1

Mach Concepts

Mach, the operating system of all NeXT computers, was designed by researchers at Carnegie Mellon University (CMU).   Mach is based on a simple communication-oriented kernel, and is designed to support distributed and parallel computation while still providing UNIX 4.3BSD compatibility.

The NeXT Mach operating system is a port of CMU Release 2.0, with additional features both from NeXT and from later versions of CMU Mach.   NeXT-only features include the Bootstrap Server and loadable kernel servers. Features from CMU Release 2.5 and beyond include scheduling and some details of messaging.

Mach consists of the following components:

- A small, extensible system kernel that provides scheduling, virtual memory, and interprocess communications; the kernel exports a small number of abstractions to the user through an integrated interface.
- Operating system support environments that provide distributed file access, transparent network interprocess communication, remote execution facilities, and UNIX 4.3BSD emulation.   Many traditional operating system functions can be implemented by user programs or servers outside the kernel.

Although Mach's design is conceptually unlike that of UNIX 4.3BSD, it maintains UNIX 4.3BSD compatibility. Mach system calls are upwardly compatible with those of UNIX 4.3BSD, and Mach supports UNIX 4.3BSD commands.   This compatibility is transparent to user programs and requires no special libraries or other utilities. Most programs that operate under UNIX 4.3BSD operate under Mach without modification, after being recompiled.

Mach provides the following features not found in UNIX 4.3BSD:

- Multiple tasks, each with a large, paged virtual memory space
- Multiple threads of execution within each task, with a flexible scheduling facility
- Flexible sharing of memory between tasks
- Efficient and consistent message-based interprocess communication
- Memory-mapped files
- Transparent network extensibility
- A flexible, capability-based approach to security and protection
- Support for multiprocessor scheduling

Mach is sometimes referred to as an object-oriented operating system because it provides most services through user-level programs accessible by a consistent system of message passing.   It's important, however, to distinguish between Mach objects and messages and the Objective C objects and messages used in higher-level software kits such as the Application Kit™.   Mach objects and messages are distinct from those used in the kits.   Kit objects can, however, communicate with the operating system by sending Mach messages to Mach objects or by using the standard UNIX system call interface.

This chapter describes both the Mach kernel and user-level programs that interact with it, but doesn't attempt to redocument standard features of UNIX 4.3BSD.   (See the ªSuggested Readingº section at the end of this manual for information about available UNIX 4.3BSD documentation.)   Individual Mach functions are described in detail in Chapter 4, ªMach Functions,º and summarized in the *NEXTSTEP Programming Interface Summary*.

Design Philosophy

Several factors were considered in choosing an operating system for NeXT computers. It was important that the operating system be:

- Multiuser and multitasking
- Network-compatible
- An excellent program-development environment
- Well-represented in the university, research, and business communities
- Extensible and robust
- Capable of providing room for growth and future extensions

Although a standard version of the UNIX operating system would have satisfied many of these criteria, NeXT wanted an operating system offering better performance and a better foundation for future extensions. Mach, with its UNIX 4.3BSD compatibility and improved system design, provided these.

UNIX 4.3BSD compatibility is important because as a multitasking, multiuser operating system, the UNIX environment has gained wide acceptance in many fields, particularly education. Since the creation of the UNIX operating system in 1969, many hours have been spent testing, improving, and extending its features. Currently the UNIX environment is considered one of the best for developing both small and large applications.

However, the success and longevity of the UNIX operating system have exacted their own costs. Many of the features that made the UNIX operating system popular have disappeared in the quest for functionality beyond the scope of the original design. During two decades, the UNIX operating system has grown from a system designed for 16-bit minicomputers without paged memory or networking, to a system that supports multiprocessor mainframes with virtual memory and support for both local and wide-area networks. As a result of these extensions, the UNIX kernel (originally attractive to developers because of its small size, handful of system calls, and ease of modification) has grown to immense proportions.

As new features have been added to the kernel, its size and complexity have grown to the point where its underlying conceptual structure is obscured. Over time, programmers have added multiple routines that perform similar services for different kernel features. The complexity added by each of these extensions ensures that future kernel extensions will be based on an even less sound understanding of what already exists. The result is a system whose complex internal state and interactions make it very difficult to extend, debug, and configure.

Not only has the UNIX kernel grown more complex as new features have been added, so has the interface presented to programmers who would like to make use of these features. For example, current UNIX systems provide an overwhelming variety of interprocess communication (IPC) facilities, including pipes, named pipes, sockets, and message queues. Unfortunately, none of these facilities is general enough to replace the others. As a result, the programmer must understand not only how to use a variety of IPC facilities, but also the tradeoffs involved in choosing one over another.

While retaining UNIX 4.3BSD functionality, Mach departs from current UNIX design and returns to the tenets on which the UNIX operating system was originally built. Foremost among these is the idea that the kernel should be as small as possible, containing only a set of conceptually simple, yet powerful, primitive functions that programmers can use to construct more complex objects.

Mach is designed to put most services provided by the current UNIX kernel into independent user-level programs, with the Mach kernel itself providing only the most basic services:

- Processor scheduling
- Interprocess communication
- Management of virtual memory

These services and others are accessed through a single form of IPC, regardless of whether they're provided by the kernel or by user-level programs. Modularity and a consistent pattern of IPC simplify the interface presented to the programmer. For example, a network expert can implement a new protocol without having to understand or modify other subsystems in the operating system.

Modularity has other advantages as well. Moving functionality to user-level programs makes the kernel smaller and therefore easier to comprehend and debug. Another advantage is the ability to use standard debuggers and other tools to develop new system services rather than having to use special, less powerful tools. Also, configuring the system is simply a matter of choosing which user-level services to initiate, rather than building and linking a new kernel.

The movement of Mach toward providing most operating system features as user-level processes is an evolutionary one. Currently, Mach supports some features within the kernel while others exist at the user level. Although Mach will change as it evolves, its developers are committed to maintaining UNIX 4.3BSD compatibility at each stage of development. If you design your programs to run under UNIX 4.3BSD, they'll run under current and subsequent releases of the Mach operating system. However, if you choose to take advantage of features unique to Mach, future releases of the operating system may require you to modify and recompile some of your programs.

The Mach Kernel

Mach minimizes kernel size by moving most kernel services into user-level processes. The kernel itself contains only the services needed to implement a communication system between various user-level processes. The kernel exports several abstractions to users, including tasks, threads, ports, and messages.

The functionality of the Mach kernel can be divided into the following categories:

- Task and thread creation and management facilities
- Port management facilities
- Basic message functions and support facilities
- Virtual memory management functions
- Scheduling functions

Descriptions of these areas of functionality are provided in the following sections. Messages are described in detail in Chapter 2, ªUsing Mach Messages.º

Mach Tasks and Threads

Mach splits the traditional UNIX notion of a process into two abstractions, the task and the€thread:

- A *task* is the environment within which program execution occurs. It's also the basic unit of resource allocationÐeach task includes a paged virtual address space and port rights that protect access to system resources such as processors, communication capabilities, and virtual memory. The task itself performs no computation; rather, it's a framework for running threads.

- A *thread* is the basic unit of execution. It's a lightweight process executing within a task, and consists solely of the processor state (such as program counter and hardware registers) necessary for independent execution. Each thread executes within the context of a single task, though each task may contain more than one thread. All threads within a task share the virtual memory address space and communication rights of that task.

The task is the basic unit of protectionÐall threads within a task have access to all that task's capabilities, and aren't protected from each other.

A traditional UNIX process is represented in Mach as a task with a single thread of execution. One major difference between a UNIX process and a Mach task is that creating a new thread in a task is faster and more conservative of system resources than creating a new UNIX process. Creating a new UNIX process involves making a copy of the parent task's address space, but threads share the address space of their task.

Threads are the basic unit of scheduling. On a multiprocessor host, multiple threads from€one task may be executing simultaneously within the task's one address space. A€thread may be in a *suspended* state (prevented from running), or in a *runnable* state (that€is,€either currently running or scheduled to run). A nonnegative suspend count is associated with each thread. The suspend count is 0 for runnable threads and positive for€suspended threads.

Tasks can be suspended or *resumed* (made runnable) as a whole. A thread can execute only when both it and its task are runnable.

Multiple threads executing within a single task are useful if several program operations need to execute concurrently while accessing the same data. For example, a word processing application could be designed as multiple threads within a single task. The main thread of execution could provide the basic services of the program: formatting text, processing user requests, and so on. Another thread could check the spelling of each word as it's typed in. A third thread could modify the shape of the cursor based on its position within the text window. Since these threads must have access to the same data and should execute concurrently, Mach's design is particularly advantageous.

In addition, threads are well adapted for use with computers that incorporate a multiprocessor architecture. With some multiprocessor machines, individual threads can execute on separate processors, vastly improving overall application performance.

To create and use threads in an application, you should use the C-thread functions. C threads are described later in this chapter; each C-thread function is described in detail in Chapter 4.

Task and Thread Ports

Both tasks and threads are represented by ports. (Ports in Mach are message queues; they're described in the following section.) The task port and the thread port are the arguments used in Mach function calls to identify to the kernel which task or thread is to be affected by the call. The two functions **task_self()** and **thread_self()** return the task and thread ports of the currently executing thread.

Tasks can have access to the task and thread ports of other tasks and threads. For example, a task that creates another task or thread gets access to the new task port or thread port. Also, any thread can pass access to these ports in a message to another thread in the same or a different task.

Having access to a task or thread port enables the possessor to perform Mach function calls on behalf of that task or thread. Access to a task's port indirectly permits access to all threads within that task with the **task_threads()** call; however, access to a thread's port doesn't imply access to its task's port.

The task port and thread port are often called *kernel ports*. In addition to the kernel ports, tasks and threads have a number of special ports associated with them. These are ports that the kernel must know about to communicate with the task or thread in a structured manner.

A task has three ports associated with it, in addition to its kernel port:

· *Notify port*ÐThe port on which the task receives messages from the kernel advising it of changes in port access rights and of the status of messages it has sent. For example, if a thread is unsuccessful in sending a message to another thread's port, its notify port will contain a status message stating that the port has been intentionally destroyed, that the port's task no longer exists, or that there has been a network failure. The task can get this port's value from the function **task_notify()**.

  Note that if a task's notify port is PORT_NULL, no notification messages are generated. This port is set to PORT_NULL at task creation, so a task that wants to receive notifications must explicitly set its notify port with the function **task_set_special_port()**.

· *Exception port*ÐThe port on which the task receives messages from the kernel when an exception occurs. *Exceptions* are synchronous interruptions to the normal flow of program control caused by the program itself. They include illegal memory accesses, protection violations, arithmetic exceptions, and hardware instructions intended to support emulation, debugging, and error detection. Some of these exceptions are handled transparently by the operating system, but some must be reported to the user program. A default exception port is inherited from the parent at task creation time. This port can be changed by the task or any one of its threads in order to take an active role in handling exceptions.

- *Bootstrap port*ÐThe port to which a new task can send a message that will return any other system service ports that the task needs (for example, a port to the Network Name Server). A default bootstrap port is inherited from the parent at task creation. This is the one port that the kernel doesn't actually use; it just makes it available to a new task.

A thread has two ports, in addition to its kernel port:

- *Reply port*ÐUsed in Mach remote procedure calls (remote procedure calls are described in Chapter 2). The **thread_reply()** function returns the reply port of the calling thread.

- *Exception port*ÐThe port to which the kernel sends exceptions occurring in this thread. This port is set to PORT_NULL at thread creation and can be set subsequently with the function **thread_set_exception_port()**. As long as the thread exception port is PORT_NULL, the task exception port is used instead.

Customarily, only threads within a task manipulate that task's state, but this custom isn't enforced by the Mach kernel. A debugger task, for example, can manipulate the state of the task being debugged by getting the task's kernel port and using it in Mach function calls.

Mach Ports and Messages

In Mach, communication among operating system objects is achieved through messages. Mach messaging is implemented by three kernel abstractions:

- *Port*ÐA protected communication channel (implemented as a finite-length message queue) to which messages may be sent and logically queued until reception. The port is also the basic object reference mechanism in Mach; its use is similar to that of object references in an object-oriented system. That is, operations on objects are requested by sending messages to and from the ports that represent them. When a task is created, a port that represents the task is simultaneously created. When the task is destroyed, its port is also destroyed.

- *Port set*ÐA group of ports, implemented as a queue combining the message queues of the constituent ports. A thread may use a port set to receive a message sent to any of several ports.

- *Message*ÐUsed to communicate between objects; the message is passed to an object by being sent to the port that represents the object. Each message is a data stream consisting of two parts: a fixed-length header and a variable-length message body composed of zero or more typed data objects. The header contains information about the size of the message, its type, and its destination. The body contains the content (or a pointer to the content) of the message. Messages may be of any size, and may contain in-line data, pointers to data, and capabilities for ports. A single message may transfer up to the entire address space of a task.

Message passing is the primary means of communication both among tasks and between tasks and the kernel. In fact, the only way one object can communicate with another object is by sending a message to that object's port. System services, for example, are invoked by a thread in one task sending a message to another task that provides the desired service. The only functions implemented by system traps are those directly concerned with message communication; all the rest are implemented by messages to the kernel port of a task.

Threads within a single task also use messages and ports to communicate with each other. For example, one thread can suspend or resume the execution of another thread by sending the appropriate message to the thread's port. A thread can also suspend or resume the execution of all threads within another task by sending the appropriate message to the task's port.

The indirection provided by message passing allows objects to be arbitrarily placed in the network without regard to programming details. For example, a thread can suspend another thread by sending a suspend message to the port representing that other thread even if the request is initiated on another node in a network. It's thus possible to run varying system configurations on different classes of machines while providing a consistent interface to all resources. The actual system running on any particular machine is more a function of its servers than its kernel.

Port Access Rights

Communication between objects is protected by a system of *port access rights*. Access rights to a port consist of the ability to send to or receive from that port. For example, before a task can send a message to a port, it must gain send rights to that port. Before a message can be received, a task must gain receive rights to the port containing the message.

The port access rights operate as follows:

· *Send access* to a portÐImplies that a message can be sent to that port. If the port is destroyed during the time a task has send access, the kernel sends a message to that task's notify port indicating that the port has disappeared. For loadable kernel servers, this notification message isn't sent unless the server has requested notification by calling **kern_serv_notify()**.

· *Receive access* to a portÐAllows a message to be dequeued from that port. Only one task may have receive access for a given port at a time; however, more than one thread within that task may concurrently attempt to receive messages from a given port. When the receive rights to a port are destroyed, that port is destroyed and tasks holding send rights are notified. Receive access implies send rights.

Although multiple tasks may hold send rights to the same port, only one task at a time may hold receive rights to a port.

A thread's right of access is identical to that of the thread's task. Also, when a thread creates a port, send and receive rights are accorded to the task within which the thread is executing. Thus, all threads within the task have equivalent access rights to the new port. Thereafter, any thread within the task can deallocate any or all of these rights, or transfer them to other tasks. The transfer of port rights is accomplished through the Mach messaging system: Access to a port is gained by receiving a message containing a port capability (that is, a capability to either send or receive messages).

Port access rights can be passed in messages. The rights are interpreted by the kernel and transferred from the sender to the kernel upon message transmission and to the receiver upon message reception. Send rights are kept by the original task as well as being transmitted to the receiver task, but receive rights are removed from the original task at the time of the send, and appear in the user task when the receive is done.

During the time between a send and receive, the kernel holds the rights, and any messages sent to the port will be queued waiting for a new task to receive on the port. If the task that was intended to receive the rights dies before it receives them, the rights are handled as though the task had received them before it died.

The type usually used for ports is **port_t**. However, ports can also be referred to as the equivalent types **port_name_t** and **port_all_t**. The **port_name_t** type implies that no port access rights are being transferred; the port is merely being referred to by its name. The **port_all_t** type implies that all rights (both send and receive) for a port are being transferred.

Port Sets

Conceptually, a port set is a bag holding zero or more receive rights. A port set allows a thread to block while waiting for a message sent to any of several ports. A port may be a member of no more than one port set at any time, and a task can have only one port set.

A task's port set right, created by **port_set_allocate()**, allows the task to receive a message from the port set with **msg_receive()** and manipulate the port set with **port_set_add()**, **port_set_remove()**, **port_set_status()**, and **port_set_deallocate()**. Unlike port rights, a port set right can't be passed in messages.

Port set rights usually have the type **port_set_name_t**, which is equivalent to **port_name_t**.

Port Names

Every task has its own port name space, used for port and port set names. For example, one task with receive rights for a port may know the port by the name 13, while another task with send rights for the same port may know it by the name 17. A task has only one name for a port, so if the task with send rights named 17 receives another message carrying send rights for the same port, the arriving rights will also be named 17.

Typically, these names are small integers, but this is implementation dependent. When a task receives a message carrying rights for a new port, the Mach kernel is free to choose any unused name. The **port_rename()** call can be used to change a task's name for a port.

Port Queues

Messages that are sent to a port are held there until removed by a thread. The queue associated with a port is of finite length and may become full. If an attempt is made to send a message to a port that's temporarily full, the sending thread has a choice of three alternatives:

· By default, the sender is suspended until it can successfully transmit the message.

· The sender can have the kernel hold the message for later transmission to the currently full port. If the sender selects this action, it can't transmit further messages to the port (nor can it have the kernel hold additional messages for the port) until the kernel notifies it that the port has received the initial message.

· The attempt to send a message to a full port can simply be reported to the sender as an€error.

Extended Communication Functionality

The kernel's message-based communication facility is the building block on which more complicated facilities may be constructed; for example, it's the underlying communication mechanism for the Mach exception-handling facility. Two properties of the Mach communication facility simplify the process of extending the functionality of systems based on it:

· IndependenceÐA port is an independent entity from the tasks that use it to communicate. Port rights can be exchanged in messages, and are tracked by the kernel to maintain protection.

· Network transparencyÐAs described in the following section, user-level network message servers transparently extend the Mach communication facility across a network, allowing messages to be sent between tasks on different computers. The forwarding process is invisible to both the sender and the receiver of the message.

This combination of independence and network transparency enables Mach to support parallel and distributed architectures with no change to the operating system kernel. These properties of the communication facility also simplify the incorporation of new operating system functionality, because user-level programs can easily be added to the existing kernel without the need to modify the underlying kernel base.

Although messaging is similar to UNIX 4.3BSD stream sockets in that it permits reliable, kernel-mediated communication between tasks, messaging has a much more fundamental role within Mach. Whereas UNIX processes obtain system services through a variety of interfaces (for example, the **open()** system call for files, the **socket()** and **bind()** system calls for network connections, and numerous access protocols for user-level services), Mach provides all services through messaging. Because of this consistency of interprocess communication, the Mach operating system can easily be extended to incorporate new€features.

As an alternative to messaging, Mach also supports interprocess communication using shared memory. However, if you use shared memory for interprocess communication, you're responsible for synchronizing the transmission and reception of the message. With the Mach messaging system, Mach itself schedules the transmission and reception of messages, thereby ensuring that no message is read before it's been sent in its entirety.

Messaging in a Network Environment

Mach's object-oriented design is well-suited for network operation. Messages may be sent between tasks on different computers just as they're sent between tasks on the same computer. The only difference is the transparent intervention of a new user-level object, the *network server*.

Programs called network servers act as intermediaries for messages sent between tasks on€separate computers. Each network server implements *network ports* that represent ports for tasks on remote nodes. A unique *network* *port identifier* is used to distinguish each network port.

A message addressed to a remote port is first received at the local network port that represents the remote port. The network server, upon receiving the message, translates it into a form compatible with the network protocol and then transmits the message to the counterpart network server on the destination node. The destination server decodes the message, and determines its ultimate destination from the network port identifier in the message. Finally, the destination network server dispatches the message to the local port to which it was addressed.

This network messaging process is transparent to the sender; all routing services are provided by the network server.

Mach Virtual Memory Management

Each Mach task receives a 4-gigabyte virtual address space for its threads to execute in. This address space consists of a series of mappings between ranges of memory addressable to the task and memory objects. Besides accommodating the task and its threads, this space serves as the basis of the Mach messaging system and allows space for memory-mapped files.

A task can modify its address space in several ways. It can:

· Allocate a region of virtual memory (on a page boundary).

· Deallocate a region of virtual memory.

· Set the protection status of a region of virtual memory.

· Specify the inheritance of a region of virtual memory.

· Create and manage a memory object that can then be mapped into the space of another€task.

The only restriction imposed by Mach on the nature of the regions that may be specified for virtual memory operations is that they must be aligned on system page boundaries. The size in bytes of a virtual memory page is contained in the **vm_page_size** variable.

Demand Paging

A NeXT computer's memory management hardware is responsible for mapping sections of the virtual memory space into pages of physical memory as needed. The process it uses to decide which virtual pages map to which physical pages is known as *demand paging*.

While a task is executing, only the page of memory containing the addresses referenced by the active thread must reside in physical memory. If the thread references an address not contained in a page of physical memory, the kernel requests the appropriate pager to read in the needed page from storage. Then, a NeXT computer's memory management unit maps the referenced virtual page onto this new physical page of memory.

If there are no further free pages of physical memory available, the Mach kernel makes room by requesting the pager to copy the least recently used page to the paging file on the disk. The kernel then reassigns the newly freed page of memory.

Mach's paged virtual address space makes it possible to run extremely large applications on a NeXT computer. With all but the largest applications, you can continue to allocate memory without concern for exceeding the system's capacity, although to prevent unnecessary performance degradation, you should deallocate memory that's no longer needed.

Inheritance and Protection of Memory

The Mach virtual memory management system also streamlines the creation of a new task (the child) from an existing task (the parent), an operation similar to forking a UNIX process. Traditionally, under the UNIX operating system, creating a new process entails creating a copy of the parent's address space. This is an inefficient operation since often the child task, during its existence, touches only a portion of its copy of the parent's address space. Under Mach, the child task initially shares the parent's address space and copying occurs only when needed, on a page-by-page basis.

A region of an address space represents the memory associated with a continuous range of addresses, marked by a starting address and an ending address. Regions consist of pages that have different protection or inheritance characteristics. The Mach kernel extends each region to include the entire virtual memory pages that contain the starting and ending addresses in the specified range.

Inheritance and protection are attached to a task's address space, not the physical memory contained in that address space. Tasks that share memory may specify different protection or inheritance for their shared regions.

Inheritance

A task may specify that pages of its address space be inherited by child tasks in three ways:

- CopyÐPages marked as copy are logically copied by value, although for efficiency copy-on-write techniques are used. This means the first time the child task attempts to write to shared memory, a protection fault occurs. The kernel responds to this fault by making a copy, for the child task, of the page being written. This is the default mode of inheritance if no mode is specified.
- SharedÐPages specified as shared can be read from and written to by both the parent and child.
- NoneÐPages marked as none aren't passed to a child. In this case, the child's corresponding address is left unallocated.

Inheritance may be specified globally or on a page-by-page basis when a task is forked. Inheritance may be changed at any time; only at the time of task creation is inheritance information used.

Copy-on-write sharing between unrelated tasks is typically the result of large message transfers. An entire address space may be sent in a single message with no actual data copy operations performed.

Currently the only way two Mach tasks can share the same physical memory is for one of the tasks to inherit shared access to memory from a parent.

Protection

Besides specifying page inheritance attributes, a task may assign protection values to protect the virtual pages of its address space by allowing or preventing access to that memory. Protection values are a combination of read, write, and execute permissions.

By default, when a child task inherits memory from a parent, it gets the same protection on that memory that its parent had.

Like inheritance, protection is specified on a per-page basis. For each group of pages there exist two protection values: the current and the maximum protection. The current protection is used to determine the access rights of an executing thread, and the maximum protection specifies the maximum value that the current protection may take. The maximum value may be lowered but not raised. If the maximum protection is lowered to a level below the current protection, the current protection is also lowered to that level.

For example, a parent task may create a child task and set the maximum protection value for some pages of memory to read-only. Thereafter, the parent task can be assured that the child won't be able to alter the information in those pages.

Interprocess Communication

Mach virtual memory management provides an efficient method of interprocess communication. Messages of any size (up to the limits imposed by the virtual address space) can be transferred between tasks by revising the mapping from the virtual address space of a process to physical address space. This is accomplished by mapping an unused portion of the virtual address space of the receiving process onto the addresses of the sender's message.

The efficiency of this method can be appreciated more fully when compared to the standard UNIX method. Under the UNIX operating system, a message must be physically copied from the address space of the sender into the address space of the kernel. From there, the message is copied into the address space of the receiver.

Memory-Mapped Files

Memory-mapped files are a further benefit of the Mach virtual memory system. Under Mach, all or part of a disk file can be mapped onto a section of virtual memory. A reference to a position within this section is equivalent to a reference to the same position in the physical file. If that portion of the file isn't currently in memory, a page fault occurs, prompting the kernel to request the file system to read the needed section of the file into physical memory. From the point of view of the process, the entire file is in memory at€once.

With Mach, the use of memory-mapped files is optional and currently only supports reading files. Mach also supports the standard UNIX **read()**, **lseek()**, and **write()** system calls.

Paging Objects

A *paging object* is a secondary storage object that's mapped into a task's virtual memory. Paging objects are commonly files managed by a file server, but as far as the Mach kernel is concerned, a paging object may be implemented by any port that can handle requests to read and write data.

Physical pages in an address space have paging objects associated with them. These objects identify the backing storage to be used when a page is to be read in as the result of a reference or written to in order to free physical memory.

Virtual Memory Functions

The Mach kernel provides a set of functions to allow a programmer to manipulate the virtual address space of a task. The two most fundamental ones are **vm_allocate()** to get new virtual memory and **vm_deallocate()** to free virtual memory. The programmer also has available the UNIX functions **malloc()**, **calloc()**, and **free()**, which have been reimplemented to use **vm_allocate()** and **vm_deallocate()**.

In addition to memory explicitly allocated using **vm_allocate()**, memory may appear in a task's address space as the result of a **msg_receive()** operation.

The decision to use one allocation method rather than another should be based on several factors. The **vm_allocate()** function always adds new, zero-filled virtual memory in page-aligned chunks that are multiples of the page size. The **malloc()** function allocates approximately the size asked for (plus a few bytes) out of a preallocated heap. The **calloc()** function is the same as **malloc()** except that it zeros the memory before returning it. Both **malloc()** and **calloc()** are library subroutine calls; **vm_allocate()** is a Mach kernel function, which is somewhat more expensive.

The most obvious basis on which to choose an allocation function is the size of the desired space. One other consideration is the desirability of page-aligned storage. If the memory that's allocated is to be passed *out-of-line* in a message (referred to by a pointer in the message), it's more efficient if it's page-aligned.

Note that it's essential that the correct deallocation function be used. If memory has been allocated with **vm_allocate()**, it must be deallocated with **vm_deallocate()**; if it was allocated with **malloc()** it must be deallocated with **free()**. Memory that's received out-of-line from a message has been allocated by the kernel with **vm_allocate()**.

Program Examples: Virtual Memory

The following three examples demonstrate various aspects of the use of virtual memory functions in C programs.

The first program, **vm_read.c**, demonstrates the use of **vm_allocate()**, **vm_deallocate()**, and another virtual memory function called **vm_read()**. First some memory is allocated and filled with data. The **vm_read()** Mach function is then called, with reading starting at the previously allocated chunk. The contents of the two pieces of memory (that is, the one retrieved by **vm_allocate()** and the one by **vm_read()**) are compared. The **vm_deallocate()** function is then used to get rid of the two chunks of memory.

```
#import <mach/mach.h>
#import <stdio.h>

main(int argc, char *argv[])
{
    char          *data1, *temp;
    char          *data2;
    int            i, min;
    unsigned int   data_cnt;
    kern_return_t rtn;

    if (argc > 1) {
        printf("vm_read takes no switches.  ");
        printf("This program is an example vm_read\n");
        exit(-1);
    }

    if ((rtn = vm_allocate(task_self(), (vm_address_t *)&data1,
            vm_page_size, TRUE)) != KERN_SUCCESS) {
        mach_error("vm_allocate failed", rtn);
        printf("vmread: Exiting.\n");
        exit(-1);
    }

    temp = data1;
    for (i = 0; (i < vm_page_size); i++)
        temp[i] = i;
    printf("Filled space allocated with some data.\n");
    printf("Doing vm_read....\n");
    if ((rtn = vm_read(task_self(), (vm_address_t)data1,
            vm_page_size, (pointer_t *)&data2, &data_cnt))
            != KERN_SUCCESS) {
        mach_error("vm_read failed", rtn);
        printf("vmread: Exiting.\n");
        exit(-1);
    }
    printf("Successful vm_read.\n");

    if (vm_page_size != data_cnt) {
        printf("vmread: Number of bytes read not equal to number");
        printf("available and requested.\n");
    }
    min = (vm_page_size < data_cnt) ? vm_page_size : data_cnt;

    for (i = 0; (i < min); i++) {
        if (data1[i] != data2[i]) {
            printf("vmread: Data not read correctly.\n");
            printf("vmread: Exiting.\n");
            exit(-1);
        }
    }
    printf("Checked data successfully.\n");

    if ((rtn = vm_deallocate(task_self(), (vm_address_t)data1,
            vm_page_size)) != KERN_SUCCESS) {
        mach_error("vm_deallocate failed", rtn);
        printf("vmread: Exiting.\n");
        exit(-1);
    }

    if ((rtn = vm_deallocate(task_self(), (vm_address_t)data2,
            data_cnt)) != KERN_SUCCESS) {
        mach_error("vm_deallocate failed", rtn);
        printf("vmread: Exiting.\n");
        exit(-1);
    }
}
```

The next program, **vm_copy.c**, demonstrates the use of **vm_allocate()**, **vm_deallocate()**, and **vm_copy()**. First, some memory is allocated and filled with data. Then another chunk of memory is allocated, and **vm_copy()** is called to copy the contents of the first chunk to the second. The data in the two spaces is compared to be sure it's the same, checking **vm_copy()**. The **vm_deallocate()** function is then used to get rid of the two chunks of memory.

```
#import <mach/mach.h>
#import <stdio.h>

main(int argc, char *argv[])
{
    int         *data1, *data2, *temp;
    int          i;
    kern_return_t    rtn;

    if (argc > 1) {
        printf("vm_copy takes no switches.    ");
        printf("This program is an example vm_copy\n");
        exit(-1);
    }

    if ((rtn = vm_allocate(task_self(), (vm_address_t *)&data1,
            vm_page_size, TRUE)) != KERN_SUCCESS) {
        mach_error("vm_allocate failed", rtn);
        printf("vm_copy: Exiting.\n");
        exit(-1);
    }

    temp = data1;
    for (i = 0; (i < vm_page_size / sizeof(int)); i++)
        temp[i] = i;
    printf("vm_copy: set data\n");

    if ((rtn = vm_allocate(task_self(), (vm_address_t *)&data2,
            vm_page_size, TRUE)) != KERN_SUCCESS) {
        mach_error("vm_allocate failed", rtn);
        printf("vm_copy: Exiting.\n");
        exit(-1);
    }

    if ((rtn = vm_copy(task_self(), (vm_address_t)data1, vm_page_size,
            (vm_address_t)data2)) != KERN_SUCCESS) {
        mach_error("vm_copy failed", rtn);
        printf("vm_copy: Exiting.\n");
        exit(-1);
    }
    printf("vm_copy: copied data\n");

    for (i = 0; (i < vm_page_size / sizeof(int)); i++) {
        if (data1[i] != data2[i]) {
            printf("vm_copy: Data not copied correctly.\n");
            printf("vm_copy: Exiting.\n");
            exit(-1);
        }
    }
    printf("vm_copy: Successful vm_copy.\n");

    if ((rtn = vm_deallocate(task_self(), (vm_address_t)data1,
            vm_page_size)) != KERN_SUCCESS) {
        mach_error("vm_deallocate failed", rtn);
        printf("vm_copy: Exiting.\n");
        exit(-1);
    }

    if ((rtn = vm_deallocate(task_self(), (vm_address_t)data2,
            vm_page_size)) != KERN_SUCCESS) {
        mach_error("vm_deallocate failed", rtn);
        printf("vm_copy: Exiting.\n");
        exit(-1);
    }
    printf("vm_copy: Finished successfully!\n");
}
```

The following program, **copy_on_write.c**, demonstrates the use of **vm_inherit()** and copy-on-write memory. A child and parent task share memory, polling this memory to see whose turn it is to proceed.

First, some memory is allocated, and **vm_inherit()** is called on this memory, the variable **lock**. Then more memory is allocated for the copy-on-write test. A fork is executed, and the parent then stores new data in the copy-on-write memory previously allocated, and sets the shared variable signaling to the child that the parent is now waiting. The child, polling the shared variable, sees that the parent is waiting. The child prints the value of the variable **lock** and a value of the copy-on-write memory as the child sees it. The value of **lock** is what the parent set it to be, but the value of the copy-on-write memory is the original value and not what the parent changed it to be. The parent then awakens and prints out the two values once more. The program then ends with the parent signaling the child using the shared variable **lock**.

Typically you wouldn't do this synchronization directly as shown here, but would use C-thread functions (described later in this chapter).

```
#import <mach/mach.h>
#import <stdio.h>

#define NO_ONE_WAIT 0
#define PARENT_WAIT 1
#define CHILD_WAIT 2
#define COPY_ON_WRITE 0
#define PARENT_CHANGED 1
#define CHILD_CHANGED 2
#define MAXDATA 100

main(int argc, char *argv[])
{
    int           pid;
    int          *mem;
    int          *lock;
    kern_return_t ret;

    if (argc > 1) {
        printf("cowtest takes no switches.    ");
        printf("This is an example of copy-on-write \n");
        printf("memory and the use of vm_inherit.\n");
        exit(-1);
    }

    if ((ret = vm_allocate(task_self(), (vm_address_t *)&lock,
            sizeof(int), TRUE)) != KERN_SUCCESS) {
        mach_error("vm_allocate failed:", ret);
        printf("Exiting with error.\n");
        exit(-1);
    }

    if ((ret = vm_inherit(task_self(), (vm_address_t)lock,
            sizeof(int), VM_INHERIT_SHARE)) != KERN_SUCCESS) {
        mach_error("vm_inherit failed:", ret);
        printf("Exiting with error.\n");
        exit(-1);
    }

    *lock = NO_ONE_WAIT;
    if ((ret = vm_allocate(task_self(), (vm_address_t *)&mem,
            sizeof(int) * MAXDATA, TRUE)) != KERN_SUCCESS) {
        mach_error("vm_allocate failed:", ret);
        printf("Exiting with error.\n");
        exit(-1);
    }

    mem[0] = COPY_ON_WRITE;
    printf("value of lock before fork: %d\n", *lock);
    pid = fork();

    if (pid) {
        printf("PARENT: copied memory = %d\n", mem[0]);
        printf("PARENT: changing to %d\n", PARENT_CHANGED);
        mem[0] = PARENT_CHANGED;
        printf("\n");
        printf("PARENT: lock = %d\n", *lock);
        printf("PARENT: changing lock to %d\n", PARENT_WAIT);
        printf("\n");
        *lock = PARENT_WAIT;
        while (*lock == PARENT_WAIT)
            /* wait for child to change the value */ ;
        printf("PARENT: copied memory = %d\n", mem[0]);
        printf("PARENT: lock = %d\n", *lock);
        printf("PARENT: Finished.\n");
        *lock = PARENT_WAIT;
        exit(-1);
    }

    while (*lock != PARENT_WAIT)
        /* wait for parent to change lock */ ;

    printf("CHILD: copied memory = %d\n", mem[0]);
    printf("CHILD: changing to %d\n", CHILD_CHANGED);
    mem[0] = CHILD_CHANGED;
    printf("\n");
    printf("CHILD: lock = %d\n", *lock);
    printf("CHILD: changing lock to %d\n", CHILD_WAIT);
    printf("\n");

    *lock = CHILD_WAIT;
    while (*lock == CHILD_WAIT)
        /* wait for parent to change lock */ ;
    if ((ret = vm_deallocate(task_self(), (vm_address_t)lock,
            sizeof(int))) != KERN_SUCCESS) {
        mach_error("vm_deallocate failed:", ret);
        printf("Exiting.\n");
        exit(-1);
    }

    if ((ret = vm_deallocate(task_self(), (vm_address_t)mem,
            MAXDATA * sizeof(char))) != KERN_SUCCESS) {
        mach_error("vm_deallocate failed:", ret);
        printf("Exiting.\n");
        exit(-1);
    }
    printf("CHILD: Finished.\n");
}
```

Mach Scheduling

Each thread has a scheduling *priority* and *policy*. The priority is a number between 0 and€31 that indicates how likely the thread is to run. The higher the priority, the more likely the thread is to run. For example, a thread with priority 16 is more likely to run than a thread€with priority 10. The policy is by default a timesharing policy, which means that whenever the running thread blocks or a certain amount of time passes, the highest-priority runnable thread is executed. Under the timesharing policy, a thread's priority gets lower as it runs (it *ages*), so that not even a high-priority thread can keep a low-priority thread from eventually running.

Priorities

Each thread has three types of priorities associated with it: its base priority, its current priority, and its maximum priority. The base priority is the one the thread starts with; it can be explicitly set using a function such as **cthread_priority()**. The current priority is the one at which the thread is executing; it may be lower than the base priority due to aging or a call to **thread_switch()**. The maximum priority is the highest priority at which the thread can execute. When a thread starts, it inherits its base priority from its parent task and its maximum priority is set to a system-defined maximum.

These priorities can be set at three levels: the thread, the task, and (on multiprocessors) the processor set. At the thread level, you can use **cthread_priority()** or **thread_priority()** to set the base priority and to optionally lower the maximum priority. You can raise or lower just the maximum priority using **cthread_max_priority()** or **thread_max_priority()**. To raise a thread's maximum priority, you must obtain the privileged port of the thread's processor set, which only the superuser can do.

At the task level, you can set the task's base priority using **task_priority()**. The task's base priority is inherited by all threads that it forks; you can also specify that all existing threads in the task get the new base priority.

You can get the priorities of running tasks using **task_info()** and **thread_info()**. Or, from a shell window, you can view the priorities of running tasks using the UNIX command **ps**. The **-l** option of **ps** displays, among other things, the lowest values for maximum priority and current priority that were found in all the threads in the task. The **-m** option displays the current priority of every thread in the task. The following example shows the **ps** displays for Terminal.

```
localhost> ps -axu | grep Terminal
me        1658   2.8  2.4 1.31M  200K p2 S      0:00 grep Terminal
root       174   2.4 11.4 3.84M  936K p1 S      0:41 /NextApps/Terminal.app/T
localhost> ps -l 174
      F  UID   PID   PPID CP PRI BASE VSIZE RSIZE WCHAN STAT TT   TIME COMMAND
      1    0   174    156  0  10   10 4.30M 1.14K     0  S    ?  0:41 /NextAp
localhost> ps -m 174
USER       PID TT  %CPU STAT PRI    SYSTEM      USER  COMMAND
root       174  ?   1.8  S    16   0:15.76   0:19.17  /NextApps/Terminal.app/
                    0.1  S    10   0:06.15   0:00.54
```

Policies

The NeXT Mach operating system has three scheduling policies:

· Timesharing
· Interactive
· Fixed priority

Every thread starts with the timesharing policy, no matter what policy the creator of the thread has. If you want the policy of any thread to be something other than timesharing, you must set that thread's policy using **thread_policy()**.

The interactive policy is a variant of timesharing that's designed to be optimized for interactive applications. If you have a non-NEXTSTEP application, such as a terminal-oriented editor, you should set the main thread's policy to interactive using **thread_policy()**. (The Application Kit automatically sets up the first thread in an application to have an interactive policy.) Currently, the interactive policy is exactly the same as timesharing, but in the future performance might be enhanced by, for example, making interactive policy threads have higher priorities than the other threads in the task.

Fixed priority can be a dangerous policy if you aren't familiar with all of its consequences. For this reason, the fixed-priority policy is disabled by default. If you want to use fixed priorities, you must enable them using **processor_set_policy_enable()**. Threads that have the fixed-priority policy have their current priority always equal to their base priority (unless their priority is depressed by **thread_switch()**). A thread with the fixed-priority policy runs until one of the following happens:

- A higher-priority process becomes available to run.
- A per-thread, user-specified amount of time (the *quantum*) passes.
- The thread blocks, waiting for some event or system resource.

Because fixed-priority threads don't lose priority over time, they can prevent lower-priority threads from running. The opposite can happen, too; a low-priority, fixed-priority thread can be kept from running for enough time by higher-priority threads. The first problem can be solved in some cases by the fixed-priority thread calling **thread_switch()** to temporarily depress its priority or hand off the processor to another thread. The fixed-priority policy is often used for real-time problems, such as on-line transaction processing.

Mach C-Thread Functions

Mach provides a set of low-level, language-independent functions for manipulating threads of control. The C-thread functions are higher-level, C language functions in a run-time library that provide an interface to the Mach facilities. The constructs provided in the C-thread functions are:

- Forking and joining of threads
- Protection of critical regions with mutual exclusion (mutex) variables
- Condition variables for synchronization of threads

Another way of protecting critical regions and synchronizing threads is to use the Mach Kit's locking classes. The Mach Kit provides an object-oriented interface to a few limited areas of Mach functionality. See the *NEXTSTEP General Reference* for information on the Mach Kit.

If you intend to build multithreaded applications, you should use the C-thread functions rather than the Mach kernel functions. The C-thread functions are a natural and efficient set of functions for multithreaded applications, whereas the Mach thread functions are designed to provide the low-level mechanisms that packages such as the C-thread functions can be built with. An on-line example of a multithreaded NEXTSTEP application is under the directory **/NextDeveloper/Examples/AppKit/SortingInAction**.

Using External Functions and Methods

Many of the functions and methods provided by NEXTSTEP weren't designed with multithreaded applications in mind. As a result, they might not work correctly when called simultaneously by two or more of your application's threads. (A function or method that can safely be called by more than one thread at once is *thread-safe*.) In general, unless you know that a function or method is thread-safe, you should assume that it isn't.

The following are thread-safe:

- Distributed Objects, which is described in the *General Reference*

- Mach functions (except for **mach_error()**)

- UNIX system calls (you should use **cthread_errno()** instead of **errno**)

- NEXTSTEP exception handling (for example, **NX_RAISE()**)

- The **malloc()** function and its related functions, though their thread safety can be disabled by calling **malloc_singlethreaded()** (which in general should not be done)

The Objective C runtime system is not thread-safe by default. To make it thread-safe, use the function **objc_setMultithreaded()**.

The following are *not* thread-safe:

- The Application Kit (messages to kit objects should be sent only from the main thread)
- DPS client routines
- The Window Server (drawing should be done only from the main thread)
- Standard I/O functions, such as **printf()**
- Most of the functions in the libc library

In particular, the **usleep()** function should never be used in multithreaded programs. As an alternative, you can use the **thread_switch()** Mach function, as follows:

```
thread_switch(THREAD_NULL, SWITCH_OPTION_WAIT, msecs);
```

Using Shared Variables

All global and static variables are shared among all threads: If one thread modifies such a variable, all other threads will observe the new value. In addition, a variable reachable from a pointer is shared among all threads that can dereference that pointer. This includes objects pointed to by shared variables of pointer type, as well as arguments passed by reference in **cthread_fork()**. You should be careful to declare all shared variables as **volatile**, or the optimizer might remove references to them.

When pointers are shared, some care is required to avoid problems with dangling references. You must ensure that the lifetime of the object pointed to is long enough to allow the other threads to dereference the pointer. Since there's no bound on the relative execution speed of threads, the simplest solution is to share pointers to global or heap-allocated objects only. If a pointer to a local variable is shared, the function that variable is defined in must remain active until it can be guaranteed that the pointer will no longer be dereferenced by other threads. The synchronization functions can be used to ensure this.

Unless a library has been designed to work in the presence of reentrancy, you should assume that the library makes unprotected use of shared data. You must protect against this through the use of a mutex that's locked before every library call (or sequence of library calls) and unlocked afterward. For example, you should lock a mutex before calling **printf()** and unlock the mutex afterward.

Synchronization of Variables

This section describes mutual exclusion and synchronization functions, which are used to constrain the possible interleavings of the execution streams of threads. These functions manipulate *mutex* and *condition* variables, which are defined as follows:

```
typedef struct mutex {...} *mutex_t;

typedef struct condition {...} *condition_t;
```

Mutually exclusive access to mutable data is necessary to prevent corruption of data. As a simple example, consider concurrent attempts to update a simple counter. If two threads fetch the current value into a (thread-local) register, increment, and write the value back in some order, the counter will only be incremented once, losing one thread's operation. A mutex solves this problem by making the fetch-increment-deposit action atomic. Before fetching a counter, a thread locks the associated mutex, and after depositing a new value the thread unlocks the mutex:

```
mutex_lock(m);
count += 1;
mutex_unlock(m);
```

If any other thread tries to use the counter in the meantime, it will block when it tries to lock the mutex. If more than one thread tries to lock the mutex at the same time, only one will succeed; the rest will block.

Condition variables are used when one thread wants to wait until another thread has finished doing something. Every condition variable should be protected by a mutex. Conceptually, the condition is a boolean function of the shared data that the mutex protects. Commonly, a thread locks the mutex and inspects the shared data. If it doesn't like what it finds, it waits, using a condition variable:

```
mutex_lock(mutex_t m);
. . .
while ( /* condition isn't true */ )
    condition_wait(condition_t c, mutex_t m);
. . .
mutex_unlock(mutex_t m);
```

The call to **condition_wait()** temporarily unlocks the mutex to give other threads a chance to get in and modify the shared data. Eventually, one of them should signal the condition (which wakes up the blocked thread) before it unlocks the mutex:

```
mutex_lock(mutex_t m);
. . .      /* modify shared data */
condition_signal(condition_t c);
mutex_unlock(mutex_t m);
```

At that point, the original thread will regain its lock and can look at the shared data to see if things have improved. It can't assume that it will like what it sees, because some other thread may have slipped in and altered the data after the condition was signaled.

You must take special care with data structures that are dynamically allocated and deallocated. In particular, if the mutex that's controlling access to a dynamically allocated record is part of the record, make sure that no thread is waiting for the mutex before freeing the record.

Attempting to lock a mutex that one already holds is another common error. The offending thread will block waiting for itself. This can happen when a thread is traversing a complicated data structure, locking as it goes, and reaches the same data by different paths. Another instance of this is when a thread is locking elements in an array, say to swap them, and it doesn't check for the special case that the elements are the same.

You must be careful to avoid deadlock, a condition in which one or more threads are permanently blocked waiting for each other. The above scenarios are a special case of deadlock. The easiest way to avoid deadlock with mutexes is to impose a total ordering on the mutexes, and then ensure that threads only lock mutexes in increasing order.

You must decide what kind of granularity to use in protecting shared data with mutexes. The two extremes are to have one mutex protecting all shared memory, or to have one mutex for every byte of shared memory. Finer granularity normally increases the possible parallelism because less data is locked at any one time. However, it also increases the overhead lost to locking and unlocking mutexes and increases the possibility of deadlock.

Program Example: C Threads

This section demonstrates the use of the C-thread functions in writing a multithreaded program. The program is an example of how to structure a program with a single master thread that spawns a number of concurrent slaves. The master thread waits until all the slaves have finished and then exits.

Once created, a slave thread simply loops calling a function that makes the processor available to other threads. After this loop is finished, the slave thread informs the master that it's done, and then dies. In a more useful version of this program, each slave process would do something while looping.

```
#import <stdio.h>
#import <mach/cthreads.h>

volatile int count;    /* number of slave threads active */
mutex_t      lock;     /* mutual exclusion for count */
mutex_t      print;    /* mutual exclusion for printfs */
condition_t  done;     /* signaled each time a slave finishes */

void init()
```

```
{
    /* Allocate mutex variables "lock" and "print". */
    lock = mutex_alloc();
    print = mutex_alloc();

    /* Allocate condition variable "done". */
    done = condition_alloc();

    count = 0;
}

/*
 * Each slave just loops, yielding the processor on each
 * iteration.  When it's finished, it decrements the global
 * count and signals that it's done.
 */
void slave(int n)
{
    int i;

    for (i = 0; i < 100; i += 1)
        cthread_yield();

    /*
     * If any thread wants to access the count variable, it
     * first locks the mutex.  When the mutex is locked, any
     * other thread wanting the count variable must wait until
     * the mutex is unlocked.
     */
    mutex_lock(lock);
    count -= 1;
    mutex_lock(print);
    printf("Slave %d finished.\n", n);
    mutex_unlock(print);
    /* Signal that this slave has finished. */
    condition_signal(done);
    mutex_unlock(lock);
}

/*
 * The master spawns a given number of slaves and then waits
 * for them all to finish.
 */
void master(int nslaves)
{
    int i;

    for (i = 1; i <= nslaves; i++) {
        mutex_lock(lock);
        /* Increment count with the creation of each slave thread.  */
        count += 1;
        /* Fork a slave and detach it.  */
        cthread_detach(cthread_fork((cthread_fn_t)slave, (any_t)i));
        mutex_unlock(lock);
    }

    mutex_lock(lock);
    /*
     * Master thread loops waiting on the condition done.  Each
     * time the master thread is signaled by a condition_signal
     * call, it tests the count for a value of zero.
     */
    while (count != 0)
        condition_wait(done, lock);
    mutex_unlock(lock);

    mutex_lock(print);
    printf("All %d slaves have finished.\n", nslaves);
```

```
        mutex_unlock(print);
    }

    main()
    {
        init();
        master(15);  /* Create master thread and 15 slaves. */
    }
```

Mach Exception Handling

Exceptions are synchronous interruptions to the normal flow of program control caused by the occurrence of unusual conditions during program execution. Raising an exception causes the operating system to manage recovery from the unusual condition.

Exceptions include:

· Illegal accesses (bus errors, segmentation and protection violations)

· Arithmetic errors (overflow, underflow, divide by zero)

· Hardware instructions intended to support facilities such as emulation, debugging, and error detection

Note: Software interrupts and other actions caused by asynchronous external events aren't considered to be exceptions.

Although many exceptions, such as page faults, can be handled by the operating system and dismissed transparently to the user, the remaining exceptions are exported to the user by the operating system's exception-handling facility (for example, by invoking a handler or producing a core dump).

Four major classes of applications use exceptions:

· DebuggingÐDebuggers rely on exceptions generated by hardware trace and breakpoint facilities. Other exceptions that indicate errors must be reported to the debugger; the presence of the debugger indicates the user's interest in any anomalous program behavior.

· Core dumpsÐIn the absence of a debugger, a fatal exception can cause the execution state of a program to be saved in a file for later examination.

· Error handlingÐCertain applications sometimes handle their own exceptions (particularly arithmetic). For example, an error handler could substitute 0 for the result of a floating underflow and continue execution. Error handlers are often required by high-level languages.

· EmulationÐGenerally, computers generate exceptions upon encountering operation codes that can't be executed by the hardware. Emulators can be built to execute the desired operation in software. Such emulators serve to extend the instruction set of the underlying machine by performing instructions that aren't present in the hardware.

The following sections contrast the UNIX approach to error handling with the general model upon which the Mach exception-handling facility is built, and then present specific information about the Mach exception-handling facility.

The UNIX Approach to Exception Handling

Designers of operating systems have approached exceptions in a variety of ways. The drawbacks of most approaches include limited functionality (often the result of designing only for debuggers) and lack of extensibility to a multithreaded environment.

The UNIX operating system generalizes exception handling to the signal facility, which handles all interruptions to normal program flow. The varying requirements of different types of interruptions (such as exceptions, timer expiration, or a control character from the terminal) entail semantics that vary from signal to signal; the default action can be nothing, stop, continue from stop, or terminate (with or without a core dump). The user can change these defaults or specify a handler to be invoked by a signal. The interface to these handlers includes a partial machine context, but registers outside this context aren't accessible.

Debugging support in UNIX is centralized in the **ptrace()** system call: It performs all data transfer and process control needed by debuggers, and interacts with the signal facility to make signals visible to debuggers (including signals that would otherwise invoke error handlers or emulators). The occurrence of a signal in a debugged process causes that process to stop in a peculiar manner and notify the debugger that something has happened. This notification is implemented by special treatment of debugged processes in the **wait()** system call; this call usually detects terminated processes, but also detects stopped processes that are being debugged. One consequence of these features and their implementation is that debuggers are restricted to debugging processes that are the immediate children of the debugger.

Two major problems with the UNIX signal facility are:

· Executing the signal handler in the same context as the exception makes many registers inaccessible. These registers are often the very registers that an arithmetic error handler needs to modify (for example, by substituting 0 for a floating underflow).

· The entire concept of signals is predicated on single-threaded applications. Adapting signals to multithreaded applications is difficult and complicates the interface to them. At least half a dozen major changes to the UNIX signal implementation in the Mach kernel have been required for this reason.

The typical use of signal handlers is to detect and respond to external events; for this they're adequate, but as an exception-handling facility, they leave much to be desired.

A Model for Generalized Exception Handling

The Mach exception-handling facility is based on a model whose generality is sufficient to describe virtually all uses of exceptions, including those made by the four classes of applications discussed earlier.

The Mach exception-handling model divides applications that use exceptions into two major classes:

· Error handlersÐPerform recovery actions in response to an exception and resume execution of the thread involved. This class includes both error handlers and emulators. Error handlers typically execute in the same address space as that thread for efficiency reasons (access to state).

· DebuggersÐExamine the state of an entire application to investigate why an exception occurred or why the program is misbehaving. This class includes both interactive debuggers and the servers that produce core dumps; the latter can be viewed as front ends to debuggers that examine core dumps. Debuggers usually execute in address spaces distinct from the application for protection reasons.

This chapter uses the terms *error handler* and *debugger* to refer to these two classes (for example, a core dumper is a debugger). The term *handler* is used to refer to any application that uses exceptions.

The Mach exception-handling model is derived by examining the requirements common to error handlers and debuggers. Specifically, the occurrence of an exception requires suspension of the thread involved and notification of a handler. The handler receives the notification and performs some computation (for example, an error handler fixes the error, a debugger decides what to do next), after which the thread is either resumed or terminated.

The model presented in this section covers all uses of exceptions. The occurrence of an exception invokes a four-step process involving the thread that caused the exception (victim) and the entity that handles the exception (handler, which may be the operating system):

1. Victim does a *raise*, causing notification of the occurrence of an exception.

2. Victim does a *wait*, synchronizing with completion of exception handling.

3. Handler does a *catch*, receiving notification. This notification usually identifies the exception and the victim, although some of this identification may be implicit in where and how the notification is received.

4. Handler takes either of two possible actions: *clear* the exception (causing the victim to return from the *wait*), or *terminate* the victim thread.

The primitives appearing in bold in this model constitute the high-level model interface to exceptions and can be viewed as operating on *exception objects*. The handler will usually perform other functions between the *catch* step and the *clear* or *terminate* step; these functions are part of the handler application itself, rather than part of the exception model.

Exception Handling in Mach

The Mach exception-handling facility was designed as a general implementation of the exception-handling model described above. The major design goals for this new facility were:

· Single facility with consistent semantics for all exceptions
· Clean and simple interface
· Full support for debuggers and error handlers
· No duplication of functionality within kernel
· Support for user-defined exceptions

A consequence of these goals is a rejection of the notion of a handler executing in the same context as the exception it's handling. There is no clean and straightforward way to make a thread's context available to the thread itself; this results in a single thread having multiple contexts (a currently executing context and one or more saved exception contexts). In turn this causes serious naming and functionality problems for operations that access or manipulate thread contexts. Because Mach supports multiple threads within the same task, it's sufficient to stop the thread that caused the exception and execute the handler as another thread in the same task.

The Mach exception-handling facility implements the exception-handling model with Mach kernel functions to avoid duplicating kernel functionality. Because the handler never executes in the context of the victim thread, the *raise*, *wait*, *notify*, and *clear* primitives constitute a remote procedure call (RPC). They're therefore implemented using a message-based RPC provided by the Mach communication facility. The remaining *terminate* primitive is exactly the **thread_terminate()** or **task_terminate()** function; no special action is required to terminate the thread or task instead of completing the RPC.

The exception RPC consists of two messages: an initial message to invoke the RPC, and a reply message to complete the RPC. The initial message contains the following items:

· Send and reply ports for the RPC
· The identities of the thread that caused the exception and of the corresponding task
· A machine-independent exception class (see the section ªException Classificationº)
· Two machine-dependent fields that further identify the exception

If the RPC is completed, the reply message contains the two RPC ports and a return code from the handler that handled the exception (success in almost all cases). MiG-generated stub routines perform the generation and decoding of the messages; this allows users to avoid dealing directly with the contents of the messages. (MiG is described in Chapter 2.)

An exception RPC corresponds to our exception model as follows:

· *raise*ÐSend initial message.
· *wait*ÐWait for and receive reply message.
· *catch*ÐReceive initial message.
· *clear*ÐSend reply message.

Exception Ports

The two messages that constitute the RPC are sent to and received from ports corresponding to the handler (initial message) and victim (reply message). The handler's port is registered as the exception port for either the victim's task or thread; the kernel consults this registration when an exception occurs. The reply port is specified in the initial message; for hardware exceptions, the kernel allocates the reply port and caches it for reuse on a per-thread basis. Mach kernel functions are available to register a port as an exception port for a task or thread, and to return the port currently registered; these functions for implementing debuggers and error handlers are described in the section ªProgram Example: Exception Handling.º

Registering exception ports for both tasks and threads effects a separation of concerns between error handlers and debuggers. Error handlers are supported by the thread exception ports because error handlers usually affect only the victim thread; different threads within a task can have different error handlers. The registered exception port for a thread defaults to the null port at thread creation; this defaults the initial error handler to no handler. Debuggers are supported by the task exception ports because debuggers operate on the application level; this includes at least all the threads in the victim's task, so at most one debugger is associated with a single task. The registered exception port for a task is inherited from the parent task at task creation; this supports debuggers that handle trees of tasks (such as a multitasking parallel program) and inheritance of core-dump servers.

The presence of both task and thread exception ports creates a potential conflict because both are applicable to any exception. This is resolved by examining the differences between error handlers and debuggers. Error handlers use exceptions to implement portions of an application; an error handler is an integral part of the application that generates its exceptions. Exceptions handled by an error handler may be unusual, but they don't indicate anomalous or erroneous behavior. In contrast, debuggers use exceptions to investigate anomalous or erroneous application behavior; as a result debuggers have little interest in exceptions successfully handled by error handlers. This implies that exceptions should invoke error handlers in preference to debuggers; this preference is implemented by having thread exception ports take precedence over task exception ports in determining where to direct the RPC invoked by an exception. If neither an error handler nor a debugger can successfully handle an exception, the task is terminated.

User Extensibility

The Mach exception-handling facility permits you to define and handle your own exceptions in addition to those defined by the system.

The software class of exceptions (see the section ªException Classificationº) contains a range of codes reserved for user-defined exceptions; this allows the handling of these exceptions to be integrated into the handling of system-defined exceptions. The same ports are used in both cases, and the interface to handlers is identical.

An advantage of this approach is that user-defined exceptions can immediately be recognized as such, even by debuggers that can't decode the machine-dependent fields that identify the exact exception.

Generation of user-defined exceptions is facilitated by a MiG stub routine that implements the exception RPC (in turn this routine is generated automatically from an interface description of the exception RPC). User code that detects an exception simply obtains the appropriate exception port from the kernel and calls this stub routine; the stub routine handles the RPC and returns a return code from the handler. Alternatively, you may use the MiG exception interface with your own exceptions and exception ports; this approach may be advantageous for applications that handle only user-defined exceptions.

Implementing Error Handlers

Error handlers are supported by thread exception ports and invoked by remote procedure calls on those ports. An error handler is associated with a thread by registering a port on which the error handler receives exception RPCs as the exception port of the thread. This registration causes all exceptions occurring in the thread to invoke RPCs to the error handler's port. Since most error handlers can't handle all possible exceptions that could occur, they must check each exception and forward it to the corresponding task exception port if it can't be handled. This forwarding can be performed by obtaining the exception port for the task specified in the initial message and sending the initial message there. Alternatively, the error handler can return a failure code in the reply message; this causes the sender of the initial message to reinitiate the RPC using the task exception port.

Implementation of error handlers requires additional functionality beyond completing the RPC. This functionality is supported by separate Mach kernel functions that can also be used by other applications. The most common actions and corresponding functions are:

· Read/write register state: **thread_get_state()**, **thread_set_state()**

· Read/write memory state: Access memory directly within task; otherwise **vm_read()**, **vm_write()**

· Terminate thread: **thread_terminate()**

· Resume thread: Send reply message to complete RPC (**msg_send()**)

Some applications may require that error handlers execute in the context of (that is, on the stack of) the thread that caused the exception (such as emulation of UNIX signal handlers). Although this appears to conflict with the principle of never executing an error handler in the context of the victim thread, it can be implemented by using a system-invoked error handler to set up the application's handler. Specifically, the error handler invoked by the exception RPC modifies the victim thread so that the application's handler is executed when the thread is resumed. Unwinding the stack when the application's error handler finishes is the responsibility of the application developer.

Implementing Debuggers

Debuggers are supported by the task exception ports; exceptions invoke debuggers with remote procedure calls on those ports. A debugger is associated with a task by registering a port on which the debugger receives exception RPCs as the task's exception port. An exception RPC stops only the victim thread pending RPC completion; other threads in the task continue running. This has two consequences:

· If the debugger wants to stop the entire task, a **task_suspend()** must be performed. A straightforward way to accomplish this is to do it inside the exception RPC and then complete the RPC; the victim thread can't resume execution upon RPC completion because its task has been suspended.

· Multiple exceptions from a multithreaded task may be outstanding for the debugger on a single debugger invocation. If the debugger doesn't handle these pending exceptions for the task, some may appear to occur at impossible times (such as a breakpoint occurring after the user has removed it).

The Mach exception-handling facility is one small component of the kernel that can be used by debuggers. The various actions required to support debuggers are implemented using general-purpose functions that also support other applications. Some of the more important debugger actions and corresponding kernel functions are:

· Detect event: **msg_receive()**. System components that generate or detect external events (such as interrupt characters on a terminal) signal the events by sending messages.

· Read and write application memory (includes setting breakpoints): **vm_read()**, **vm_write()**.

· Read and write application registers (includes setting single-step mode if available): **thread_get_state()**, **thread_set_state()**.

· Continue application: Task and thread control functions.

· End debugging session: **task_terminate()**.

Exceptions that invoke error handlers using thread exception ports aren't visible to debuggers. A debugger that wants to detect error handler invocation can insert one or more breakpoints in the error handler itself; exceptions caused by these breakpoints will be reported to the debugger.

Debugger Attachment

The independence property of the Mach kernel described previously allows Mach to support debugger attachment and detachment without change to the kernel itself. Traditional UNIX systems require that the debugged process be the child of the debugger; this makes it impossible to debug a process that wasn't started by the debugger. Subsequent developers have expended considerable effort to implement an **attach** primitive that allows a debugger to attach to a previously started process and debug it; this allows analysis of failures that may not be repeatable. Similarly these systems allow a debugger to detach from a running process and exit without affecting the process. No design change is required to support this functionality; the debugger need only obtain the port representing the task to be debugged, and may then use all of the functions previously discussed to debug that task. A debugger can detach from a task by resetting the task's exception port to its former value; there is no other connection between the debugger and task being debugged.

Parallel and Distributed Debugging

The design of the exception-handling facility also supports parallel and distributed debugging without change. There are several cases to be considered based on the structure of the debugger and the application being debugged. In all of these cases the debugger itself may be a parallel or distributed application consisting of multiple tasks and threads.

For parallel applications composed of multiple threads within a single task, a debugger need only register its exception RPC port as that task's exception port. Multiple concurrent exceptions result in multiple RPC invocations being queued to that port; each invocation identifies the thread involved. The Mach communication facility allows the debugger to accept all of these RPCs before responding to any of them, and to respond to them in any order. (Of course the debugger must keep track of the RPCs and make sure they're all responded to when continuing the application.) A straightforward implementation is to suspend the task in response to the first RPC, and then complete all pending exception RPCs recording the threads and exceptions involved. The exceptions can then be reported to the user all at once.

For parallel applications composed of multiple tasks within a single machine, only minor changes to the above debugger logic are required. The debugger must now register its exception RPC port as the task exception port for each task, and may choose to identify components of the parallel application by tasks instead of threads. Suspending or resuming the entire application now requires an operation on each task. If the application dynamically creates tasks, an additional interface to report these new tasks to the debugger may be required so that the new tasks can be suspended and resumed by the debugger.

Network transparency allows the components of a debugger and the debugged application to be spread throughout a network; all required operations extend transparently across the network. This supports a number of possible debugging scenarios:

- The application and the debugger are on separate hosts.
- The application being debugged is distributed over the network. The debugger doesn't require modifications beyond those needed to deal with applications composed of multiple tasks.
- The debugger itself can be distributed over the network.

The last scenario is useful for implementing fast exception response in a debugger for applications that run in parallel on several distributed hosts; if the exception RPC stays within the host, suspending of all application components on that host can be done faster.

GDB Enhancements

The Mach exception-handling facility and other Mach kernel functions have been used to enhance GDB (the GNU source-level debugger) for debugging multithreaded tasks. This enhanced version of GDB operates at the task level (that is, any exception causes GDB to suspend the entire task). A notion of the current thread has been added; this thread is used by any thread-specific command that doesn't specify a thread. New commands are provided to list the threads in the task, change the current thread, and examine or control individual threads. Thread-specific breakpoints are supported by logic that transparently continues the application from the breakpoint until the desired thread hits it. Implementation of attachment to running tasks, as described earlier in the section ªDebugger Attachment,º is in progress, as are changes to deal with multiple concurrent breakpoints.

The existence of multiple threads within a debugged task complicates GDB's execution control logic. In addition to the **task_suspend()** required upon exception detection, resuming from a breakpoint becomes somewhat intricate. Standard GDB removes the breakpoint, single-steps the process, puts back the breakpoint and continues. The enhanced version must ensure that only the thread at the breakpoint executes while performing the single step; this requires switching from task suspension to suspension of all of the threads except one and then back again before resuming the application.

The Mach exception-handling facility is an important implementation base for the enhancements to GDB. Identification of the victim thread in the initial message makes it possible to handle multiple concurrent exceptions; all the UNIX functions (for example, **ptrace()**) are restricted to one current signal per task, and hence preclude handling of multiple concurrent exceptions. Additionally, the independence of the debugger from the debugged application makes it possible to implement debugger attachment without kernel modifications; the UNIX operating system requires extensive kernel modifications to achieve similar functionality.

Exception Classification

The Mach exception-handling facility employs a new hardware-independent classification of exceptions. This is in contrast to previous systems (such as UNIX), whose exception classifications are closely wedded to the hardware they were originally developed on. This new classification divides all exceptions into six classes based on the causes and uses of the exceptions; further hardware-specific and software-specific distinctions can be made within these classes as needed. The six classes are:

- Bad accessÐA user access to memory failed for some reason and the operating system was unable to recover (such as invalid memory or protection violation).
- Bad instructionÐA user executed an illegitimate instruction (such as an undefined instruction, reserved operand, or privileged instruction).
- ArithmeticÐA user arithmetic instruction failed for an arithmetic reason (such as overflow, underflow, or divide by zero).
- EmulationÐA user executed an instruction requiring software emulation.
- SoftwareÐA broad class including all exceptions intended to support software. These fall into three subclasses:

Hardware	Hardware instructions to support error detection (such as trap on overflow or trap on subscript out of range).
Operating system	Exceptions detected by operating system during system call execution (such as no receiver on pipe). These are for operating system emulation (such as UNIX emulation). Mach doesn't use exceptions for system call errors.
User	Exceptions defined and caused by user software for its own purposes.

- DebuggerÐHardware exceptions to support debuggers (such as breakpoint instruction and trace trap).

In cases of potential confusion (for example, is it a bad instruction or an instruction requiring emulation?) the correct classification is always clear from the intended uses of the instruction as determined by the hardware and system designers.

Two machine-dependent fields are used to identify the precise exception within a class for flexibility in encoding exception numbers. Two fields are needed for emulation instructions containing a single argument (one for the instruction, one for the argument), but we have also found them useful for constructing machine-dependent exception classifications (for example, using one field to hold the trap number or vector, and the other to distinguish this trap from the others that use this number or vector). Cases in which two fields don't suffice require a separate interface to extract the additional machine-dependent status.

Kernel Interface

This section lists functions that relate directly to the exception-handling facility. The following Mach functions let you raise exceptions, handle them, and get or set exception ports. See Chapter 4 for descriptions of each of these functions and macros.

- **exception_raise()**
- **exc_server()**
- **mach_NeXT_exception()**
- **mach_NeXT_exception_string()**
- **task_set_exception_port()**
- **task_get_exception_port()**
- **thread_set_exception_port()**
- **thread_get_exception_port()**

Another important function is one you implement yourself: **catch_exception_raise()**. If you implement this function, it must have the following syntax:

kern_return_t **catch_exception_raise(**port_t *exception_port*, port_t *thread*, port_t *task*, int *exception*, int *code*, int *subcode***)**

Program Example: Exception Handling

The following example shows how to raise and handle user-defined exceptions. The program sets up a new exception port, sets up a thread to listen to this port, and then raises an exception by calling **exception_raise()**. The thread that's listening to the exception port receives the exception message and passes it to **exc_server()**, which calls the user-implemented function **catch_exception_raise()**.

This program's implementation of **catch_exception_raise()** determines whether it understands the exception. If so, it handles the exception by displaying a message. If not, this implementation of **catch_exception_raise()** sets a global variable that indicates that its calling thread should forward the exception to the old exception port. This program doesn't know which exception handler is listening to the old exception port; it could be the default UNIX exception handler, GDB, or any other exception handler.

```
/*
 * raise.c: This program shows how to raise user-specified exceptions.
 * If you use GDB, you can't set any breakpoints or step through any
 * code between the call to task_set_exception_port and the return
 * from exception_raise().  (You can never use GDB to debug exception
 * handling code, since GDB stops the program by generating an
 * EXC_BREAKPOINT exception.)
 */
#import <mach/mach.h>
#import <mach/exception.h>
#import <mach/cthreads.h>
#import <mach/mig_errors.h>

typedef struct {
    port_t old_exc_port;
    port_t clear_port;
    port_t exc_port;
} ports_t;

volatile boolean_t pass_on = FALSE;
mutex_t            printing;
```

```
/* Listen on the exception port. */
any_t exc_thread(ports_t *port_p)
{
    kern_return_t   r;
    char           *msg_data[2][64];
    msg_header_t   *imsg = (msg_header_t *)msg_data[0],
                   *omsg = (msg_header_t *)msg_data[1];

    /* Wait for exceptions. */
    while (1) {
        imsg->msg_size = 64;
        imsg->msg_local_port = port_p->exc_port;
        r = msg_receive(imsg, MSG_OPTION_NONE, 0);

        if (r==RCV_SUCCESS) {
            /* Give the message to the Mach exception server. */
            if (exc_server(imsg, omsg)) {
                /* Send the reply message that exc_serv gave us. */
                r = msg_send(omsg, MSG_OPTION_NONE, 0);
                if (r != SEND_SUCCESS) {
                    mach_error("exc_thread msg_send", r);
                    exit(1);
                }
            }
            else { /* exc_server refused to handle imsg. */
                mutex_lock(printing);
                printf("exc_server didn't like the message\n");
                mutex_unlock(printing);
                exit(2);
            }
        }
        else { /* msg_receive() returned an error. */
                mach_error("exc_thread msg_receive", r);
                exit(3);
        }

        /* Pass the message to old exception handler, if necessary. */
        if (pass_on == TRUE) {
            imsg->msg_remote_port = port_p->old_exc_port;
            imsg->msg_local_port = port_p->clear_port;
            r = msg_send(imsg, MSG_OPTION_NONE, 0);
            if (r != SEND_SUCCESS) {
                mach_error("msg_send to old_exc_port", r);
                exit(4);
            }
        }
    }
}

/*
 * catch_exception_raise() is called by exc_server().  The only
 * exception it can handle is EXC_SOFTWARE.
 */
kern_return_t catch_exception_raise(port_t exception_port,
    port_t thread, port_t task, int exception, int code, int subcode)
{
    if ((exception == EXC_SOFTWARE) && (code == 0x20000)) {
        pass_on = FALSE;
        /* Handle the exception so that the program can continue. */
        mutex_lock(printing);
        printf("Handling the exception\n");
        mutex_unlock(printing);
        return KERN_SUCCESS;
    }
    else { /* Pass the exception on to the old port. */
        pass_on = TRUE;
        mutex_lock(printing);
        mach_NeXT_exception("Forwarding exception", exception,
            code, subcode);
        mutex_unlock(printing);
        return KERN_FAILURE;  /* Couldn't handle this exception. */
    }
}

main()
{
    int             i;
    kern_return_t   r;
    ports_t         ports;

    printing = mutex_alloc();

    /* Save the old exception port for this task. */
    r = task_get_exception_port(task_self(), &(ports.old_exc_port));
    if (r != KERN_SUCCESS) {
        mach_error("task_get_exception_port", r);
        exit(1);
    }

    /* Create a new exception port for this task. */
    r = port_allocate(task_self(), &(ports.exc_port));
    if (r != KERN_SUCCESS) {
        mach_error("port_allocate 0", r);
        exit(1);
    }
    r = task_set_exception_port(task_self(), (ports.exc_port));
    if (r != KERN_SUCCESS) {
        mach_error("task_set_exception_port", r);
        exit(1);
    }

    /* Fork the thread that listens to the exception port. */
    cthread_detach(cthread_fork((cthread_fn_t)exc_thread,
        (any_t)&ports));
    /* Raise the exception. */
    ports.clear_port = thread_reply();
#ifdef NOT_OUR_EXCEPTION
    /* By default, EXC_BAD_ACCESS causes a core dump. */
    r = exception_raise(ports.exc_port, ports.clear_port,
        thread_self(), task_self(), EXC_BAD_ACCESS, 0, 0);
#else
    r = exception_raise(ports.exc_port, ports.clear_port,
        thread_self(), task_self(), EXC_SOFTWARE, 0x20000, 0);
#endif

    if (r != KERN_SUCCESS)
        mach_error("catch_exception_raise didn't handle exception",
            r);
    else {
        mutex_lock(printing);
        printf("Successfully called exception_raise\n");
        mutex_unlock(printing);
    }

    sleep(5);  /* Exiting too soon can disturb other exception
                * handlers. */
}
```