a small number of tasks handling incoming requests (events)

If carried beyond a reasonable conclusion, however, a task could be created to handle each potential event. This could lead to a large number, perhaps hundreds, of potentially idle tasks, each taking a portion of memory.

In addition to the impact of memory use, more tasks may not even increase concurrency. If the processing time for the task were small, it might be slower to create and delete tasks than to queue the events and let them wait until a task became available to handle the event. In addition, on a single-processor system, if it takes just a few milliseconds to execute the task, the time-slice interval would make it unnecessary to have more than a few tasks handle incoming requests—the task would be finished before the time interval expired, meaning that it wouldn't be necessary to have another task waiting to handle a request.

Crossing Protection Boundaries

The execution performance of software can be affected by whether or not it needs to communicate with software in a different address space or in a different execution environment. Most applications, for example, run solely in the cooperative process address space as user mode tasks. Because these tasks are in the same execution environment (user mode task) and in the same address space, communication between the tasks is relatively inexpensive. The most expensive communication, however, is between user mode tasks in different address spaces.

If you want the advantages of memory protection for your software, or if it is desirable to implement your software in the client-server model with a client running in one address space, such as the cooperative process address space, and the server running in a protected address space, you must decide how to divide your software between address spaces and how each piece of software will communicate with the others.

The following points can be made about the relative performance characteristics of software execution, based on communication needs:

- Communication between user mode tasks in the same address space are relatively inexpensive.
- Communication between privileged software (supervisor mode tasks, accept functions, hardware and secondary interrupts, and software interrupts from supervisor mode tasks) is relatively inexpensive.

- Communication between user mode software (a user mode task or software interrupt from a user mode task) and privileged software using an accept function is inexpensive.
- Communication between user mode software and privileged software by other means is relatively more expensive.
- Communication between user mode software in different address spaces is most expensive.

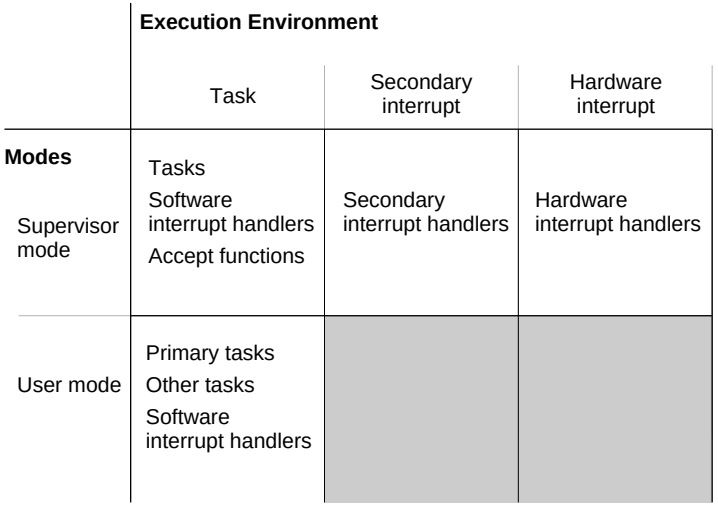
Execution Environments

Execution environments specify the rules under which software can execute. They are used along with privilege mode and area access rights to control the kernel services and resources that can be used. The environment isolates and protects software that is running concurrently—you cannot access code or data in a different environment directly; you must use kernel services. The three execution environments you can use are

- task (user mode and supervisor mode) environment
- hardware interrupt environment
- secondary interrupt environment

Figure 2-3 shows the execution environments and the kinds of software that can be executed within each one.

Figure 2-3 Execution environments



You structure your software to take advantage of one of the execution environments. The major criteria for choosing an environment are

- services available from the kernel and operating system within the environment
- data addressability, meaning the memory that can be referenced from an environment
- data residency, meaning whether the memory must remain resident or whether it can be paged

Table 2-2 shows the kind of services available to software in the different environments.

Table 2-2 Comparison of software by mode and execution environment

Software	Available kernel services	Data addressability	Must be resident?
User mode task or user mode software interrupt handler	All except for those related to interrupts and accept functions	Process's address space, including global data	No
Supervisor mode task or supervisor mode software interrupt handler	All	Global data	No
Hardware interrupt handler	Only those related to event flags and interrupts	Global data	Yes
Secondary interrupt handler	Those related to event flags and kernel notification, naming, timers, and messages	Global data	Yes
Accept function	All	Process's address space, including global data	No

Scheduling Algorithm

To share the processor with all tasks, the kernel can preempt the execution of one task and start—or resume—the execution of another. This form of processor sharing is called **preemptive multitasking**.

The kernel schedules all tasks preemptively, based on their priority and on their eligibility to execute. A task is eligible for execution whenever it is not waiting for some operation to complete, such as an I/O operation or loading a page into memory. Tasks that are not eligible for execution are said to be **blocked** on some event. Many tasks can be eligible for execution, but only one can be executing on a processor at a time.

The highest priority task that is eligible for execution is guaranteed to be the task that is executing. A task's **priority** is based on its relative importance. You

specify the priority based on the kind of software, for example, server processes, applications, drivers, and real-time operations.

A **context switch** saves the processor state of the currently executing task and restores the processor state of the next task to execute. The kernel performs a context switch when

- a task with a priority greater than the currently executing task becomes eligible for execution
- the currently executing task becomes blocked
- a task's **time slice**, which is the maximum time a task can execute before it must pause, is used up

For example, when a task is blocked on an event and the event occurs, the task becomes eligible to execute. If this newly eligible task has a priority greater than the currently executing task, the kernel performs a context switch, where the execution of current task is suspended (it still remains eligible) and the higher-priority task is resumed from the point at which it was blocked.

If several tasks have the same highest priority and are all eligible for execution, the kernel allows each task to execute for an internally specified time slice. When a time slice expires, the kernel switches to the next task with the same priority. The kernel uses this time-slice form of scheduling to give each task at this highest priority access to the CPU on a round-robin basis, such that the tasks take turn executing, in order. A task cannot starve the others unless it is the only task at the highest priority and it does not block.

The kernel never uses time slicing over its priority-based scheduling algorithms; it uses time slicing only when several tasks are all eligible for execution at the same priority and no higher-priority tasks are eligible. If a higher-priority task becomes eligible for execution, it will always get immediate access to the processor.

Time slicing is not used for all priority levels. It is not used with some tasks whose priority level is higher than an application's priority level.

The Process Manager assigns the same priority to all primary tasks in the cooperative process address space. If the primary task creates a secondary task, the primary task assigns the priority of the secondary task.

Note
The Process Manager blocks all primary tasks but one. This action ensures that only one primary task is eligible for access to cooperative services at a time. This primary task can be preempted by a secondary task, or a task in a process that is not a cooperative process, or by an interrupt; however, this primary task is not time sliced with other primary tasks. The primary task remains eligible until its process is no longer the current application. For specific switching rules, see *Inside Macintosh: Processes*. ◆

Interrupts immediately suspend the currently executing task. Hardware interrupts can interrupt secondary interrupts and both hardware and secondary interrupts can interrupt software interrupts. These interrupts are serialized and must run to completion before task execution can resume. For specific rules on the interactions between kinds of interrupts, see "Interrupts and Synchronization," beginning on page 74.

CHAPTER 3

Memory Management

Contents

The previous chapter described how you should structure your code depending on kind of tasks you define and whether the tasks execute as part of a cooperative process. Memory-related issues such as address spaces and addressability were touched upon in that chapter. This chapter provides a more complete discussion of memory management issues.

Address Spaces

Memory is organized into **address spaces**, which are a set of logical addresses that may be accessed by a processor at a given time. A 32-bit address space is used; thus, an address space can contain up to 4 gigabytes (2^{32}) of logical addresses. Logical addresses are mapped to physical locations in memory when the contents of the logical addresses are accessed. Logical addresses range from address 0 to address xFFFF FFFF.

An address space is divided into three parts, based on its usage and content:

- global memory for data
- global memory for code
- nonglobal private memory

Global memory is shared by all processes. The logical addresses of global memory locations are the same in each process, thus the content of each address in global memory is the same regardless of the process with which a task is associated.

Privileged software can only address global memory. User mode software can address private memory and global memory. Within an address space, areas determine the ranges of addressable locations, access rights to these locations, and whether or not they must be resident in memory. For more information about areas, see "Areas," beginning on page 45.

Resident, Pageable, and Virtual Memory

Memory can be **resident**, meaning that it must always be present in physical memory, or it may be **pageable**, meaning that it need only be present in

physical memory when it is referenced. The operating system provides a mechanism, called **virtual memory**, that allows address spaces to have many more logical addresses than the number of locations that are physically present in memory. For example, 16 MB of physical memory may serve to hold the contents of 128 GB of virtual memory containing 32 four-gigabyte address spaces. Thus, virtual memory allows you to access any logical memory location without concern for physical residency—the operating system is responsible for ensuring that the data is resident when it is needed.

When a memory location that is not resident is accessed, a page fault occurs. The software that caused the page fault is blocked until data obtained from the backing provider is physically resident in memory. A **page fault** is a hardware exception for which the kernel provides a handler. The handler responds by to the page fault by loading the page of data that is needed into physical memory from a backing provider.

Note
Only software running in the task level execution environment is allowed to cause page faults. For more information about execution environments, see "Execution Environments," beginning on page 36. ◆

A **backing provider** is an entity that manages backing objects. A **backing object** is typically a mapped file on disk, however, the backing provider could provide access to a backing object across a network instead.

Virtual memory is always present and it operates transparently to applications and other software executing in user mode. You typically need to be concerned with virtual memory only when you need to prevent page faults:

- When implementing an I/O driver you must ensure that hardware and secondary interrupt handlers do not cause page faults. Thus, all code executed by a hardware or secondary interrupt handler and all data accessed by them must be resident.
- When you wish to avoid untimely page faults; for example, when rendering live video onscreen, you may not want rendering to pause while data is being obtained.

In these rare cases, the data needs to be made resident before the software that uses it starts to execute. You can specify that a range of memory be made resident in one of two ways:

- You can specify that an area be physically resident when it is created.

- You can temporarily lock a page of pageable memory in physical memory, forcing it to become resident.

The use of resident memory removes physical memory from the set that is available for use with virtual memory. Regardless, enough physical memory must be available to make the data resident.

Areas

An **area** is a contiguous range of addresses; there are no holes in an area. When you create an area, you simply define a range of addresses that become addressable. After creating an area, you can choose how to structure it, for example as a pool, a heap, a stack, and so on.

An area associates part of the address space with a range of locations in a backing object. Backing objects may be memory-mapped files or scratch space. If you specify a memory-mapped file, the area has the structure and contents of the file. If you specify scratch space, the area's structure and contents must still be defined.

Areas have attributes that apply to the entire area, not just part of it. These attributes are:

- addressability, that is, whether the area is globally shared or private to a particular address space
- accessibility, which specifies the memory access rights (excluded, read only or read/write) for user mode and supervisor mode software. For information about accessibility, see the section "Access Rights," beginning on page 48.
- residency, which specifies whether the memory area is always resident in physical memory or whether it is pageable, meaning that it can be moved in and out of physical memory as it is used. (Note that a pageable area is not required to be paged in all at once.)

You can create your own memory areas and specify their attributes to suit your software's needs. The operating system provides areas to match the execution environment in which your software needs to run. The operating system may also provide higher-level APIs that create areas for you; for example, it may create an area for a mapped file when you call a routine to perform the mapping.

As an alternative to actually creating memory areas, you may be able to use preallocated pools that have the desired attributes instead. Your software performance might be enhanced if you can use a preallocated pool instead of creating your own area. Pools are discussed in the next section, "Pools," beginning on page 50.

There are several cases when creating an area may be useful:

- You need to allocate several pages of memory in a single chunk and you don't need to deallocate it or reallocate it frequently.
- You need to allocate page-aligned structures.
- You need to share memory.
- You need physically contiguous memory; this should seldom be necessary, as explained below.
- You need to allocate your own pool instead of using pools supplied by the operating system, as explained in the following section, "Pools," beginning on page 50.

When you create a memory area, in addition to specifying the area's attributes, you can specify how to initialize the area. You can specify

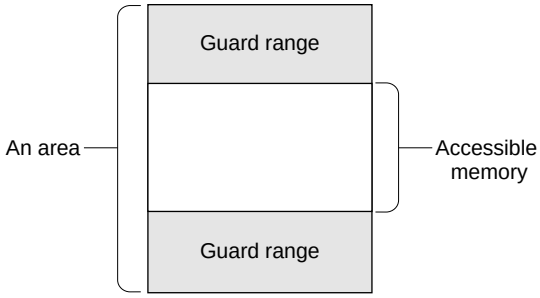
- whether or not to initially zero-fill the area. It may be useful in some cases for the software to be able to assume the area contains all zeros when the area is created.
- whether or not a pageable area is represented as a **sparse area** on the backing storage device. For a pageable area, the disk space that is needed when the area is not physically resident in memory may be allocated all at once or sparsely. Sparse allocation means that space is allocated incrementally as pages are actually needed. Using sparse areas may reduce the amount of disk space you need.
- whether or not a resident area will be created contiguously or not. A resident contiguous area is physically contiguous; a noncontiguous resident area is an area that is not required to be contiguous, whether or not it actually is. You should not need to create contiguous areas for most operations; you may need it if you are writing a device driver for a device and the device cannot handle scatter-gather transfers.
- whether a resident area is created as a sparse area or not. This is the same option as for a pageable area; in the case of resident areas, the option has implications for page faults when memory is referenced the first time. For a

resident area, the physical memory that is needed may be allocated all at once or sparsely. Sparse allocation means that a page is allocated incrementally as it is actually needed. Using sparse areas may reduce the amount of physical memory you actually need; however, a page fault will occur when the page is first allocated. Once a page has been allocated, a page fault cannot occur, regardless of whether the resident area was created sparsely or not.

- where to place the area. Placing an area allows you to specify the logical address at which the area should begin; otherwise the operating system finds an available range of address and uses them when creating an area. Typically, you only need to place an area that you want to appear in the same location in the private part of two address spaces. If you place an area, you should first make a memory reservation to ensure that the specified range of addresses are available. For information about memory reservations, see the section "Memory Reservations," beginning on page 48.

When you create a memory area, you can specify the size of a **guard range** of memory to be placed at the beginning and end of the area. The kernel allows no access whatsoever to these addresses; neither user nor supervisor mode software can write to or read from them. Figure 3-1 illustrates a memory area created with guard ranges. If any software, even the software residing in the area itself, attempts to access a guard range, the processor generates an exception. This makes it possible to detect conditions like stack overflows before they adversely affect surrounding areas.

Figure 3-1 A memory area with guard ranges



Access Rights

An area specifies two sets of access rights, one for user mode software, and one for privileged (supervisor mode) software. The three levels of access are

- excluded (no access)
- read only
- read/write

The access rights for privileged software, must be the same or greater than the access rights for user mode software. For example, if you specify read only for user mode access, you need to specify read only or read/write for supervisor mode access. If you specify read/write access for user mode, you need to specify read/write access for supervisor mode.

If you specify incompatible access rights, the operating system promotes the supervisor mode access to match the user mode access. For example, if you specify read/write access for user mode and read only access for supervisor mode, the area will be created with read/write access for both user mode and supervisor mode software.

Note
Some processors do not allow all combinations of user mode and supervisor mode access rights, even when supervisor mode access is the same or greater than the user mode access. You need not be concerned about which combinations are supported; the kernel always chooses the closest valid combination to the access rights you request.

For example, on Power Macintosh computers, a combination of excluded access for user mode software and read only access for supervisor mode software is not supported. In this case, the user mode access is promoted to read only. You can determine the actual access rights by calling a function to retrieve the area information after you create an area. ◆

Memory Reservations

A **memory reservation** allows you to reserve a range of logical addresses before the area is actually created. Thus, you can set aside a range of addresses

before you need to choose a backing object and specify other characteristics of an area. A memory reservation does not actually create an area; it prevents an area from being created arbitrarily in the specified range after the reservation has been made. After you have a reservation, you can create an area in it. You must create an area before the specified range can be addressed.

You should always obtain a reservation before creating a placed area. When you obtain the reservation, you know that you will be able to create the area at the specified location. The reservation also prevents the situation in which a task in another process attempts to create an area in the same place unintentionally while the first task is also creating the area.

A **global memory reservation** causes a reservation to be made for the same range of addresses in all address spaces. The memory reservation will also be made in all address spaces that are created afterwards as well.

Note
A global memory reservation can be made for either global or private part of an address space. ◆

Memory reservations can be used to reserve a range of addresses for sharing between the private part of two or more address spaces. This is necessary if the shared area contains pointers that may be referenced by software in each address space.

Memory Data Structures

After an area has been created, you can start to allocate structures within the area. You can leave the area unstructured, as well; for example, when you only need to store a large graphics image temporarily.

For areas that are associated with a file mapping, the structure is defined to be the same as the file's structure. To structure non-file mapped areas, you must define the structure. For example, you can create a pool in an area, which provides a structure for the area, and then allocate (and deallocate) memory from the pool.

You could also structure all or part of an area as a heap, stack, or some other kind of data. Although you can create several kinds of structures in a single area, you typically structure an entire area in only one way. For example, you might create an area and use it exclusively as a pool.

Some of these structures are automatically provided for you by the operating system. The following sections describe several kinds of data structures that you might find or create in an area:

- pools
- application heaps
- per-task structures, such those used for stacks and local storage

Pools

Pools are memory that can be allocated from areas created by the operating system or from areas that you create. A pool is identified by its address. Pointers are used to reference data in a pool. The Pool Manager handles allocation and deallocation from a pool.

You typically use pools to store transient data because storage in pools can be allocated and deallocated very efficiently. Advantages to using pools are that

- allocation and deallocation from pools is fast; these operations are much faster than creating or deleting areas
- pools can be dynamically grown
- access is pointer based (therefore, you don't have the indirection associated with Heap Manager handles)
- the Pool Manager is reentrant; therefore tasks that use pools need not synchronize their access to the Pool Manager—this is done for you

The operations allowed for pools depend on the area from which the pool was created. Table 3-1 shows how residency affects the operations that are allowed.

Table 3-1 Allowable pool operations

Execution environment and mode	Operations allowed for pool in pageable area	Operations allowed for pool in resident area
Privileged task level	Reference, allocate, deallocate	Reference, allocate, deallocate
Secondary interrupt handler level	Reference if known to be resident; otherwise none	Reference, deallocate
Hardware interrupt handler level	Reference if known to be resident; otherwise none	Reference
User task level	Reference, allocate, deallocate	Reference, allocate, deallocate

The operating system provides several pools that are created from different areas, thus the pools have different characteristics and access rights. You should use these pools if possible, because allocation and deallocation from an existing pool is much faster than creating an area and a pool within it before you can allocate and deallocate memory.

Each process has access to a **default pool**. Tasks and other user mode software can use the default pool as a heap for storing per-process global data. The default pool can also be accessed by accept functions. In general, applications will use either the default pool or the application heap. Application heaps are described in the section "Application Heaps," beginning on page 53.

In addition to the default pool associated with each process, the operating system provides three other pools for specific kinds of software:

- a system resident pool
- a system pageable pool
- a system global pool

Device drivers and other supervisor mode software commonly use the **system resident pool** when the software cannot tolerate page faults. (See Table 3-2 on page 52.) The kernel holds memory allocated from this pool in physical memory at all times. Only software running in supervisor mode can allocate memory from this pool. The data stored in the system resident pool is read-only for all user mode software.

Supervisor mode software that can tolerate page faults should allocate memory from the system pageable pool. The **system pageable pool** acts as the default pool for supervisor mode software. This pool, too, is read only for all user mode software.

User mode and supervisor mode software can use the **system global pool** to allocate memory that must be globally accessible to all code in every address space. Use this pool sparingly—any software can corrupt the contents of memory allocated from the system global pool.

Table 3-2 summarizes the use of default memory pools.

Table 3-2 Default memory pools

Pool	References allowed by	Pageable?	One per
System resident	Software at all levels except for user mode task level	No	System
System pageable	Software at all levels except user mode task, secondary interrupt, and hardware interrupt levels	Yes	System
System global	Software at all levels	Yes	System
Default	Software at user mode task level or by accept functions	Yes	Process

Note
The levels in Table 3-2 refer to execution environments. To determine which kinds software can execute in the various execution enviornments, see Figure 2-3 on page 37. ◆

In general, you should be able to implement your software using one of the pools provided. If you cannot use one of them, your software can create its own memory area and specify attributes suitable for its needs. Pools can then be created within the area. For example, if you need to address or share memory whose attributes are different than those of the existing pools, you can create a memory area with the needed attributes and create a pool within the area.

When you use the Pool Manager to create pools, they inherit the attributes of the areas from which they were created. The Pool Manager allows you to specify the initial size of a pool as it is created. You can also specify a specialized grow function that is executed to grow your pool when it runs out of space. The default grow function allocates a new area for a pool when it runs out of space.

Application Heaps

Each cooperative process has an **application heap** that is accessible from a primary task. You use it directly when you call Memory Manager functions such as `NewHandle`, and you use it indirectly when you call other Toolbox functions that need to allocate memory, such as for the window record that is created when you create a window.

Nonprimary tasks should not use the application heap, because the application heap is managed by the Memory Manager, which is non-reentrant. You should use pools, or areas, or call the standard C library function `malloc` instead.

Per-Task Data

As mentioned in the previous section, primary tasks have access to an application heap for storing data. All tasks can use several kinds of **per-task data**, which are memory-based data structures used for storing data. They include the following ones:

- stacks
- task local storage

Stack space for a task is managed by the operating system. A task's stack is used by software to hold local variables and parameters passed to functions. This data is always accessible to the task when it is executing.

It sometimes may be necessary to maintain static data for each task executing the same code, yet keep the data off the stack so that the value is retained even after the function that created the variable terminates. For these cases, you should use **task local storage** so the data will persist until the task is deleted. For example, if a task needs to maintain a connection with another piece of software, the software's ID can be kept in task local storage.

Software within each application or process that uses the context is allowed read/write access to the variables in the context. If you wish to prevent concurrent access to the variables, you must use synchronization services or, if you are implementing an application, you might restrict the use of these variables to the primary task.

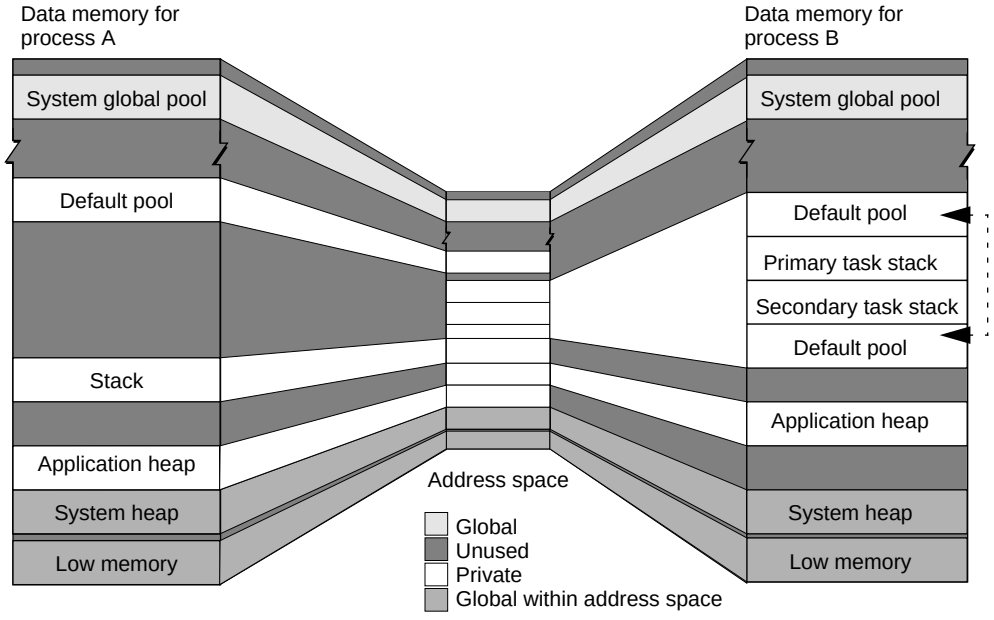
Cooperative Process Address Space

This section shows the various kinds of memory in the cooperative process address space. Each cooperative process has the following kinds of memory and data structures:

- global memory, which is mainly used for sharing code
- a default pool, from which tasks can allocate private data
- an application heap, for use by a primary task
- a system heap and low memory globals, which are used only for compatibility

Figure 3-2 illustrates how the data used by two cooperative processes might be arranged. Process A has a single primary task. Process B has a primary task and one secondary task.

Figure 3-2 Data memory areas for two cooperative processes

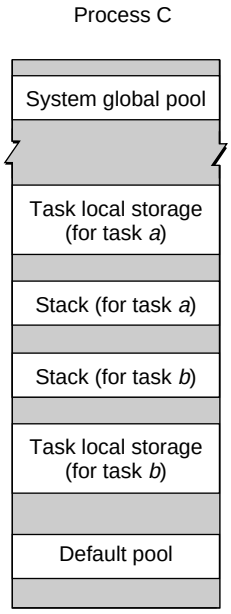


The cooperative process address space contains a system heap. The system heap, along with low memory globals, are global within the cooperative process address space; they are not, however, global across address spaces. Notice that the default pool for process B consists of two discontiguous areas. Pools can be dynamically grown; when they grow, they may appear in discontiguous areas.

A Protected Address Space

Every process except cooperative processes can have its own address space. Figure 3-3 shows two tasks associated with a process in a protected address space.

Figure 3-3 Data memory areas in an address space for a process with two tasks



Process C consists of two user mode tasks. There is a user mode stack for every task in the process, and a default pool shared by all the tasks in the process. Each task can have its own task local storage.

Shared Memory

Shared memory is used for sharing code and data. The following sections discuss how you can use shared memory.

Shared Data

Shared data provides the ability for software executing in one address space to share memory locations with software running in another address space. Thus, shared data allows communication without significant overhead. A synchro-

nization mechanism must be used to prevent accidental overwriting of memory that is shared. You should use shared memory if you need to transfer a large amount of memory between address spaces frequently.

As already noted in the section "Address Spaces" on page 43, global memory appears at the same location in every address space. All software can address global memory. You can use global memory for sharing; however, using it for that purpose requires cooperation and can leave the entire system vulnerable:

- Global memory is a limited resource; because other processes may allocate global memory also, you must be concerned about how to handle failures.
- When user mode software is allowed write access to global memory, anything within the shared area can be changed by any software running on the system.

Given these limitations on the use of global memory, you should only use it for privileged software and only when a small amount of data is being shared.

There are several alternatives to using global memory for shared data. They require more effort to set up; however, they overcome the vulnerability associated with global memory:

- Use memory whose addresses map to the same locations on a backing storage device. This technique is called **file mapping** on some systems. It uses the virtual memory mechanism which makes changes to a backing object associated with virtual memory immediately visible to all address spaces that use the locations. This technique, however, is only useful if the data is pageable, because resident memory is not associated with a backing storage device.
- Explicitly specify an area to be shared between processes in two or more address spaces.

Shared Code

Shared code allows several instances of software to run concurrently with only one copy of the code present in logical memory. Shared code is implemented as **code fragments,** simply called fragments, which are containers of executable code packaged by a linker and prepared for execution as they are loaded. Each fragment consists of its code, static data, imported symbols, and exported symbols.

The **Code Fragment Manager** automatically loads fragments into memory and prepares them for execution. Once loaded, another instance of the software can be executed with only the additional overhead of stacks and per-task data for maintaining the execution state; another copy of the code is not required.

Fragments that export functions and variables to other fragments are called **shared libraries.** Because all fragments are potentially sharable (although not all are actually shared), the terms *fragments* and *shared libraries* are often used interchangeably. In general, a shared library is used to resolve imported symbols during linking and also during the loading and preparation of some other fragment.

A shared library that is dynamically linked at execution time is called a **dynamically linked library**. A dynamically linked library exports code or data that can be referenced by another fragment. For example, while linking, an application fragment can import a math library and the Window Manager library. At execution time those libraries are dynamically bound to the application.

Note
Do not confuse shared libraries built with the Code Fragment Manager with shared libraries used by the Apple Shared Library Manager (ASLM). ASLM is not supported. ◆

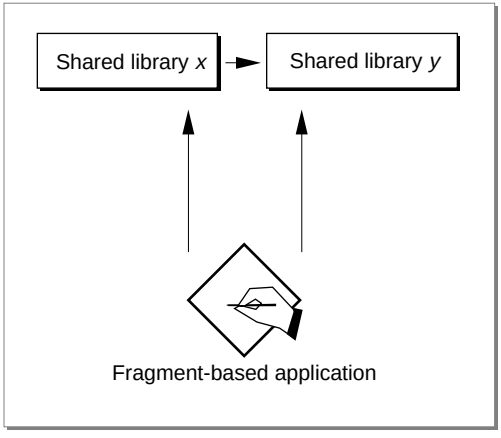
Using shared libraries for software development has many benefits, including the following:

- Having software in separate pieces simplifies development. For example, if you need to enhance the spell-checking module of your application, you need only to change and replace that shared library. You can distribute the enhanced shared library as a replacement, instead of distributing a new version of the application or a patch.
- When two or more applications use the same shared library, memory is saved because only one copy of the code is in memory.

All system services, including system software, are provided through shared libraries. All fragment-based software gains access to system services by directly calling shared libraries. System services also access each other directly, one shared library to another.

For example, Figure 3-4 shows a fragment-based application accessing two shared libraries, *x* and *y*, directly. The figure also shows shared library *x* accessing shared library *y* directly.

Figure 3-4 Access to system services in Copland



To allocate per-process static data, the Code Fragment Manager allocates one copy of a library's static data from the default pool for each process that uses that library.

The **System Object Model (SOM)**, a new model for developing and packaging object-oriented software, is also supported. SOM makes object-oriented shared libraries viable by providing release-to-release binary compatibility, compiler and language independence, and a basic level of dynamic language support. SOM is implemented as a layer on top of the Code Fragment Manager.

CHAPTER 4

Synchronization Services

Contents

Whenever two or more resources could be shared by concurrently executing software, synchronization is required. Although concurrent execution is not possible on a single processor machine, interleaved execution between tasks is possible. Interleaved execution exhibits the same effects as concurrent execution with respect to synchronization. The actual result of an operation can be different than the correct result if you allow unsynchronized access to a shared resource.

Synchronization services provide ways that you can guarantee orderly access to resources by your software. You must use synchronization services whenever you allow access to a resource from software that has the potential to execute concurrently with other software.

This chapter introduces why you need to use synchronization and describes the synchronization services provided by the kernel and operating system. Then, a section follows that explains the effects of using various synchronization services with different kinds of software.

Introduction to Synchronization Issues

Consider two tasks, task A and task B, that each execute the following C-language statement:

```
if (x==0)
    x = x + 1;
```

These tasks can be used to show a classic example of the effect of serialized versus interleaved execution on a shared resource; in this case, a single memory location represented by the variable `x`. In this example, if the test for `x==0` is interleaved with the execution of `x = x + 1`, the result can be wrong.

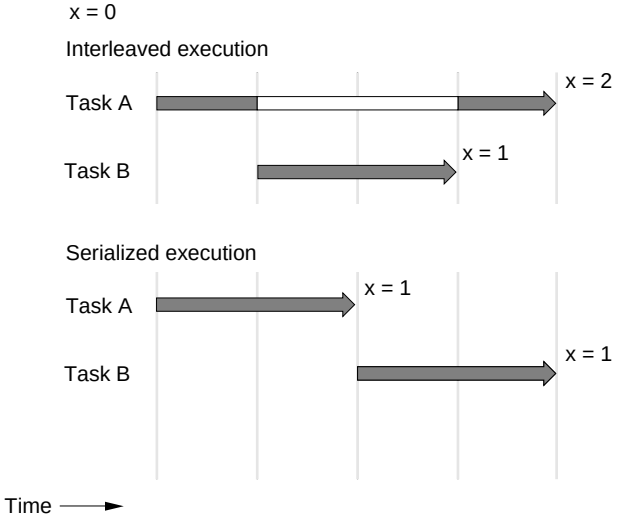
If task A executes the statement first and then task B executes the statement, the result for `x` is 1. If task B executes the statement first and then task A executes the statement, the result is the same. In fact, for any initial value of `x` and any number of tasks executing the code atomically, the result will be either 0 or 1.

If the execution of the tasks is interleaved so that the comparison for `x==0` is separated from changing the variable, the result may be different (and wrong) from the serialized execution. To ensure that one of the correct results is achieved, you must prevent interleaved execution of statements that affect a

shared resource. In this case, the compare and change actions must be atomic; that is, the actions must be completed, start to finish, as a single undivided operation.

Figure 4-1 shows the serialized and interleaved execution sequences.

Figure 4-1 Serialized versus interleaved execution



In this example, the comparison and the potential change to the value form a **critical section**, which is a section of code whose execution must be serialized so that it is atomic with respect other code that may affect the same shared resources. In this example, the comparison and change to a shared resource, the variable x, must be in a critical section in each task that wants to manipulate the variable.

Note
The critical section in this example consists of code that compares the value of only one memory location, in which case you can use the compare-and-swap atomic instruction without having to consider the broader issue of what code must be in the critical section. For more information about the compare-and-swap instruction, see the section "Atomic Instructions," beginning on page 66. ◆

The key to using synchronization services is to be able to

- make explicit decisions about which resources are to be shared. If more than one task has concurrent access—and therefore the potential for interleaved execution—for either reading or writing to a shared resource, synchronization of access to the resource is probably required. If several tasks can write to the resource concurrently or if tasks should not attempt to read from the resource while it is being updated lest an inconsistency result, synchronization is definitely required.
- determine which part of the execution must be serialized and therefore must be in critical sections. Synchronization reduces the potential for concurrency because execution of a task may have to wait while another piece of software is executing within the critical section. In the worst case, serialization could eliminate concurrent execution altogether. You must make decisions about the scope of the operation that must be serialized and try to minimize the amount of time spent in critical sections.
- make sure that access to a shared resource is handled consistently. The kernel and operating system provide the services that implement synchronization. You are responsible for developing a protocol for using synchronization services with respect to each shared resource. The synchronization services know nothing about your protocol. If the protocol is violated, for example, if a task is allowed to change a shared variable outside of a critical section, erroneous results that are often difficult to diagnose can occur. Perhaps the best way to implement a protocol like this is to build the synchronization calls into the routines that handle the access to the resource so that whenever you call the routines to access the resource, synchronization is automatically performed.

Synchronization services can provide a communication service in addition to controlling access to critical sections of code. You seldom will use a synchronization service unless you wish to communicate data, because the reason you share data is to communicate values between tasks. For this reason,

you should also be familiar with communication services provided by the kernel and operating system before making decisions about how best to synchronize communication. For an overview of communication services, see "Interprocess Communication Services," beginning on page 17.

About Synchronization Services

The following sections describe the major kinds of synchronization services provided by the kernel and operating system:

- Synchronization primitives and locking provide processor-supported atomic instructions and locks for implementing critical sections.
- Event groups let you wait for a condition to occur so that execution of a critical section will not start until the condition is satisfied.
- Kernel queues also let you wait for a condition to occur; however, they maintain more information about changes in condition and can also be used for explicit but limited communication.
- Interrupts can also be used to synchronize operations. They can also be disabled, which is not recommended in most cases.

The following sections describe each of the synchronization services.

Synchronization Primitives and Locking

Synchronization primitives are atomic instructions that operate on 4-byte long-word aligned memory locations. Synchronization primitives are used to implement locking, which has two variations, simple locks and read/write locks. You can use both synchronization primitives and locking mechanisms.

Atomic Instructions

The operating system provides several kinds of atomic instructions. **Atomic instructions** implement undivided operations that, once started, are carried to completion. The data manipulated by an atomic instruction is a 32-bit value in a 4-byte longword-aligned structure, as follows:

- *compare and swap* the contents of a memory location, which allows you to make a comparison of the contents of a single memory address and, if true, allows a value to be exchanged for the current one.
- *test and set* an arbitrary bit in memory, which is similar to compare and swap but for a 1-bit value rather than a 32-bit value. The bit is specified as an offset from a single memory address.
- *addition* to (and by addition of negative values, subtraction from) the contents of a memory location
- *increment* or *decrement* by 1 the contents of a memory location. Note that `n++` or `n = n - 1` in a higher level language, such as C, is not by itself guaranteed to be atomic; it could be implemented as several machine instructions.
- *bitwise operations*, such as AND, OR, and XOR.

Atomic instructions for addition, increment, decrement, and bitwise operations are also provided for 8-bit and 16-bit values as well.

If you need synchronization, you should use atomic instructions whenever possible because they are very efficient. (If you don't need synchronization, don't use them.) Atomic instructions are simply wrappers around hardware instructions and thus do not even require kernel intervention. Unfortunately, they guarantee serialized access only to individual memory locations. If the resource is some other kind of data structure, you cannot synchronize access with atomic instructions. In addition, you must use caution when deciding whether part of a data structure for which you want to serialize access is aligned correctly; for example, many fields in Toolbox data structures are not long-word aligned.

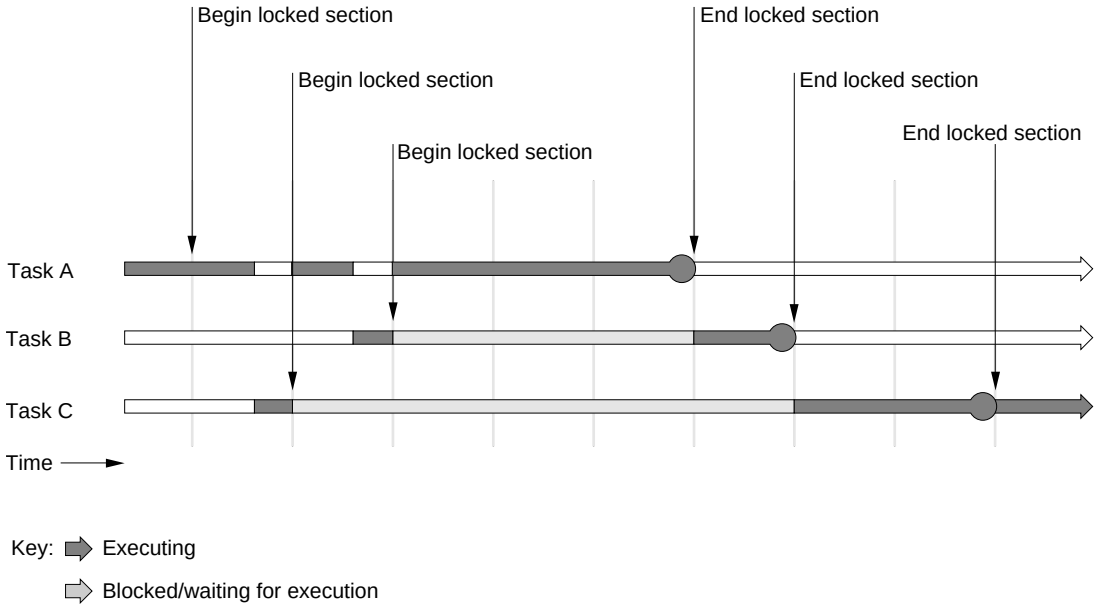
Simple Locks

A simple lock can be used to synchronize access to several memory locations. Typically, your code attempts to acquire the lock at the beginning of a critical section. After executing the critical section, your code must release the lock.

If the lock is in use, the request blocks waiting for the lock or it fails, depending on how you attempted to acquire the lock. Your choice of whether to block or fail depends, at least in part, by the kind of software; you cannot allow a hardware or secondary interrupt handler to block.

Figure 4-2 shows how critical sections in tasks A, B, and C that affect the same data can synchronize access to the data by using a simple lock.

Figure 4-2 Using simple locks



In Figure 4-2, task A receives access to the critical section and tasks B and C block when they attempt to enter their critical sections. When task A finishes, either task B or C could acquire access to their critical section; in Figure 4-2, task B is shown executing its critical section before task C executes its critical section; however, the scheduling algorithm determines which task executes next.

If you allow a task and its software interrupt handler to try to obtain the same lock, you must disable software interrupts when the lock is acquired by the task. This is an option when acquiring the lock and is performed very efficiently. If you do not disable software interrupts, a software interrupt handler for a task might try to acquire the lock that is already held by the task. The software interrupt handler would preempt the task before the task could release the lock, and the software interrupt handler would block waiting for a lock that cannot be released.

You should use a simple lock when an atomic instruction is insufficient—that is, whenever more than one word needs to be manipulated atomically. There

are three cases where you need to use a more robust, but somewhat slower, locking mechanism:

- when you want to allow read access to the data most of the time yet want to synchronize access while updating the data
- when tasks running at different priorities might attempt to acquire the same lock and you are concerned about priority inversion
- when you want to allow recursive locking, for example, when a routine in a library acquires a lock and the routine is not aware of which other routines might attempt to acquire the lock.

In these cases you should use read/write locks, as described in the next section.

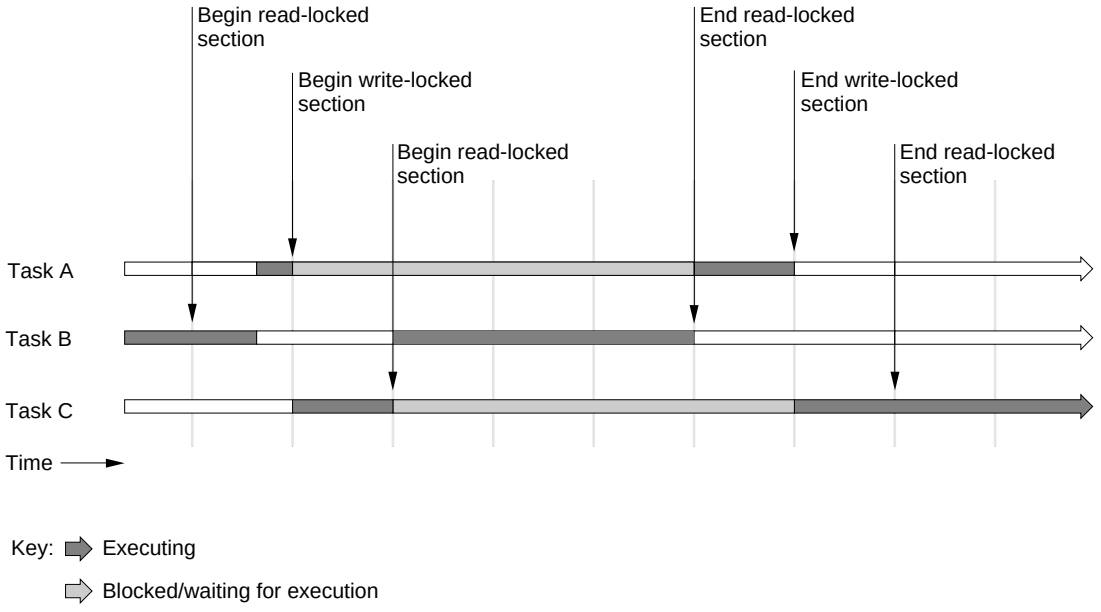
Read/Write Locks

Read/write locks provide all the capabilities of simple locks. In addition, they can be held exclusively by software (identified as a *writer*) that wants to change data in a critical section, or be shared by software (called *readers*) that wants only to view the values. The readers and writer always attempt to access the data from within a critical section.

Before entering the critical section, a reader or writer must obtain access by acquiring a lock. A writer obtains an exclusive lock, which prevents any readers or another writer from obtaining the lock; these tasks remain blocked until the writer releases the lock. The writer cannot obtain the lock, however, until all readers have released their locks; the writer remains blocked until no readers (or another writer) hold the lock.

Figure 4-3 shows how task A, a writer, might interact with tasks B and C, which are readers.

Figure 4-3 Using a read/write lock



In this example, task A attempts to enter its critical section, and because it wants to perform an update, it attempts to acquire an exclusive lock. Task B has already acquired a shared lock and therefore task A is blocked waiting for task B to release the shared lock. During this time, task C attempts to acquire a shared lock. Because task A's request for an exclusive lock is pending, task C must wait for task A to acquire and then release its exclusive lock.

If another writer makes a request for an exclusive lock while task B is executing its critical section, task C will have to wait longer because a new writer would gain exclusive access before any reader gains shared access.

The previous discussion assumes that tasks A, B, and C are running at the same priority level. If a lock is held by a lower priority task and a higher priority task is waiting for it, a priority inversion may occur, such that the higher priority task may be blocked indefinitely waiting for the lower-priority task to release its lock. If a read/write lock is being used by different priority tasks, you can specify an option that temporarily raises the priority of the lower-priority task

so that it can release the lock and allow the higher-priority task to run; after releasing the lock, the lower-priority task continues at its original priority.

There is one other feature of read/write locks. A writer, holding an exclusive lock, can be demoted to a reader holding a shared lock without having to release the lock. This is subtly different from a writer that releases its exclusive lock and immediately thereafter attempts to acquire a shared lock. In the first case, where the writer is demoted, the lock is not actually released. In the second case, the lock is released and another writer may gain the lock before the shared lock is granted to the former writer.

Event Groups

Event groups allow a task to specify one or more conditions (with a maximum of 32 conditions) and then wait for any or all of the conditions to be met before resuming execution. Each condition is represented by an **event flag** (numbered from 0 to 31) in a 32-bit **event group**.

You can manipulate each flag in one of two ways:

- Set the flag, which specifies that the condition has been met.
- Clear the flag, which specifies that this condition is no longer of interest, until the flag is set again.

You set and clear flags for an event group in one operation. Setting, clearing, or testing flags in an event group are atomic operations; thus, you are guaranteed that several flags can be set, cleared, or tested as a single undivided operation.

When you create an event group, all flags are cleared and each flag remains clear until it is set. After a flag has been set, it remains set until it is cleared. If a task waits on a flag that has already been set, it will not be blocked, because the condition has already been met.

The waiting task specifies the group that contains the flags, a timeout value (which is allowed to be infinite), a mask, and some options. The mask specifies the flags that can unblock the task, and the options specify how to interpret the mask and whether to clear the flags.

When one or more flags are set, tasks waiting for flags in the event group are tested against these flags in priority order, and if the priority is the same, they are tested in order of how long the task has been blocked. Thus, a higher priority task is unblocked before a lower priority task and, given equal priority, the task that has been blocked longer becomes unblocked sooner.

The fact that more than one task can become unblocked makes event groups a powerful yet potentially complex technique for coordinating execution, because the options for unblocking a task can affect just the highest priority task (or the longest waiting task when there is a tie) or the options can affect all waiting tasks.

The options allow you to specify rules for unblocking waiting tasks. The conditions for which each task is waiting are examined task-by-task in priority (and waiting time within priority) order. You can specify one of the following options:

- If any of the conditions are met, unblock the task.
- If all of the conditions are met, unblock the task.
- If any of the conditions are met, unblock the task and clear the flags; other tasks waiting on those flags remain blocked until those flags are set again.
- If all of the conditions are met, unblock the task and clear the flags; other tasks waiting on those flags remain blocked until those flags are set again.
- If any of the conditions are met, unblock the task; clear the flags only after conditions for all waiting tasks have been tested—thus, all eligible software becomes unblocked.

Typically, you use the same clear options for each task in an event group; otherwise, the complexity of interaction increases significantly. A typical example might be to set up several tasks capable of responding to the same event. If all tasks specify clearing the flags, then only one task will handle the event. Unblocking a task would prevent the other tasks from being unblocked as well.

Event groups are useful in the following situations:

- You need to wait on a combination of events occurring.
- You don't need to distinguish between multiple occurrences of an event; an event flag could be set multiple times before it is cleared.
- You need to signal an event from a hardware or secondary interrupt handler.
- You don't need to associate data with an event

If you need to queue events so that you can take action on each one rather than treat the event as a gate that is opened when a condition has been met, you should use kernel queues. The performance of kernel queues is slower than

that of event groups, and notification of the event cannot be made from a hardware interrupt handler. Kernel queues are described in the next section.

Kernel Queues

Kernel queues are a mechanism that allows software to wait on an event—the event, called a **notification**, is an entry being made to a queue. Kernel queues are similar to event groups in that they both allow software to block waiting for an event to occur. For kernel queues, the event notification is placed in a queue; whereas events for an event group are simply set.

Kernel queue notification has different capabilities leading to different results than does setting an event for an event group:

- The notification allows you to transfer three words of data from the notifier to the waiting software.
- Notifications are queued, which allows software to distinguish between multiple occurrences of the same event.
- The notification and its data is delivered only to a single piece of software, which is the one that has been waiting the longest. Thus, waiting software is notified on a first-come, first-served basis regardless of priority, and once the notification is delivered, it is no longer available to other software.

Kernel queues are often used for notifying software about the completion of asynchronous events. For example, a secondary interrupt handler could notify a task when an I/O operation completes execution.

Any kind of data may be transferred between the notifying software and the waiting software. You must establish a protocol for interpreting the data. For example, you might use a protocol that interprets the data in the following way:

- The first word specifies the kind of event, for example, the completion of an I/O operation.
- The second word specifies information about the operation, for example, the status resulting from the I/O operation.
- The third word specifies information about the state of the operation when it was started, for example, whether it was reading or writing.

When you wait on a queue, you also can specify a timeout value. A timeout value of 0, for example, could be used to check if the queue is empty. You would not use a timeout value of 0 in a loop, however, because it would be a

polling loop. Polling is not recommended, because it causes the software to be active most of the time and results in unnecessary context switches.

In general, you should use kernel queues for synchronization only if you cannot use other services, such as event groups or locking, because kernel queues have more overhead than other synchronization services. If, however, you need a communications capability and can accept the three-word message size restriction, kernel queues are one of the most efficient interprocess communications services. For more information about interprocess communication services, see the section "Interprocess Communication Services," beginning on page 17.

Interrupts and Synchronization

The kernel serializes the execution of some interrupt handlers; thus these interrupt handlers provide synchronization. Also, disabling interrupts has synchronization implications. The following sections describe the synchronization that occurs with the use of software interrupt handlers, secondary interrupt handlers, and disabling hardware interrupts.

Software Interrupt Synchronization

Software interrupt handlers associated with the same task are guaranteed to be serialized. Once a software interrupt handler has started execution, it cannot be interrupted by another one associated with the same task.

Software interrupt handlers that are associated with different tasks are not serialized in any way; locking must be used before a software interrupt handler attempts to access shared data. If it is necessary to acquire a lock within a software interrupt handler, you must take additional steps to avoid deadlock if that lock is also shared by a task and a software interrupt handler associated with the task.

If a software interrupt handler associated with a task and the task itself attempt to use a lock, either the software interrupt handler's request must not be allowed to block or the task's software interrupts must be disabled—disabling software interrupts is an option you can specify when requesting the lock. Otherwise, the lock might be granted to the task, and when a software interrupt occurs, the software interrupt handler could block waiting to acquire the lock. In this case the task could not release the lock, because it is now blocked waiting for the software interrupt handler to complete.

Secondary Interrupt Synchronization

Secondary interrupt handlers are guaranteed to be executed sequentially. You can use secondary interrupt handlers as an intermediary to allow a hardware interrupt handler to share data with a task. Locking cannot be used in this case, because a hardware interrupt handler cannot be allowed to block. If a hardware interrupt handler fails to acquire the lock, the handler has no recourse that ensures synchronization.

Consider the following example. A hardware interrupt handler may fill up a buffer with input data. A task is responsible for taking data out of the buffer and processing it in some way. A secondary interrupt handler would be scheduled by the hardware interrupt handler whenever data became available for processing. A secondary interrupt handler would be called by the task when it is ready to process the data. Since both secondary interrupt handlers cannot execute at the same time, the data will only be accessed serially by one or the other secondary interrupt handlers.

Note
If only a small quantity of data needs to be transferred from the hardware interrupt handler to the task, a secondary interrupt handler could notify the task through a kernel queue. This would reduce the performance bottleneck of using another secondary interrupt handler from the task. Transferring more than three words at a time, however, would require a solution such as this one. ◆

Synchronization by Disabling Hardware Interrupts

You rarely need to disable hardware interrupts; however, you may need to do so if you must update more than a single memory location atomically and these locations are shared by a hardware interrupt handler and a secondary interrupt handler. In this case, you may need to disable hardware interrupts. If possible, you should try to disable only a particular device's interrupts using mechanisms provided by the I/O system.

Only privileged software can disable hardware interrupts. Keep in mind that hardware interrupts could be lost during the time that the interrupts are disabled, thus you must consider the absolute amount of time you have on a processor-by-processor basis before lost interrupts become an issue.

IMPORTANT
Although you can disable hardware interrupts, you should seldom if ever find the need to do so. If you must disable hardware interrupts, call `DisableHardwareInterrupts` rather than a processor-level instruction; otherwise, your software will fail on multiprocessor systems. ▲

Synchronization and Software Structure

The choice of software affects the kind of synchronization services that are available. This section outlines how you might use synchronization to share resources between different pieces of software. Remember that:

- Hardware interrupt handlers can only use atomic operations, use locking operations that are not allowed to block, and set event flags.
- Secondary interrupt handlers can use all synchronization services available to hardware interrupt handlers; in addition, secondary interrupt handlers can clear event flags.
- All other software (software that runs in the task level execution environment) can use any of the synchronization services provided by the kernel and operating system.

The following points can be made about how to synchronize between different kinds of software:

- Synchronization is *not* required
  - between hardware interrupt handlers on Power Macintosh computers
  - between secondary interrupt handlers
  - between the software interrupt handlers associated with the same task

  This is because execution is already serialized
- Atomic instructions can be used by any kind of software.
- Secondary interrupt handlers and supervisor mode tasks and their software interrupt handlers can disable hardware interrupts, effectively preventing hardware interrupts or secondary interrupts from occurring.
- A task can disable the receipt of software interrupts directed to the task.

- Locking can be used with tasks (user mode or supervisor mode) and software interrupt handlers; however, software interrupts may need to be disabled or nonblocking locking operations may need to be used in conjunction with resources shared between a task and its software interrupt handlers.
- User mode tasks and their software interrupt handlers cannot synchronize with hardware or secondary interrupt handlers except through atomic instructions.

IMPORTANT

Never use task priorities to attempt synchronization—it will not work. For example, a higher-priority task may be preempted by a lower-priority task at any time because of conditions such as page faults. Always use synchronization techniques when sharing data between tasks. ▲

Synchronization and Multiprocessing

The kernel and operating system are designed to handle multiprocessing. Unless you're using hardware or secondary interrupt level or disabling interrupts, you shouldn't have to worry about it — everything will just work. (These things should not be relevant unless you're writing a device driver). For example, imagine that you have a server with multiple tasks, and two of your tasks get scheduled simultaneously on different processors. This doesn't introduce any new synchronization requirements for those tasks — it looks just like a single CPU that happens to be switching between the two tasks very frequently. If your server would fail on a multiprocessing system, it would also fail in a single-processor system that happened to preempt your task at just the wrong moment.

It is possible to write task level code that works only on a single-processor system because of the way the kernel schedules tasks at certain priorities. For example, in the absence of page faults, run-til-block scheduling behavior can be misused to simulate cooperative scheduling—and it will fail on an multiprocessor system. As stated in the previous section, you should always enforce execution order using explicit synchronization operations, never with knowledge of scheduling behavior. The kernel's scheduling behavior or other

aspects of the system may change, causing your code to fail even on a single-processor system.

If you are using secondary interrupt level or disabling hardware interrupts, you must remember to follow these rules:

- If you synchronize access to shared data with atomic routines, always use them to access that data at hardware or secondary interrupt level as well as at task level.
- If you synchronize access to shared data using a secondary interrupt handler, always use a secondary interrupt handler to access that data. Don't assume that just because hardware interrupts are disabled that the data is safe.
- Always call the `DisableHardwareInterrupts` subroutine to disable hardware interrupts.

CHAPTER 5

Messaging Service

Contents

The messaging service is an interprocess communications service for sending messages between tasks. Both synchronous (send an message and wait for a reply) and asynchronous (send a message and continue) forms of messaging can be used. This chapter introduces the messaging service and explains how it can be used.

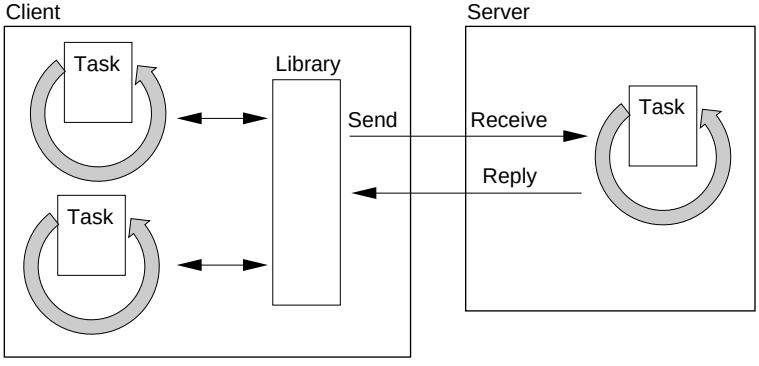
About the Messaging Service

Messaging is an interprocess communications service that allows a message to be sent from one piece of software, the **sender**, to another piece of software, the **receiver**. The **message** is a contiguous set of bytes (which may be zero bytes) that is understood only by the sender and receiver; it is not interpreted by the messaging service. A message is always associated with a **reply**, which is a response (that might also be zero bytes long) from the receiver to the sender. Thus, the message and its reply form a transaction between the sender and receiver.

The sender of a message is typically the client of the receiver. The receiver is often a kernel process set up to be a server. Clients seldom need to call functions that interact with the messaging service directly because messages typically are sent, received, and replied to as the result of the client calling functions in a library provided with the server software.

Figure 5-1 shows how a client application could call a library that establishes communication with a server.

Figure 5-1 Client-server communication using messaging



This example implies that the client must wait until it receives a reply. This technique is implemented with a **synchronous send** operation, which causes the sender to block while waiting for a reply. Alternatively, you can use an asynchronous send, which does not require the sender to wait. For more information about send operations, see"Sending Messages" on page 85.

If you develop an application, you seldom need to deal with messages directly; you simply call functions in the library provided with the server. If you are implementing a server that uses the messaging communications service, however, you must consider how to implement the software both as a client library and as a server process. The major issues are how to

- set up the messaging service
- send a message from the client side
- receive a message on the server side
- handle asynchronous communication
- use accept functions
- reply to a message

The following sections discuss these issues.

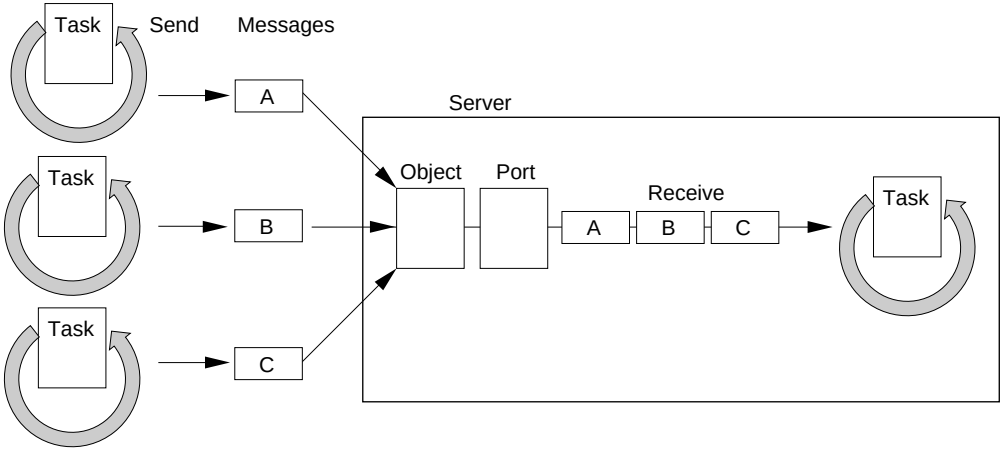
Setting Up the Messaging Service

The messaging service allows the sender to send a message to an object. An **object** represents something to which a request can be made. For example, an object might represent a window or dialog box that can be used to formulate a search of a database (and display the results of the search), or an object might represent a file that can be read or written to, and so on. Messages from many different clients can be sent to the same object.

A **port** represents the place where a message is delivered after it is sent to an object. An object is associated with only one port at a time. The receiver software can get the next message from a port, take some action, and then reply. Objects are small. Ports are larger than objects.

Figure 5-2 shows how a single object can be associated with a port. This model could be used to implement a datagram service.

Figure 5-2 A port handling one object

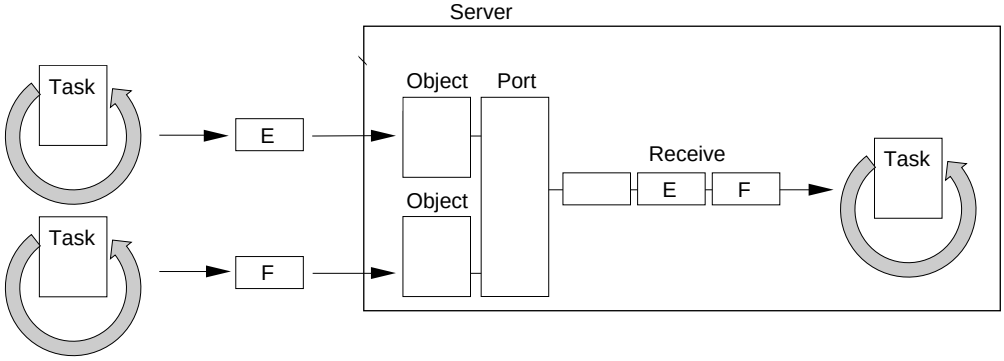


When only a single object is associated with a port, it may not seem necessary to make a distinction between an object and a port. However, although an object can be associated with only one port at a time, a port can have several objects associated with it. This is the typical way that a connection-based

service (for example, a file server) is implemented. The server is associated with one port and there is one connection per object to the server.

Figure 5-3 shows a server process whose port is associated with two objects.

Figure 5-3 A port handling several objects



As the message conceptually "moves" from the object to the port, the messaging service notes the object to which the message is sent; the server can make use of this information as needed. Each object has a reference constant that is only used by the server. Typically, the server places the address of a control block in this reference constant. In the example of a file server, the control block might be the actual control block associated with a file.

You must create a port before you can create an object. The client sends a message to an object; the object is identified by ID. The client typically does not know the ID of the object initially. To handle this situation, you must create at least one object for the port and use the system registry to make it available for clients to use. Clients should look up the server by name and obtain an ID. If the server supports multiple objects, such as one per connection, the server could respond to a message sent to the published ID by creating a new object and returning the new object's ID. For more information about the system registry, see "System Registry," beginning on page 93.

Sending Messages

Messages are sent to objects. A message may be sent either synchronously, meaning that the sender blocks until a reply is received, or asynchronously, meaning that the sender can continue and a reply will be delivered later. For information about asynchronous sends, see the section "Asynchronous Sends and Receives," beginning on page 88.

When you send a message, you can specify the following items:

- the message buffer and its length
- a message type (see the following section, "Receiving Messages")
- an optional reply buffer and its length
- an optional timeout value
- options

You place the message in the buffer and specify its length. Depending on the options you choose, you can allocate a buffer that will contain the reply. You also can specify a timeout value; however, using one can complicate programming for two reasons: you won't know why the receiver didn't reply in time (it could just be slow or it may not be running), and you must decide what to do when a timeout occurs (for example, whether or not to try again). The additional complexity associated with using a timeout value, however, may be justified to prevent the sender from hanging.

Options specify, among other things, how to transfer the data. You can transfer the data by value, or reference, or you can let the messaging service decide. In most cases, you should let the messaging service decide how to transfer the data because it will choose the most efficient method. The most efficient method is not always obvious.

If the message is sent by value, the messaging service copies the message into the buffer provided by the receiver. If the message is sent by reference, the messaging service automatically makes the sender's buffer addressable and accessible to the receiver.

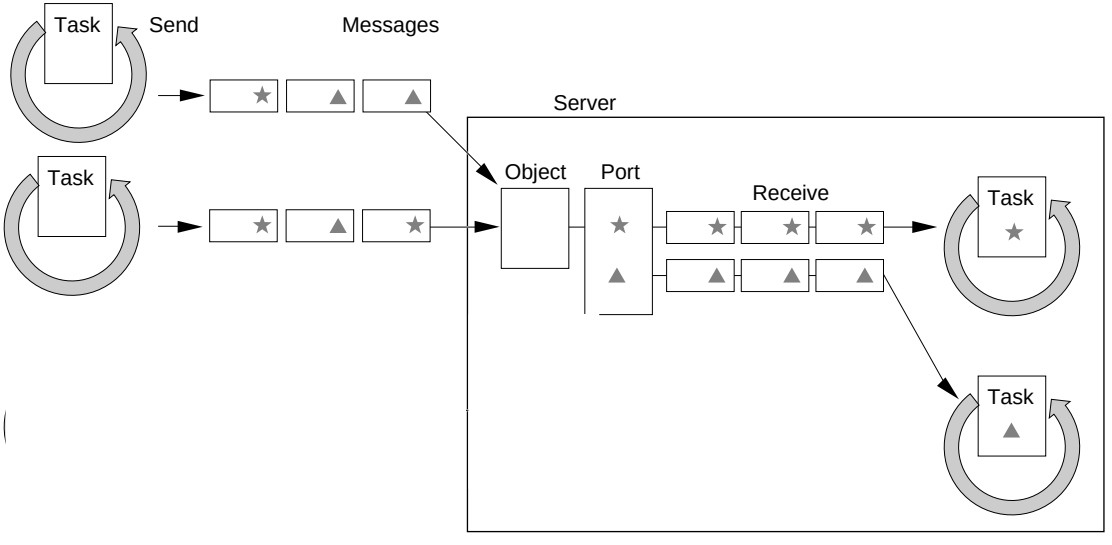
The messaging system will also make a best effort for delivery. For example, if it would be more efficient to copy the data but there is not enough buffer space in the receiver, the messaging system will transfer the data by reference, if possible.

Receiving Messages

Messages are received in the order they are sent to an object and are released to the port for processing. The receiver can distribute the messages to tasks or an accept function for processing based on the message type of the message. A **message type** is a 32-bit number specified by the function that receives the message. It is ANDed together with the message type specified in the message itself; if the result is nonzero, the message is returned by the function for processing.

It is typically much easier to implement the receiver as several tasks, or as a combination of tasks and accept functions, each operating synchronously. For example, one task could handle certain types of messages and another task could handle other types. The message type could determine which messages were sent to the two tasks. Figure 5-4 shows conceptually how messages can be directed to tasks based on message type.

Figure 5-4 A receiver implementation



In Figure 5-4, messages that are conceptually typed with a star are sent to one task. Messages that are conceptually typed with a triangle are sent to another task.

Messages may be received either synchronously, meaning that the receiver blocks until a message arrives, or asynchronously, meaning that the receiver can do other work until it is notified that a message has arrived. For information about asynchronous receives, see the section, "Asynchronous Sends and Receives," beginning on page 88. Messages can also be received by accept functions; for more information, see "Using Accept Functions," beginning on page 87.

Using Accept Functions

An accept function can handle messages that arrive at a port. Only one accept function can be installed in a port at one time. Using an accept function to handle a message is more efficient than using a task to handle it, because a context switch is not required. Accept functions also guarantee that buffers are not copied.

There are several issues you need to consider when using an accept function:

- Accept functions execute concurrently; thus if several messages have been sent to a port but have not yet been replied to, several instances of the accept function will be executing. If the accept function shares a resource such as a data structure, access to the resource must be synchronized.
- An accept function is privileged code; therefore, you must install it from code running in supervisor mode. The best way to do this is to install it from a shared library running in supervisor mode; when the library is loaded, the accept function is installed.
- Accept functions do not receive messages that were sent to the port before the accept function was installed. For that reason you should create the port and install the accept function before creating objects.

You can use the message type to direct a subset of messages to an accept function and allow tasks to handle other messages. For example, you could set up the accept function to handle the most common kinds of messages sent to a server or to handle large messages. Less frequent messages, or those that did not involve large amounts of data being transferred, could be handled by tasks. Thus, an accept function could handle messages that might otherwise result in copying data and allow for maximum concurrency of the senders' tasks.

Asynchronous Sends and Receives

Messages may be both sent and received asynchronously. The functions that handle asynchronous sends and receives allow you to specify the notification mechanism that, in the case of sending, indicates that a reply is available to the sender and, in the case of receiving, indicates that the receiver has a message to process and reply to.

Notification is handled the same way for either the sender or the receiver. You specify one or more of the following methods in the function call to send or receive the message:

- a software interrupt
- an event group and a set of flags
- a kernel queue

For more information about these notification mechanisms, see "Asynchronous Notifications," beginning on page 95.

There are several other issues you should consider when using either asynchronous sends or receives:

- When sending a message asynchronously, you must ensure that the memory locations that contain the message and those reserved for the reply remain accessible. For this reason, you typically cannot use a stack to hold the message or its reply. If you must use the stack, you can specify an option that buffers the message in the kernel; however, if buffer space is not available, the message will be sent as a synchronous message and the sender will block until a reply is received.
- You can cancel asynchronous sends and receives. If you cancel an asynchronous send, the message is removed from the object or the port if it is waiting. If it has already been received, a kernel message is sent that indicates the message should be dropped. If you cancel an asynchronous receive, no further action is necessary.

Replying to Messages

Each message that arrives at a port must be replied to. The ID of the message specifies the target of the reply; the reply is sent to a message, not to an object. The contents of the reply are used to set values in the function that sent the message:

- The reply buffer and buffer length specify the data that is destined for the sender's reply buffer. The messaging service determines how the transfer is to be made, which depends, in part, on the options the sender specified.
- The status of the transaction. The server and client software agree on the meaning of the status; the messaging service simply passes the value back as the return code for synchronous sends or as a value in a kernel notification record for asynchronous sends.

CHAPTER 6

Other Services

Contents

This chapter identifies other kernel and operating services. These services include

- the system registry
- timing services
- notification services
- interspace block copy

System Registry

The **system registry** is an operating system service that allows you to store well-known names so that software can retrieve values based on them. A **well-known name** is a name that software knows to use; of course, you must explicitly identify the name to use to the software. The system registry is a data structure that resides in global memory. Data in the registry is not persistent; it is re-created whenever the system is restarted.

For example, a mail server could put its well-known name in the registry when it starts to execute. A client that wants to use the server could look up the name and obtain information allowing it to send messages to an object associated with the mail server's port. (For information about why you might need to use the system registry for messaging, see the section "Setting Up the Messaging Service" on page 83.)

The well-known name can be any C-style character string up to 255 bytes long. Names in the registry must be unique. You need to avoid placing a name in the registry that could conflict with another name that might be placed there later. You should also remove the name when it is no longer valid, for example, when the server associated with the name is closed down.

Once it the registry, your software can look up the name. The software must specify the name exactly as it exists in the registry. An array of bytes is returned. The bytes can contain any values, such as an object ID in the previous example. You must establish the rules for interpreting these values.

Timing Services

The timing services enable the precise measurement of elapsed time. The timing services allow tasks to suspend their execution until a given time or to cause a specified subroutine to be called at a given time. The following sections describe

- how elapsed time is measured
- how timing services can be used to control task execution
- how timers can be used for notification

Measuring Elapsed Time

Measurement of elapsed time is done by obtaining the time before and after the event to be timed. The difference of these two values indicates the elapsed time of the event. In this context, time refers to the 64-bit absolute time count that is maintained by the kernel. The count is set to zero by the kernel during its initialization at system start-up time. Conversion routines are provided in a shared library to convert from absolute time to 64-bit nanoseconds or 32-bit durations.

Suspending Task Execution

A given task can suspend its execution until a specified time in the future. This process is called *delaying*. When this time is reached, the task again becomes eligible for execution. The task does not actually execute until it is scheduled for execution according to its priority and the priorities of the other eligible tasks. In any case, the task never executes prior to the specified time.

When a task uses a delay service, it can specify the time, in relative or absolute terms, at which it should resume execution. Relative times indicate that execution should resume, for example, 5 minutes from now. Absolute times indicate that execution should resume, for example, at 3 o'clock. Absolute times are a bit more cumbersome to use but allow periodic timing with no long-term drift.

Using Asynchronous Timers for Notification

Asynchronous timing services cause notification at a given time. Asynchronous timers always specify absolute expiration times, which allows you to use them to do drift-free timing.

One of the asynchronous notification methods, described on page 95 can be used. The notification can be delivered in any or all of three ways. First, one or more event flags within a single event flag group can be set. Second, a queue can be notified. Third, a specified subroutine can be run as a software interrupt.

Note
These are the same ways that notification for asynchronous sends and receives of messages can be delivered. For more information about asynchronous notification techniques, see "Asynchronous Notifications," beginning on page 95. ◆

Once set, an asynchronous timer remains in effect until it is either canceled or expires. A timer can be canceled, using the ID of the timer returned by the kernel when the timer was set, at any time prior to expiration. Expiration of the timer causes the notification to be delivered.

Notification Services

Notification services include asynchronous notifications via kernel queues, event groups, and software interrupts, and broadcast notifications (which by their nature are also asynchronous) from the system. The following sections discuss:

- asynchronous notifications
- system notifications

Asynchronous Notifications

Asynchronous notifications allow your software to be notified whenever an event occurs. There are three ways in which software can be notified:

- You can specify a software interrupt to deliver to a task when an operation completes execution. For information about software interrupts, see "Software Interrupt Handlers," beginning on page 29.
- You can specify an event group and flags to be set on completion of an operation. When the operation completes, waiting tasks can become unblocked. For information about event groups, see "Event Groups," beginning on page 71.
- You can specify a kernel queue. A notification is placed in the queue when an operation occurs. For more information about kernel queues, see "Kernel Queues," beginning on page 73.

For example, the messaging service provides asynchronous sends and receipts of messages. You can specify the notification mechanism when you send a message or set up to receive one. For more information about messages and asynchronous sends and receives, see "Asynchronous Sends and Receives," beginning on page 88.

System Notification

System notification is an operating system service that allows a piece of software to broadcast a notification about a change in the state of the system. A **broadcast** is a notification that is sent to a potentially wide audience. The audience is software that can then respond to the notification. For example, a device driver could use system notification to announce a change in the status of the hardware for which it is responsible. Software using the device could then take action based on the notification.

The software that initiates the broadcast is called a **producer**. Software that is interested in the particular notification is called a **consumer**. Between a producer and its consumers are distributors. A **distributor** represents a service to which producers send notifications. It is responsible for directing the notification to consumers.

A producer determines which distributor handles the kind of event being produced. A distributor can handle different kinds of events; however, any specific kind of event can be handled by only one distributor. By default, there is only one distributor, which handles all the different kinds of events.

When a producer sends a notification to a distributor, it specifies the subject of the notification. Consumers subscribe to a service on the basis of the kind of notification and the subject. The distributor for the service sends a notification

to a consumer if the consumer has subscribed to it, in other words, if the kind of notification and the subject of the notification match.

System notification is built on kernel queues. A distributor is responsible for placing a notification in each consumer's queue, and each consumer is responsible for handling the notification. (For information about kernel queues, see "Kernel Queues," beginning on page 73.)

Producers can produce notifications either synchronously or asynchronously. Producers of synchronous notifications are blocked until the notification is received by the distributor. Consumers can also receive notifications either synchronously or asynchronously. Synchronous consumers block until a notification arrives.

System notification can only be used by software executing in the task-level execution environment. (For information about execution environments, see "Performance and Software Structure," beginning on page 34.)

Interspace Block Copy

The kernel provides a function that copies data between address spaces. While not strictly a interprocess communications service, you can use the interspace block copy function when you need to update data in another address space. For example, the messaging service uses interspace block copy when sending a message by value. The function checks that the data to be copied is accessible and returns a status value.

When you use the interspace block copy function, you must be aware of these issues:

- Because this function does not perform synchronization, you must synchronize access between software executing in different address spaces and potentially in different execution environments as well.
- Specifying a "bad" but accessible (perhaps random) address for the destination could have disastrous consequences. You should obtain the destination address from software in the destination address space, thus placing responsibility for the destination address with the receiver of the data.

CHAPTER 7

System 7 Compatibility

Contents

This chapter identifies System 7 services that you may be using that could affect the way your software runs under Copland.

Compatibility With System 7 Services

The following sections briefly describe System 7 services that have implications for kernel and operating system software.

Threads

The Thread Manager implements a cooperative threads capability in System 7 applications. You can continue to use threads, however, they only provide threads of execution for the currently executing application and do not enter into the kernel's scheduling algorithm.

You should seriously consider replacing threads by tasks, especially if you want to use services such as locking within an application. For example, if you attempt to acquire a lock from within a thread and block, the entire application (if it is a single primary task) blocks, not just the thread. Another thread cannot execute to release the lock.

High-Level Events

Primary tasks can use high-level events; however, secondary tasks cannot send high-level events other than Apple events. In most cases you should use Apple events to communicate with other applications.

PPC Toolbox Services

Primary tasks can use the PPCBrowser mechanism and the PPC Toolbox functions. Secondary tasks cannot use the PPCBrowser mechanism but can call all other PPC Toolbox functions. However, before using the PPC Toolbox to send data between tasks, you should consider whether any of the other interprocess communication methods are more appropriate for your needs.

System 7 Hardware Interrupt Level and Deferred Tasks

In System 7, I/O completion routines, VBLs, and Time Manager tasks run at either hardware interrupt level or as deferred tasks. These are now run at user mode task level instead. Therefore, most code on the system, including completion routines, is run at task level, and less is run at interrupt level or with interrupts disabled. This strategy provides the following benefits:

- The kernel doesn't allow application code to be run at interrupt level, because the code could cause page faults. Interrupt time is minimized. Because applications never disable interrupts, interrupt latency is minimized and the time available for applications to run is increased. **Interrupt latency** is the time between when an interrupt is generated and the associated interrupt handler is executed.
- Page faults become invisible to application code, including completion routines.

System Extensions

Copland does *not* support the use of system extensions of type `'INIT'`. To support replacements for software of this type, Copland provides enhanced system services, many of which also eliminate the need for the patching that your application might have done in System 7.

Patching

The Patch Manager ensures that the trap patching API defined with System software 7 is supported in the Copland operating system. Support is limited to local patching. Table 7-1 lists the names of the functions that continue to be supported in Copland.

Table 7-1 Programmatic patching calls supported under Copland

Function
`GetTrapAddress`
`SetTrapAddress`
`NGetTrapAddress`

Table 7-1 Programmatic patching calls supported under Copland

Function
`NSetTrapAddress`
`GetOSTrapAddress`
`SetOSTrapAddress`
`GetToolTrapAddress`
`SetToolTrapAddress`
`GetToolboxTrapAddress`
`SetToolboxTrapAddress`
`GetTrapVector`

These functions are fully described in the Trap Manager chapter of *Inside Macintosh: Operating System Utilities*.

Memory Management

Copland applications use the Memory Manager when allocating and releasing memory space in their heaps, which are still used by the Toolbox. The Copland Memory Manager fully supports the System 7 Memory Manager APIs. The calls work the same way as they do if System 7 virtual memory is turned off.

Due to the number of changes in the addressing model introduced by Copland, software that circumvents the System 7 Memory Manager functions may require revision to run compatibly with Copland's virtual memory.

A-Trap Support

Because it is fragment-based, PowerPC native code compiled for System 7 is supported by the Copland runtime environment. All Code Fragment Manager–based calling conventions in Copland remain consistent with those of System 7. The Copland runtime environment also supports System 7 software based on the use of the A-trap table—developed for the original 68K runtime environment—by running this software under emulation on the PowerPC processor.

Index

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W