Operating System Software

NeXT Developer's Library

NeXTstep

Draw upon the library of software contained in NeXTstep to develop your applications. Integral to this development environment are the Application Kit and Display PostScript.

Concepts

A presentation of the principles that define NeXTstep, including user interface design, object-oriented programming, event handling, and other fundamentals.

Reference, Volumes 1 and 2

Detailed, comprehensive descriptions of the NeXTstep Application Kit software.

Sound, Music, and Signal Processing

Let your application listen, talk, and sing by using the Sound Kit and the Music Kit. Behind these capabilities is the DSP56001 digital signal processor. Independent of sound and music, scientific applications can take advantage of the speed of the DSP.

Concepts

An examination of the design of the sound and music software, including chapters on the use of the DSP for other, nonaudio uses.

Reference

Detailed, comprehensive descriptions of each piece of the sound, music, and DSP software.

NeXT Development Tools

A description of the tools used in developing a NeXT application, including the Edit application, the compiler and debugger, and some performance tools.

NeXT Operating System Software

A description of NeXT's operating system, Mach. In addition, other low-level software is discussed.

Writing Loadable Kernel Servers

How to write loadable kernel servers, such as device drivers and network protocols.

NeXT Technical Summaries

Brief summaries of reference information related to NeXTstep, sound, music, and Mach, plus a glossary and indexes.

Supplemental Documentation

Information about PostScript, RTF, and other file formats useful to application developers.

NeXT Operating System Software

We at NeXT Computer have tried to make the information contained in this manual as accurate and reliable as possible. Nevertheless, NeXT disclaims any warranty of any kind, whether express or implied, as to any matter whatsoever relating to this manual, including without limitation the merchantability or fitness for any particular purpose. NeXT will from time to time revise the software described in this manual and reserves the right to make such changes without obligation to notify the purchaser. In no event shall NeXT be liable for any indirect, special, incidental, or consequential damages arising out of purchase or use of this manual or the information contained herein.

Copyright ©1990 by NeXT Computer, Inc. All Rights Reserved.
[2913.00]

The NeXT logo and NeXTstep are registered trademarks of NeXT Computer, Inc., in the U.S. and other countries. NeXT is a trademark of NeXT Computer, Inc. PostScript is a registered trademark of Adobe Systems Incorporated. UNIX is a registered trademark of AT&T. Sun and NFS are registered trademarks of Sun Microsystems, Inc. All other trademarks mentioned belong to their respective owners.

Notice to U.S. Government End Users:

Restricted Rights Legends

For civilian agencies: This software is licensed only with "Restricted Rights" and use, reproduction, or disclosure is subject to restrictions set forth in subparagraph (a) through (d) of the Commercial Computer Software—Restricted Rights clause at 52.227-19 of the Federal Acquisition Regulations.

For units of the Department of Defense: Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

NeXT, Inc., 900 Chesapeake Drive, Redwood City, CA 94063.

Written by Gary Miller and Kathy Walrath
Edited by Caroline Rose, Roy West, and Helen Casabona
Book design by Eddie Lee
Illustrations by Jeff Yaksick
Production by Adrienne Wong, Jennifer Yu, and Katherine Arthurs
Publications management by Cathy Novak

Reorder Product #N6007B

Contents

Introduction

1-1 Chapter 1: The Mach Operating System

Index

Introduction

This manual describes the NeXT™ Mach operating system. It's part of a collection of manuals called the *NeXT Developer's Library*; the illustration on the first page of this manual shows the complete set of manuals in this Library.

You don't have to read this manual to be able to create NeXTstep® applications; however, the information in this manual is necessary if you want to use Mach features such as multiple threads in a task.

UNIX®-related topics aren't covered in this manual. For information about the UNIX operating system, you should refer to the books listed in "Suggested Reading" (in the *NeXT Technical Summaries* manual) or to the on-line UNIX manual pages (available through the NeXT Digital Library).

A version of this manual is stored on-line in the Digital Library, which is described in the user's manual *NeXT Applications*. The Digital Library also contains Release Notes, which provide last-minute information about the latest release of the software.

How This Manual is Organized

This manual contains the following four chapters:

- Chapter 1, "The Mach Operating System," describes NeXT's version of Mach. It discusses concepts such as the kernel, tasks and threads, and ports and messages. This chapter also explains how Mach manages virtual memory allocation and how it handles exceptions.
- Chapter 2, "Using Mach Messages," describes how to create Mach messages, either by hand or by using the Mach Interface Generator (MiG).
- Chapter 3, "Using Loadable Kernel Servers," describes how to communicate with loadable kernel servers and the kernel-server loader by using the kernel-server loader functions.
- Chapter 4, "C Functions," provides detailed descriptions of all Mach operating system functions that are available to user-level programs. These functions are summarized in the *NeXT Technical Summaries* manual.

Note: UNIX library functions and system calls aren't covered in Chapter 4; they're described in the on-line UNIX manual pages.

Conventions

Syntax Notation

Where this manual shows the syntax of a function, command, or other programming element, the use of bold, italic, square brackets, and ellipsis has special significance, as described here.

Bold denotes words or characters that are to be taken literally (typed as they appear). *Italic* denotes words that represent something else or can be varied. For example, the syntax

> **print** *expression*

means that you follow the word **print** with an expression.

Square brackets [ ] mean that the enclosed syntax is optional, except when they're bold **[ ]**, in which case they're to be taken literally. The exceptions are few and will be clear from the context. For example,

> *pointer* [*filename*]

means that you type a pointer with or without a file name after it, but

> **[**_receiver message_**]**

means that you specify a receiver and a message enclosed in square brackets.

Ellipsis (...) indicates that the previous syntax element may be repeated. For example:

Syntax	**Allows**
pointer ...	One or more pointers
pointer [, *pointer*] ...	One or more pointers separated by commas
pointer [*filename* ...]	A pointer optionally followed by one or more file names
pointer [, *filename*] ...	A pointer optionally followed by a comma and one or more file names separated by commas

Special Characters

In general, notation like

Alternate-x

represents the character you get when you hold down the Alternate key while typing x. Because the modifier keys Alternate, Command, and Control interpret the case of letters differently, their notation is somewhat different:

Notation	Meaning
Alternate-x	Hold down Alternate while typing lowercase x.
Alternate-X	Hold down Alternate while typing uppercase X (with either Shift or Alpha Lock).
Alternate-Shift-x	Same as Alternate-X.
Command-d	Hold down Command while typing lowercase d; if Alpha Lock is on, pressing the D key will still produce lowercase d when Command is held down.
Command-Shift-D	Hold down Command and Shift while pressing the D key. Alpha Lock won't work for producing uppercase D in this case.
Control-X	Hold down Control while pressing the X key, with or without Shift or Alpha Lock (case doesn't matter with Control).

Notes and Warnings

Note: Paragraphs like this contain incidental information that may be of interest to curious readers but can safely be skipped.

Warning: Paragraphs like this are extremely important to read.

Chapter 1
The Mach Operating System

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's Release 2.0, with additional features both from NeXT and from later versions of CMU's Mach. NeXT-only features include the Bootstrap Server and loadable kernel servers. NeXT also added support for Sun® Microsystem's NFS®. 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 via 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's 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's objects and messages and the Objective-C objects and messages used in higher-level software kits such as the Application Kit. Mach's objects and messaging system 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 "Suggested Reading" for information about available UNIX 4.3BSD documentation). Individual Mach functions are described in detail in Chapter 4, "C Functions," and summarized in the *NeXT Technical Summaries* manual.

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 puts most services provided by the current UNIX kernel into independent user-level programs—the Mach kernel itself provides 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, kernel debuggers. Also, configuring the system is simply a matter of choosing which user-level services to initiate, rather than building and linking a new kernel.

Mach's movement 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, "Using Mach Messages."

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 allocation—each 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 protection—all 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). There's a nonnegative suspend count associated with each thread. The suspend count is 0 for runnable threads and positive for suspended threads.

Tasks may be suspended or resumed as a whole. A thread may only execute when both it and its task are runnable. Resuming a task doesn't cause all component threads to begin executing, but only those threads that aren't suspended.

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 functions 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 kernel 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 kernel calls on behalf of that task or thread. Access to a task's port indirectly permits access to all threads within that task via 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 their 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:

- The notify 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 has receive rights to this port and can get its value from the function **task_notify()**.

  Note that if a task's notify port is set to 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()**.

- The exception 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.

- The bootstrap 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). Send rights to this port are 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:

- The thread reply port, which is used in Mach remote procedure calls (remote procedure calls are described in Chapter 2). The **thread_reply()** function returns send and receive rights to the reply port of the calling thread.

- The thread exception 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 will be used instead.

Customarily, only threads within a task will 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's messaging facility is implemented by three kernel abstractions, *ports*, *port sets*, and *messages*:

- A *port* is a 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.

- A *port set* is a 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.

- A *message* is used 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: one 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 operating system kernel itself. 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 a task's task port.

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 port implies 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 port allows 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.

A task may hold just send rights, or both receive and 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 task within which the thread is executing. 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 Mach's 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 does the receive, the rights are handled as though the receive had been done before the task 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**. **port_name_t** implies that no port access rights are being transferred; the port is merely being referred to by its name. **port_all_t** 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 at most one port set at any time.

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 may not 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.

The Port Queue

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:

- Independence: A 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 transparency: As 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 access 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 accesses 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 via shared memory. However, if you use interprocess communication you're responsible for synchronizing the transmission and reception of the message. With Mach's 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 Mach's messaging system and allows space for memory-mapped files, to be discussed below.

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

Mach's 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 any of three ways—*copy*, *shared*, or *none*:

- *copy*: Pages 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.

- *shared*: Pages specified as shared can be read from and written to by both the parent and child.

- *none*: Pages 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's virtual memory management scheme 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 a process's virtual address space to that of physical memory. This is accomplished by mapping an unused portion of the receiving process's virtual address space 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 sending process's address space into the kernel's address space. From there, the message is copied into the receiver's address space.

Memory-Mapped Files

Memory-mapped files are a further benefit of Mach's virtual memory system. Under Mach, all or part of a disk file can be mapped onto a section of a process's 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. Thus, 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. **vm_allocate()** always adds new, zero-filled virtual memory in page-aligned, multiple of page-sized chunks. **malloc()** allocates approximately the size asked for (plus a few bytes) out of a preallocated heap. **calloc()** is the same as **malloc()** except that it zeros the memory before returning it. **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. There's one other consideration, however, which is the desirability of page-aligned storage. If the memory that's allocated is to be passed out-of-line in a 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. **vm_read()** 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. **vm_deallocate()** is then used to get rid of the two chunks of memory.

```
#include <mach.h>
#include <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()**. **vm_deallocate()** is then used to get rid of the two chunks of memory.

```
#include <mach.h>
#include <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 will 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 via 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).

```
#include <mach.h>
#include <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 or the one that's explicitly set using a function such as **cthread_priority()**. The current priority is the one that the thread is really executing at; this 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 MAXPRI_USER (defined in the header file **kern/sched.h**).

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 -Mach
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 3.84M  912K     0  S   p1  0:41 /NextAp
localhost> ps -m 174
USER      PID TT  %CPU STAT PRI    SYSTEM      USER  COMMAND
root      174 p1   1.8  S     16   0:15.76   0:19.17  /NextApps/Terminal -Mac
                   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're not 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

If you intend to do 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 thread kernel calls are designed to provide the low-level mechanisms that packages such as the C thread functions can be built with.

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 else 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, the operations provided by the library must be presumed to make unprotected use of shared data. Hence, 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 threads' execution streams. 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);
```

This operation also 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.

```
#include <stdio.h>
#include <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);
    cthread_exit(0);
}

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

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:

- Debugging. Debuggers 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 dumps. In 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 handling. Certain 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.

- Emulation. Generally, 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 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.

There are two major problems with the UNIX signal facility:

- 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.

Mach's exception handling model divides applications that use exceptions into two major classes:

- Error handlers: These components perform 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).

- Debuggers: These components examine 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 here 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 an exception's occurrence.

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 earlier in this chapter. The major design goals for this new facility were:

- A 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 via Mach kernel functions to avoid duplication of 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). We therefore implement them using a message-based RPC provided by Mach's communication facility. The remaining

terminate primitive is exactly Mach's **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 thread that caused the exception and the corresponding task.
- A machine-independent exception class (see the section "Exception Classification")
- 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**: send initial message
- **wait**: wait for and receive reply message
- **catch**: receive initial message
- **clear**: send 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 "Program Example: Exception Handling."

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 ever 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

Mach's 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 "Exception Classification") 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 which 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 thread's exception port. 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 via 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 only stops 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 via 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 via 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 above 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. Mach's 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.
- Debugging of a distributed application. 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 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 in the section "Debugger Attachment" 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 avoids confusion over which thread in the process is being manipulated by **ptrace**(); without this identification it's necessary to compare the context accessed by **ptrace()** to all other thread contexts in the task to determine this. This identification also make it possible to handle multiple concurrent exceptions; all the UNIX functions are restricted to one current signal per task, and hence preclude handling of multiple concurrent exceptions. Finally, 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. Our new classification divides all exceptions into six classes based on the causes and uses of the exceptions; further hardware and software specific distinctions can be made within these classes as needed. The six classes are:

- Bad Access: A user access to memory failed for some reason and the operating system was unable to recover (such as invalid memory, protection violation).

- Bad Instruction: A user executed an illegitimate instruction (such as an undefined instruction, reserved operand, privileged instruction).

- Arithmetic: A user arithmetic instruction failed for an arithmetic reason (such as overflow, underflow, divide by zero).

- Emulation: A user executed an instruction requiring software emulation.

- Software: A 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, 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.

- Debugger: Hardware 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.)
 */
#include <mach.h>
#include <sys/exception.h>
#include <cthreads.h>
#include <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. */
}
```

Chapter 2
Using Mach Messages

This chapter describes how to use Mach messages for interprocess communication (IPC). Programs can either send and receive Mach messages directly, or they can use MiG-generated remote procedure calls (RPCs), which appear to be simple function calls but which actually involve messages. Many kernel functions, such as **host_info()**, are really RPCs.

This chapter first describes the structure of all messages. It then discusses how to set up messages for direct sending. Finally, it discusses how to use MiG (Mach Interface Generator) to build a *Mach server*—a program that provides services to clients, using remote procedure calls. This chapter assumes that you understand the concepts of ports, port sets, and messages, which are described in Chapter 1, "The Mach Operating System."

You should usually use MiG to generate messages. MiG-generated code is easier for clients to use, and using MiG is a good way to define an interface that's separate from the implementation. However, you might want to build messages by hand if the messages are very simple or if you want fine control over communication details.

Message Structure

A message consists of a fixed header often followed by the message body. The body consists of alternating type descriptors and data items. Here's a typical message structure:

```
typedef struct {
    msg_header_t Head;
    msg_type_t   aType;
    int          a;
    msg_type_t   bType;
    int          b;
} Request;
```

Message Header

The C type definition for the message header is as follows (from the header file **sys/message.h**):

```
typedef struct {
        unsigned int msg_unused : 24,
                     msg_simple : 8;
        unsigned int msg_size;
        int          msg_type;
        port_t       msg_local_port;
        port_t       msg_remote_port;
        int          msg_id;
} msg_header_t;
```

The **msg_simple** field indicates whether the message is *simple* or *nonsimple*; the message is simple if its body contains neither ports nor out-of-line data (pointers).

The **msg_size** field specifies the size of the message to be sent, or the maximum size of the message that can be received. When a message is received, Mach sets **msg_size** to the size of the received message. The size includes the header and in-line data and is given in bytes.

The **msg_type** field specifies the general type of the message. For hand-built messages, it's MSG_TYPE_NORMAL. Other values for the **msg_type** field are defined in the header files **sys/message.h** and **sys/msg_type.h** (MiG-generated servers use the type MSG_TYPE_RPC).

The **msg_local_port** and **msg_remote_port** fields name the ports on which a message is to be received or sent. Before a message is sent, **msg_local_port** must be set to the port to which a reply, if any, should be sent; **msg_remote_port** must specify the port to which the message is being sent. Before a message is received, **msg_local_port** must be set to the port or port-set to receive on. When a message is received, Mach sets **msg_local_port** to the port the message is received on, and **msg_remote_port** to the port any reply should be sent to (the sender's **msg_local_port**).

The **msg_id** field can be used to identify the meaning of the message to the intended recipient. For example, a program that can send two kinds of messages should set the **msg_id** field to indicate to the receiver which kind of message is being sent. MiG automatically generates values for the **msg_id** field.

Message Body

The body of a message consists of an array of type descriptors and data. Each type descriptor contains the following structure:

```
typedef struct  {
    unsigned int
        msg_type_name : 8,       /* Type of data */
        msg_type_size : 8,       /* Number of bits per item */
        msg_type_number : 12,    /* Number of items */
        msg_type_inline : 1,     /* If true, data follows; else a
                                    pointer to the data follows */
        msg_type_longform : 1,   /* Name, size, number follow */
        msg_type_deallocate : 1, /* Deallocate port rights or
                                    memory */
        msg_type_unused : 1;
} msg_type_t;
```

msg_type_name describes the basic type of data comprising this object. There are several system-defined data types, including:

- Ports, including combinations of send and receive rights.
- Port and port set names. This is the same language data type as port rights, but the message only carries a task's name for a port and doesn't cause any transferral of rights.
- Simple data types, such as integers, characters, and floating-point values.

msg_type_size indicates the size in bits of the basic object named in the **msg_type_name** field.

msg_type_number indicates the number of items of the basic data type present after the type descriptor.

msg_type_inline indicates that the actual data is included after the type descriptor; otherwise, the word following the descriptor is a pointer to the data to be sent.

msg_type_longform indicates that the name, size, and number fields were too long to fit into the **msg_type_t** structure. These fields instead follow the **msg_type_t** structure, and the type descriptor consists of a **msg_type_long_t**:

```
typedef struct {
    msg_type_t  msg_type_header;
    short       msg_type_long_name;
    short       msg_type_long_size;
    int         msg_type_long_number;
} msg_type_long_t;
```

msg_type_deallocate indicates that Mach should deallocate this data item from the sender's address space after the message is queued. You can deallocate only port rights or out-of-line data.

A data item, an array of data items, or a pointer to data follows each type descriptor.

Creating Messages by Hand

This section shows how to create messages to be sent using **msg_send()** or **msg_rpc()**. You don't usually have to set up messages by hand. For example, although Mach servers call **msg_send()**, almost all the message fields are already set up in MiG-generated code. However, this section might be useful to you if you want to send messages without using MiG, or if you want to read through MiG-generated code.

Setting Up a Simple Message

As described earlier, a message is simple if its body doesn't contain any ports or out-of-line data (pointers). The **msg_remote_port** field must contain the port the message is to be sent to. The **msg_local_port** field should be set to the port a reply message (if any) is expected on.

The following example shows the creation of a simple message. Because every item in the body of the message is of the same type (**int**), only one type descriptor is necessary, even though the items are in two different fields.

```
#define BEGIN_MSG 0 /* Constants to identify the different messages */
#define END_MSG 1
#define REPLY_MSG 2

#define MAXDATA 3

struct simp_msg_struct {
    msg_header_t  h;                /* message header */
    msg_type_t    t;                /* type descriptor */
    int           inline_data1;     /* start of data array */
    int           inline_data2[2];
};
struct simp_msg_struct  msg_xmt;
port_t                  comm_port, reply_port;

/* Fill in the message header. */
msg_xmt.h.msg_simple = TRUE;
msg_xmt.h.msg_size = sizeof(struct simp_msg_struct);
msg_xmt.h.msg_type = MSG_TYPE_NORMAL;
msg_xmt.h.msg_local_port = reply_port;
msg_xmt.h.msg_remote_port = comm_port;
msg_xmt.h.msg_id = BEGIN_MSG;
```

```
/* Fill in the type descriptor. */
msg_xmt.t.msg_type_name = MSG_TYPE_INTEGER_32;
msg_xmt.t.msg_type_size = 32;
msg_xmt.t.msg_type_number = MAXDATA;
msg_xmt.t.msg_type_inline = TRUE;
msg_xmt.t.msg_type_longform = FALSE;
msg_xmt.t.msg_type_deallocate = FALSE;

/* Fill in the array of data items. */
msg_xmt.inline_data1 = value1;
msg_xmt.inline_data2[1] = value2;
msg_xmt.inline_data2[2] = value3;
```

Setting Up a Nonsimple Message

A message is *nonsimple* if its body contains ports or out-of-line data. The most common reason for sending data out-of-line is that the data block is very large or of variable size.

In-line data is copied by the sender into the message structure and then often copied out of the message by the receiver. Out-of-line data, however, is mapped by the kernel from the address space of the sender to the address space of the receiver. No actual copying of out-of-line data is done unless one of the two tasks subsequently modifies the data.

This example shows how to construct a message containing out-of-line data:

```
#define BEGIN_MSG 0 /* Constants to identify the different messages */
#define END_MSG 1
#define REPLY_MSG 2

#define MAXDATA 3

struct ool_msg_struct {
    msg_header_t  h;                /* message header */
    msg_type_t    t;                /* type descriptor */
    int          *out_of_line_data; /* address of data */
};
struct ool_msg_struct   msg_xmt;
port_t                  comm_port, reply_port;

/* Fill in the message header. */
msg_xmt.h.msg_simple = FALSE;
msg_xmt.h.msg_size = sizeof(struct ool_msg_struct);
msg_xmt.h.msg_type = MSG_TYPE_NORMAL;
msg_xmt.h.msg_local_port = reply_port;
msg_xmt.h.msg_remote_port = comm_port;
msg_xmt.h.msg_id = BEGIN_MSG;
```

```
/* Fill in the type descriptor. */
msg_xmt.t.msg_type_name = MSG_TYPE_INTEGER_32;
msg_xmt.t.msg_type_size = 32;
msg_xmt.t.msg_type_number = MAXDATA;
msg_xmt.t.msg_type_inline = FALSE;
msg_xmt.t.msg_type_longform = FALSE;
msg_xmt.t.msg_type_deallocate = FALSE;

/* Fill in the out-of-line data. */
msg_xmt.out_of_line_data = (int *)&mydata;
```

The fields that change values from those in the simple message example are **msg_simple**, **msg_type_inline**, and possibly **msg_type_deallocate**. The **msg_type_name**, **msg_type_size**, and **msg_type_number** fields remain the same as before, so that Mach can determine how much memory to map.

The **msg_remote_port** field must contain the port the message is to be sent to. The **msg_local_port** field should be set to the port a reply message (if any) is expected on.

Setting Up a Reply Message

Once a message has been received, a reply message may have to be sent to the sender of the received message. In the example below, the reply message, **msg_xmt**, is simply a **msg_header_t** since no data is required. The **msg_remote_port** field—where to send the message—is set to the remote port of the previously received message (which Mach set to the previous sender's **msg_local_port** field). The outgoing message's **msg_local_port** field is set to PORT_NULL because no reply to this reply message is expected.

```
#define BEGIN_MSG 0 /* Constants to identify the different messages */
#define END_MSG 1
#define REPLY_MSG 2

struct simp_msg_struct {            /* format of received message */
    msg_header_t  h;                /* message header */
    msg_type_t    t;                /* type descriptor */
    int           inline_data1;     /* start of data array */
    int           inline_data2[2];
};
msg_header_t              msg_xmt;
struct simp_msg_struct   *msg_rcv;

msg_xmt.h.msg_remote_port = msg_rcv->h.msg_remote_port;
msg_xmt.h.msg_local_port = PORT_NULL;  /* no reply expected */
msg_xmt.h.msg_id = REPLY_MSG;
msg_xmt.h.msg_size = sizeof(msg_header_t);
msg_xmt.h.msg_type = MSG_TYPE_NORMAL;
msg_xmt.h.msg_simple = TRUE;
```

The Mach Interface Generator

The Mach Interface Generator (known as MiG) is a program that generates remote procedure call (RPC) code for communication between a client and a server process. The operations of sending a message and receiving a reply are represented as a single remote procedure call.

For example, if a program makes a call to **host_info()**, it actually calls a library routine that sends a message to the Mach kernel and then waits to receive a reply message. After the Mach kernel sends a reply message containing the information, the library routine takes the data out of the reply message and returns it to the program in parameters to the **host_info()** call. However, the program sees none of this complexity—it merely makes the following function call:

```
ret = host_info(host_self(), HOST_SCHED_INFO,
                (host_info_t)&sched_info, &sched_count);
```

A Mach server executes as a separate task and communicates with its clients by sending Mach interprocess communication (IPC) messages. As you can see from the previous sections in this chapter, Mach messages are fairly complex. The MiG program is designed to automatically generate procedures in C to pack and send, or receive and unpack the messages used to communicate between processes.

Because of the complexity of sending and decoding messages, Mach remote procedure calls are an order of magnitude slower than real function calls, even if the server is on the local machine. Calls to servers on remote machines take longer. However, Mach RPC has the advantages of the separation of interface and implementation, and of network transparency.

Using MiG, you can create RPC interfaces for sending messages between tasks on the local machine, or between tasks on separate machines in a network. In the network environment, MiG both encodes messages to be transmitted and decodes them upon arrival at the destination node, taking into account dissimilarities in machine architecture.

MiG is especially useful if you're faced with a heterogeneous network environment. Without MiG, you're responsible for providing routines to translate messages between two machines with different data representations. Using MiG, you need only specify the calling arguments of the procedure and the procedure's return variables. The low-level routines required to translate messages between these machines are then generated automatically.

MiG is flexible enough to describe most data structures that might be sent as messages between processes. MiG supports the data types boolean, character, signed and unsigned integers, integer subranges, strings, reals, and communication port types. MiG also supports the limited creation of new data types through the use of enumerations, fixed- and variable-sized arrays, records, pointers to these types, and unions.

Creating Mach Servers with MiG

To create a Mach server, you must provide a specification file defining parameters of both the message passing interface and the procedure call interface. MiG then generates three files from the specification file:

- User interface file (*xxx***User.c**, where *xxx* is the subsystem name). This file should be compiled and linked into the client program. It implements and exports the procedures and functions for sending and receiving the appropriate messages to and from the server.

- User header file (*xxx***.h**). This file defines the functions to be called by a client of the server. It is included in the user interface file (*xxx***User.c**) and defines the types and routines needed at compilation time.

- Server interface file (*xxx***Server.c**). This file should be compiled and linked into the server process. It extracts the input parameters from an IPC message, and calls a server procedure to perform the operation. When the server procedure or function returns, the Server interface also gathers the output parameters and formats a reply message.

Besides the specification file, you must write at least two functions for the Mach server. One is the main routine of the server, which registers the server and then goes into a loop that receives a message, calls the MiG-generated code to process the request, and sends a reply message. You must also write one function for each remote procedure call, so that the MiG-generated server code can call the appropriate function for each request.

In addition, you should provide a library routine that clients can use to look up your server. For example, the kernel-server loader has a routine called **kern_loader_look_up()** that clients call to obtain the kernel-server loader's port. This port must be specified as the first argument in every RPC to the kernel-server loader.

You can register your server with either the Network Name Server or the Bootstrap Server, depending on whether you want your server to be available to other machines on a network. The Bootstrap Server allows only processes that are on the local machine (or a subset of local processes) to get your server's port. For example, the sound driver registers its port with the Bootstrap Server so that only processes descended from the local machine's Login Window can control sound. The Network Name Server allows tasks on remote machines to get the server's port. See Chapter 4, "C Functions," for more information on NetworkName Server and Bootstrap Server functions.

The Client's View

This section describes how clients use servers, so that you can better create and document your own server.

Before a client can make remote procedure calls to the server, it must find the server's port. If the server doesn't provide a library function to do this lookup, then the client must call either **netname_look_up()** or **bootstrap_look_up()**, supplying the name of the server.

When a client makes a remote procedure call, it appears to be a simple function call. The return type depends on whether the RPC is defined in the server's MiG specification file to be a routine, procedure, or function (as described later in this chapter).

The most convenient interfaces are to routines, which return a value of type **kern_return_t**. The returned value is either KERN_SUCCESS or a MiG, Mach, or server-specific error code. MiG and Mach error codes can be interpreted by **mach_error()** and **mach_error_string()**.

Procedure and function RPCs are less convenient than routines because they don't directly return error codes. Instead, the client must provide an error handling routine named either **MsgError()** or whatever name the server developer specified in the server's MiG specification file. The error handling routine must be defined as follows:

> void *error_proc*(kern_return_t *error_code*)

Common Error Codes

The most common system error that an RPC returns to a client is an invalid port. This can mean several things:

- The request port (usually the first parameter in the RPC) is an invalid port, or the client doesn't have send rights for it.
- The reply port is invalid or lacks receive rights. (This problem can't occur unless the client provides the reply port; usually the system provides it.)
- Another port that the client is passing in the message is invalid.
- A port that's being passed back to the client is invalid.

Another system error a client might receive is a timeout. This can happen only if a timeout is specified in an argument or in the server's specification file, and usually doesn't happen unless the server is on a different machine from the client.

MiG errors, which are defined in the header file **mig_errors.h**, usually occur only if the client is using a different version of the interface than the server.

Out-of-Line Data

When making specific interface calls the client should be aware if any out-of-line data is being returned to it. If so, it might want to deallocate the space with a call to **vm_deallocate()**.

Compiling the Client

The client must be compiled and linked with the *xxx***User.c** and *xxx***.h** files that MiG produced from the server's specification file. The client should also include or be linked with any files that are necessary to communicate with the server (such as the file containing the routine that looks up the server). For example, clients of the kernel-server loader must be linked against the **kernload** library, which supplies all non-RPC kernel-server loader functions.

Programming Example

This example shows the implementation of a very simple server that adds two integers and returns the answer. The files used to produce this server and a sample client program, including make files, are located in the directory **/NextLibrary/Documentation/NextDev/Examples/MiG**.

The user-written files required for the server are the following:

- The MiG specification file (**Server/add.defs**)
- The type definition file, which is included by both the server and the client (**Library/add_types.h**)
- The implementation file, which contains the server's main loop and the function that does the addition (**Server/add_server_main.c**)

Once the server has been generated, any client programs need to have the following files:

- The MiG-generated user interface file, in a form that can be compiled or linked into the client program (**Library/addUser.o**). (If this file isn't already compiled, then the client program also needs access to the ".h" file that was generated by MiG; for example, **add.h**.)
- The type definition file (**Library/add_types.h**)
- One or more files containing the main parts of the client program (**Client/add.c**).

Below is **add.defs**, a simple MiG specification file. It declares the remote routine **add2nums()**, which takes as arguments the request port (the default first argument to every MiG operation), two integers, and a pointer to a third integer. Because all types mentioned in **add.defs** are already defined in the included header file **std_types.defs**, it isn't necessary to define any types directly in **add.defs**.

```
/* add.defs:  MiG definition file for add server */

subsystem add 0;

/* Get standard definitions of int and port_t. */
#include <std_types.defs>

routine add2nums(server: port_t; a:int; b:int; out c:int);
```

The header file **std_types.defs** defines **int** and **port_t** as the following:

```
type int = MSG_TYPE_INTEGER_32;
type port_t = MSG_TYPE_PORT;
```

The header file **add_types.h** contains definitions needed by both the client and the server:

```
/* add_types.h:  Definitions for add server */
#import <mach.h>

#define ADD_SERVER_NAME "Addition-Server"

extern port_t add_look_up(void);
```

The code that does the work for the server is in the file **add_server_main.c**. It contains a main loop and the function that performs the addition. It also defines a message structure to be passed into the MiG-generated message server, **add_server()**. Such a message structure should start with a **msg_header_t** field, and it must be at least as large as the largest possible incoming message (unless you use the RCV_LARGE option to **msg_receive()** to dynamically determine message size). You can see the structures of all possible incoming messages by running **mig** on the ".defs" file and looking at the **Request** structures that are defined in the generated user file (**addUser.c**).

```
/* add_server_main.c:  Main loop and implementation of add server */

#import <mach.h>
#import <sys/message.h>
#import "../Library/add_types.h"

void         server_loop(port_t port);
/* defined by MiG: */
boolean_t    add_server(msg_header_t *in, msg_header_t *out);
```

```
/* from Request types in addUser.c */
struct message {
    msg_header_t    head;         /* standard header field */
    msg_type_t      arg1_type;    /* first arg type */
    int             arg1;         /* first arg */
    msg_type_t      arg2_type;    /* second arg type */
    int             arg2;         /* second arg */
};

main()
{
    port_t        server_port;
    kern_return_t r;

    /* Register with the Network Name Server. */
    r = port_allocate(task_self(), &server_port);
    if (r != KERN_SUCCESS) {
        mach_error("port_allocate failed", r);
        exit(1);
    }
    r = netname_check_in(name_server_port, ADD_SERVER_NAME,
        PORT_NULL, server_port);
    if (r != KERN_SUCCESS) {
        mach_error("netname_check_in failed", r);
        exit(1);
    }

    /* Enter our main loop. */
    server_loop(server_port);
}

void server_loop(port_t port)
{
    struct message   msg, reply;
    kern_return_t    ret;

    while (TRUE)
    {
        /* Receive a request from a client. */
        msg.head.msg_local_port = port;
        msg.head.msg_size = sizeof(struct message);
        ret = msg_receive(&msg.head, MSG_OPTION_NONE, 0);
        if (ret != RCV_SUCCESS) /* ignore errors */;

        /* Feed the request into the server. */
        (void)add_server((msg_header_t *)&msg,
            (msg_header_t *)&reply);

        /* Send a reply to the client. */
        reply.head.msg_local_port = port;
        ret = msg_send(&reply.head, MSG_OPTION_NONE, 0);
        if (ret != SEND_SUCCESS) /* ignore errors */;
    }
}
```

```
/*
 * This function is called by add_server, which was created by MiG.
 * It is NOT directly called by any client process.
 */
kern_return_t add2nums(port_t server, int n1, int n2, int *n3)
{
    *n3 = n1 + n2;
    return KERN_SUCCESS;
}
```

In general, your message receive loop should return a reply for every message it receives unless the reply message returned from the MiG-generated server has MIG_NO_REPLY in its **RetCode** field. MIG_NO_REPLY is used only when the received message was part of an RPC that never expects a return message (a **simpleprocedure** or **simplefunction**, both of which are defined later in this chapter). For example:

```
(void)my_server(&msg, &reply);
ret_code = reply->RetCode;

if (reply->RetCode == MIG_NO_REPLY)
    ret_code = KERN_SUCCESS;
else
    ret_code = msg_send(&reply->Head, MSG_OPTION_NONE, 0);
```

Finally, a typical client process, such as **Client/add.c**, makes the RPC as follows:

```
. . .
#import "../Library/add_types.h"
int           n1, n2, n3;
kern_return_t ret;
port_t        server;
. . .
/* Find the server. */
server = add_look_up();
if (server == PORT_NULL)
{
    fprintf(stderr, "Couldn't find the add server.\n");
    exit(2);
}

/* Send a message to the server. */
ret = add2nums(server, n1, n2, &n3);
if (ret != KERN_SUCCESS)
    printf("Call to add2nums failed.\n");
else
    printf("According to the server, %d + %d = %d.\n", n1, n2, n3);
```

Note that although the RPC looks like it directly calls **add2nums()** in the server, it really doesn't. The client instead sends a message that's received in **server_loop()**, which calls **add_server()**. **add_server()** calls **add2nums()** and passes the result back to the client in a message.

Making a function such as **add2nums()** an RPC gives the advantages of network independence, interface independence, and automatic type checking, at the expense of some complexity in the server.

MiG Specification File

You must first write a MiG specification file to specify the details of the procedure arguments and the messages to be used. A MiG specification file contains the following components, some of which may be omitted:

- Subsystem identification
- Type declarations
- Import declarations
- Operation descriptions
- Options declarations

The subsystem identification should appear first for clarity. Types must be declared before they're used. Code is generated for the operations and import declarations in the order in which they appear in the specification files. Options affect the operations that follow them.

See the section "Programming Example" for an example of a complete subsystem definition.

Subsystem Identification

The subsystem identification statement has the following form:

subsystem *sys message_base_id* ;

sys is the name of the subsystem. It's used as the prefix for all generated file names. The user file name will be *sys***User.c**, the user header file will be *sys***.h**, and the server file will be *sys***Server.c**.

message_base_id is a decimal integer that's used as the IPC message ID of the first operation in the specification file. Operations are numbered sequentially beginning with this base. The MiG-generated server function checks the message ID of an incoming message to make sure that it's no less than *message_base_id* and no greater than *message_base_id* + *num_messages* –1, where *num_messages* is the number of messages understood by the server.

Several servers can use just one message receive loop as long as they have different subsystem numbers (and they have few enough messages so that message IDs don't overlap). The message receive loop should call each MiG-generated server function in turn until one of them returns true (indicating the message ID is in the range understood by that server.) Once a MiG-generated server function has returned true or all the servers have returned false, the receive-serve-send loop should send a reply (unless the reply message returned by the server function has MIG_NO_REPLY in its **RetCode** field).

Example:

```
subsystem random 500;
```

Type Declarations

Simple Types

A simple type declaration has the following form:

> **type** *user_type_name* = *type_desc* [*translation_info*]

where a *type_desc* is either a previously defined *user_type_name* or an *ipc_type_desc,* which has one of the following forms:

> *ipc_type_name*
> (*ipc_type_name* [, *size* [, **dealloc**]])

The *user_type_name* is the name of a C type that will be used for some parameters of the calls exported by the user interface file. The *ipc_type_desc* of simple types are enclosed in parentheses and consist of an IPC type name, decimal integer, or integer expression that's the number of bits in the IPC type and optionally, the **dealloc** keyword.

The standard system-defined IPC type names are:

MSG_TYPE_BOOLEAN
MSG_TYPE_BIT
MSG_TYPE_BYTE
MSG_TYPE_CHAR
MSG_TYPE_INTEGER_8
MSG_TYPE_INTEGER_16
MSG_TYPE_INTEGER_32
MSG_TYPE_REAL
MSG_TYPE_STRING
MSG_TYPE_PORT
MSG_TYPE_PORT_ALL
MSG_TYPE_UNSTRUCTURED

The current set of these type names is contained in the header file **sys/message.h**, which defines all the message-related types needed by a user of the Mach kernel. The programmer may define additional types. If the *ipc_type_name* is a system-defined one other than MSG_TYPE_STRING, MSG_TYPE_UNSTRUCTURED, or MSG_TYPE_REAL, *size* (the bit length) need not be specified and the parentheses can be omitted.

The **dealloc** keyword controls the treatment of ports and pointers after the messages they're associated with have been sent. **dealloc** causes the deallocation bit in the IPC message to be set on; otherwise, it's off. If **dealloc** is used with a port, the port is deallocated after the message is sent. If **dealloc** is used with a pointer, the memory that the pointer references will be deallocated after the message has been sent. An error results if **dealloc** is used with any argument other than a port or a pointer.

Some examples of simple type declarations are:

```
type int = MSG_TYPE_INTEGER_32;
type my_string = (MSG_TYPE_STRING,8*80);
type kern_return_t = int;
type disposable_port = (MSG_TYPE_PORT_ALL,32,dealloc);
```

The MiG-generated code assumes that the C types **my_string**, **kern_return_t**, and **disposable_port** are defined in a compatible way by a programmer-provided header file. The basic C and Mach types are defined in the file **std_types.defs**.

MiG assumes that any variable of type MSG_TYPE_STRING is declared as a C **char** * or **char** *array*[*n*]. Thus it generates code for a parameter passed by reference and uses **strncpy**() for assignment statements.

Optional *translation_info* information describing procedures for translating or deallocating values of the type may appear after the type definition information:

- Translation functions, **intran** and **outtran**, allow the type as seen by the user process and the server process to be different.
- Destructor functions allow the server code to automatically deallocate input types after they have been used.

For example:

```
type task_t = (MSG_TYPE_PORT,32)
intran:     i_task_t PortToTask(task_t)
outtran:    task_t TaskToPort(i_task_t)
destructor: DeallocT(i_task_t)
;
```

Note: Because *translation_info* is part of the type declaration, the semicolon (**;**) doesn't appear until after the end of *translation_info*.

In this example, **task_t**, which is the type seen by the user code, is defined as a port in the message. The type seen by the server code is **i_task_t**, which is a data structure used by the server to store information about each task it's serving. The **intran** function **PortToTask**() translates values of type **task_t** to **i_task_t** on receipt by the server process. The **outtran** function **TaskToPort**() translates values of type **i_task_t** to type **task_t** before return. The destructor function **DeallocT**() is called on the translated input parameter, **i_task_type**, after the return from the server procedure and can be used to deallocate any or all parts of the internal variable. The destructor function won't be called if the parameter is also an **out** parameter (as described in the section "Operation Descriptions" below), because the correct time to deallocate an **out** parameter is after the reply message has been sent, which isn't code that's generated by MiG. A destructor function can also be used independently of the translation routines. For example, if a large out-of-line data segment is passed to the server it could use a destructor function to deallocate the memory after the data was used.

Although calls to these functions are generated automatically by MiG, the function definitions must be hand-coded and imported using:

```
i_task_t PortToTask(task_t x)
task_t TaskToPort(i_task_t y)
void DeallocT(i_task_t y)
```

Structured Types

Three kinds of structured types are recognized: arrays, structures, and pointers. Definitions of arrays and structures have the following syntax:

array [*size*] **of** *comp_type_desc*
array [* : *maxsize*] **of** *comp_type_desc*
struct [*size*] **of** *comp_type_desc*

where *comp_type_desc* may be a simple *type_desc* or may be an **array** or **struct** type, and *size* may be a decimal integer constant or expression. The second array form specifies that a variable-length array is to be passed in-line in the message. In this form *maxsize* is the maximum length of the item. Currently, only one variable-length array may be passed per message. For variable-length arrays an additional count parameter is generated to specify how much of the array is actually being used.

If a type is declared as an **array**, the C type must also be an array, since the MiG RPC code will treat the user type as an array (that is, MiG will assume that the user type is passed by reference and it will generate special code for array assignments). A variable declared as a **struct** is assumed to be passed by value and treated as a C structure in assignment statements. There is no way to specify the fields of a C structure to MiG. The *size* and *type_desc* are used only to give the size of the structure. The following example shows how to declare a C structure as a **struct**.

```
/* declaration in MiG .defs file */
type short = MSG_TYPE_INTEGER_16;
type port_t = MSG_TYPE_PORT;
type lock_struct = struct [9] of short;
routine fl_message(server_port: port_t; inout arg: lock_struct);
```

```
/* declaration in C code */
typedef struct {
    short l_type;
    short l_whence;
    long l_start;
    long l_len;
    short l_pid;
    long l_hostid;
} lock_struct;
```

Pointer Types

In the definition of pointer types, the symbol ^ precedes a simple, array, or structure definition.

^ *comp_type_desc*
^ **array** [*size*] **of** *comp_type_desc*
^ **struct** [*size*] **of** *com_type_desc*

size may be left blank or be *. In either case, the array or structure is of variable size, and a parameter is defined immediately following the array parameter to contain its size. Data types declared as pointers are sent out-of-line in the message. Since sending out-of-line data is considerably more expensive than sending in-line data, pointer types should be used only for large or variable amounts of data. A call that returns an out-of-line item allocates the necessary space in the user's virtual memory. It's up to the user to call **vm_deallocate()** on this memory when finished with the data.

Some examples of complex types are:

```
type procids = array [10] of int;
type procidinfo = struct [5*10] of (MSG_TYPE_INTEGER_32);
type vardata = array [ * : 1024 ] of int;
type array_by_value = struct [1] of array [20] of (MSG_TYPE_CHAR);
type page_ptr = ^ array [4096] of (MSG_TYPE_INTEGER_32);
type var_array = ^ array [] of int;
```

Import Declarations

If any of the *user_type_name*s or *server_type_name*s are other than the standard C types (such as **int** and **char**), C type specification files must be imported into the user interface and server interface files so that they'll compile. The import declarations specify files that are imported into the modules generated by MiG.

An import declaration has one of the following forms:

import *file_name*;
uimport *file_name*;
simport *file_name*;

where *file_name* has the same form as file name specifications in **#include** statements (that is, <*file_name*> or "*file_name*").

For example:

```
import "my_defs.h";
import "/usr/mach/include/cthreads.h";
import <cthreads.h>;
```

import declarations are included in both the user-side and server-side code. **uimport** declarations are included in just the user side. **simport** declarations are included in just the server side.

Operation Descriptions

Any of five standard operations may be specified by using the following keywords:

function
routine
procedure
simpleprocedure
simpleroutine

One other keyword, **skip**, may be used in place of a standard operation.

Functions and routines have a return value; procedures don't. Routines are functions whose result is of type **kern_return_t**. This result indicates whether the requested operation was successfully completed. If a routine returns a value other than KERN_SUCCESS the reply message won't include any of the reply parameters except the error code. Neither procedures nor functions return indications of errors directly; instead they call a hand-coded error function in the client. The name of the error function is **MsgError()**, by default; you can specify another name using the **error** declaration in the MiG specification file.

Simple procedures and simple routines send a message to the server but don't expect a reply. The return value of a simple routine is the value returned by the function **msg_send()**. Simple routines or simple procedures are used when asynchronous communication with a server is desired. All the rest of the operations wait for a reply before returning to the caller.

The syntax of the **procedure**, **simpleprocedure**, **simpleroutine**, and **routine** statements are identical. The syntax of **function** is also the same except for the type name of the value of the function. The general syntax of an operation definition for everything except **function** has the following form:

operation_type operation_name (*parameter_list*) ;

For **function** the form is:

function *operation_name* (*parameter_list*) : *function_value_type* ;

The *parameter_list* is a list of parameter names and types separated by a semicolon. The form of each parameter is:

[*specification*] *var_name* : *type_description* [, **dealloc**]

If not omitted, *specification* must be one of the following:

in
out
inout
requestport
replyport
waittime
msgtype

type_description can be any *user_type_name* or a complete type description (see the section "Type Declarations").

The first unspecified parameter in any operation statement is assumed be the **requestport** unless a **requestport** parameter was already specified. This is the port that the message is to be sent to. If a **replyport** parameter is specified, it will be used as the port that the reply message is sent to. If no **replyport** parameter is specified a per-thread global port is used for the reply message.

The keywords **in**, **out**, and **inout** are optional and indicate the direction of the parameter. The keyword **in** is used with parameters that are to be sent to the server. The keyword **out** is used with parameters to be returned by the server. The keyword **inout** is used with parameters to be both sent and returned. If no such keyword is given, the default is **in**.

The keywords **waittime**, **replyport**, and **msgtype** may be used to specify dynamic values for the wait time, the reply port, or the message type for this message. These parameters aren't passed to the server code, but are used when generating the send and receive calls. The **requestport** and **replyport** parameters must be of types that resolve to MSG_TYPE_PORT. The **waittime** and **msgtype** parameters must resolve to MSG_TYPE_INTEGER_32.

The keyword **skip** is provided to allowed a procedure to be removed from a subsystem without causing all the subsequent message interfaces to be renumbered. It causes no code to be generated, but uses up a **msg_id** number.

Here are some examples:

```
procedure init_seed (       server_port : port_t;
                            seed        : dbl);
routine get_random (        server_port : port_t;
                            out num     : int);
simpleroutine use_random (  server_port : port_t;
                            info_seed   : string80;
                            info        : comp_arr;
                            info_1      : words);
simpleprocedure exit (      server_port : port_t);
```

See the section "Programming Example" for an example of a complete subsystem definition.

Options Declarations

Several special-purpose options about the generated code may be specified. Defaults are available for each, and simple interfaces don't usually need to change them. First-time readers may want to skip this section. These options may occur more than once in the specification file. Each time an option declaration appears, it sets that option for all the following operations.

waittime Specification

The **waittime** specification has one of the following two forms:

waittime *time* **;**
nowaittime ;

The word **waittime** is followed by an integer or an identifier that specifies the maximum time in milliseconds that the user code will wait for a reply from the server. If an identifier is used, it should be declared as an **extern** variable by some module in the user code. If the **waittime** option is omitted, or if the **nowaittime** statement is seen, the RPC doesn't return until a message is received.

The timeout value for the **msg_receive()** can alternatively be controlled by using a **waittime** parameter to the RPC.

msgtype Specification

The **msgtype** specification has the following form:

msgtype *msgtype_value* **;**

msgtype_value may be one of the values from the header file **msg_type.h**. The available types are MSG_TYPE_RPC and MSG_TYPE_NORMAL. The MSG_TYPE_RPC is set to a correct value by default; this value normally shouldn't be changed. The value MSG_TYPE_NORMAL can be used to reset the **msgtype** option.

The **msgtype** value for the **msg_send()** can alternatively be controlled by using a **msgtype** parameter to the RPC.

error Specification

The **error** specification has the following form:

error *error_proc* ;

The **error** specification is used to specify how message-passing errors are to be handled for operations other than routines or simple routines. In all types of routines, any message errors are returned in the return value of the routine. For operations of types other than routines, the procedure *error_proc* is called when a message error is detected. The procedure specified by *error_proc* has to be supplied by the user, and must be of the form:

void *error_proc* (kern_return_t *error_code*)

If the **error** specification is omitted, *error_proc* is set to **MsgError()**.

serverprefix Specification

The **serverprefix** specification has the following form:

serverprefix *string* ;

The word **serverprefix** is followed by an identifier string that will be prepended to the actual names of all the following server-side functions implementing the message operations. This is particularly useful when it's necessary for the user-side and server-side functions to have different names, as must be the case when a server is also a user of copies of itself.

userprefix Specification

The **userprefix** specification has the following form:

userprefix *string* ;

The word **userprefix** is followed by an identifier string that will be prepended to the actual names of all the following user-side functions calling the message operations. **serverprefix** should usually be used when different names are needed for the user and server functions, but **userprefix** is also available for the sake of completeness.

rcsid Specification

The **rcsid** specification has the following form:

> **rcsid** *string* **;**

This specification causes a string variable *sys*_**user_rscid** in the user module and *sys*_**server_rcsid** in the server module to be set equal to the input string. The subsystem name *sys* was described earlier in the section "Subsystem Identification."

Syntax Summary

This section summarizes the syntax of MiG specification files. Note the following conventions:

- Terminal symbols (literals) are shown in bold.
- Nonterminal symbols are shown in italic.
- Alternatives are listed on separate lines.
- Brackets indicate zero or one occurrence of the bracketed item. Ellipsis (...) indicates one or more repetitions of the preceding item. Brackets and ellipsis combined, as in [*item* ...] indicate zero, one, or more repetitions of the item.
- Types must be declared before they're used.
- Comments may be included in a ".defs" file if surrounded by /* and */. Comments are parsed and removed by the C preprocessor.

specification_file:
    subsystem_description [*waittime_description*] [*msgtype_description*]
        [*error_description*] [*server_prefix_description*] [*user_prefix_description*]
        [*rscid_description*] [*type_description* ...] [*import_declaration* ...]
        operation_description ...

subsystem_description:
    subsystem *identifier decimal_integer* **;**

waittime_description:
waittime *time_value* **;**
nowaittime ;

time_value:
MSG_TYPE_INTEGER_32

msgtype_description:
msgtype *msgtype_value* **;**

msgtype_value:
MSG_TYPE_RPC
MSG_TYPE_NORMAL

error_description:
error *error_procedure* **;**

server_prefix_description:
serverprefix *identifier_string* **;**

user_prefix_description:
userprefix *identifier_string* **;**

rcsid_description:
rcsid *identifier_string* **;**

type_description:
type *type_definition* **;**

import_declaration:
import_keyword include_name **;**

import_keyword:
import
uimport
simport

include_name:
"*file_name***"**
<*file_name***>**

operation_description:
routine_description
simpleroutine_description
procedure_description
simpleprocedure_description
function_description

routine_description:
routine *argument_list* **;**

simpleroutine_description:
simpleroutine *argument_list* **;**

procedure_description:
procedure *argument_list* **;**

simpleprocedure_description:
simpleprocedure *argument_list* **;**

function_description:
function *argument_list* **:** *type_definition* **;**

argument_list:
([*argument_definition*] [**;** *argument_definition*] ...)

argument_definition:
[*specification*] *identifier* **:** *type_definition* [**, dealloc**]

specification:
in
out
inout
requestport
replyport
waittime
msgtype

type_definition:
identifier = [^] [*repetition* ...] *ipc_info* [*translation*]

repetition:
array **[** [*size*] **] of**
struct **[** [*size*] **] of**

size:
integer_expression

integer_expression:
integer_expression + *integer_expression*
integer_expression – *integer_expression*
integer_expression * *integer_expression*
integer_expression / *integer_expression*
(*integer_expression*)
integer

ipc_info:
(*ipc_type_name* , *size_in_bits* [**, dealloc**])
ipc_type_name
identifier

translation:
[*input_function*] [*output_function*] [*destructor_function*]

input_function:
intran : *identifier*

output_function:
outtran : *identifier*

destructor_function:
destructor : *identifier*

ipc_type_name:
integer
manifest_constant

Compiling MiG Specification Files

To compile a MiG specification file, specify the name of your ".defs" file (or files) and any switches as arguments to the **mig** command. For example:

```
mig -v random.defs
```

If **random** is the subsystem name declared in the definitions file, MiG will produce the files **random.h**, **randomUser.c**, and **randomServer.c** as output. If the **-MD** switch was given, a **random.d** file will also be generated.

MiG recognizes the following switches:

[p,P]	If **p**, use two-byte message padding. You should use this option only if your server or client might be exchanging messages containing fields shorter than four bytes with a client or server that was built using NeXT Software Release 1.0. If **P**, use four-byte message padding. The default value is **P**. For example, a one-byte message element would be padded to two bytes if you specify **-p**, or four bytes by default.
[q,Q]	If **q**, suppress warning statements. If **Q**, print warning statements. The default value is **Q**.
[r,R]	If **r**, use **msg_rpc**(); if **R**, use **msg_send(), msg_receive**() pairs. The default value is **r**.

[s,S] If **s**, generate symbol table with *sys***Server.c** code. The layout of a symbol table (**mig_symtab_t**) is defined in the header file **mig_error.h**. If **S**, suppress the symbol table. The default value is **S**. This is useful for protection systems where access to the server's operations is dynamically specifiable or for providing a run-time indirected server call interface with **syscall()** (server-to-server calls made on behalf on a client).

[v,V] If **v** (verbose), print routines and types as they're processed. If **V**, compile silently. The default value is **V**.

Any switches MiG doesn't recognize are passed to the C preprocessor. MiG also notices if the **-MD** option is being passed to the C preprocessor. If it is, MiG fixes up the resulting ".d" file to show the dependencies of the ".h," *sys***User.c**, and *sys***Server.c** files on the ".defs" file and any included ".defs" files. For this feature to work correctly the name of the subsystem must be the same as the name of the ".defs" file.

MiG runs the C preprocessor to process comments and preprocessor macros such as **#include** or **#define**. For example, the following statement can be used to include the type definitions for standard Mach and C types:

```
#include <std_types.defs>
```

The output from the C preprocessor is then passed to the program **migcom**, which generates the C files.

Chapter 3
Using Loadable Kernel Servers

This chapter discusses how to use the kernel-server loader functions to interact with *loadable kernel servers*. Loadable kernel servers are modules, such as device drivers and network protocols, that can be added to the NeXT Mach kernel. One example of interaction with a loadable kernel server is using the function **kern_loader_load_server**() to load a loadable kernel server. Another example is using the function **kern_loader_server_list**() to get a list of all kernel servers that are either loaded or prepared for loading (allocated).

The following section gives some more information on loadable kernel servers and on the kernel-server loader, **kern_loader**. After that are some examples of using the kernel-server loader functions. Each kernel-server loader function is described in detail in Chapter 4, "C Functions."

For more information about loadable kernel servers, see the manual *Writing Loadable Kernel Servers*. That manual also has information about the kernel-server utility, **kl_util**, which is a command-line interface to many of the functions described in this chapter.

Loadable Kernel Server Concepts

The kernel-server loader is a server task that's automatically called during system startup. When started, it reads a list of loadable kernel servers out of its configuration file, **/etc/kern_loader.conf**, and allocates these servers.

A loadable kernel server is a module that's loaded into the kernel after the system has been booted. Because loadable kernel servers are the only way to add kernel functionality without recompiling the whole kernel, they're the only way for anyone outside of NeXT to write kernel-level device drivers and network protocols. However, third parties aren't the only ones to use loadable kernel servers—NeXT uses them for drivers of devices that many people won't have.

For example, the graphics tablet driver is a loadable kernel server that is loaded by the application **/NextAdmin/InstallTablet**. Having the tablet driver be loadable is advantageous because performance on NeXT computers that don't have a graphics tablet (which is the majority of NeXT computers) is better than if the tablet driver were always in the kernel.

Loadable kernel servers can have three states:

- Allocated. The kernel-server loader (**kern_loader**) has allocated space and resources for the loadable kernel server and is listening for Mach messages to its ports. However, the server isn't currently loaded into the kernel.
- Loaded. The loadable kernel server is running.
- Unallocated. The kernel-server loader has no space or other resources allocated for the loadable kernel server.

Not all loadable kernel servers stay in the allocated state when they're initialized. Servers that don't use Mach messages, for example, are loaded immediately. Most message-based servers, however, stay in the allocated state until the kernel-server loader receives either a message for the server or a request such as **kern_loader_load_server**() that tells it to load the server.

Each loadable kernel server stays loaded until the kernel-server loader either shuts down or receives a request to unload or delete the server (such as **kern_loader_unload_server**()).

See the *Writing Loadable Kernel Servers* manual for more information on loadable kernel servers and on using **kern_loader**.

Overview of Kernel-Server Loader Functions

This section describes the usage of the kernel-server loader functions. See the "Kernel-Server Loader Functions" section of Chapter 4 for more information on each of the functions.

The kernel-server loader functions are broken into two groups—those that deal with a single loadable kernel server, and those that deal with the kernel-server loader itself. There are also two functions to help you print error messages, **kern_loader_error**() and **kern_loader_error_string**().

Before you can call any other kernel-server loader function, you must call **kern_loader_look_up**() to obtain the port of the kernel-server loader. You must provide this port as a parameter to all of the following function calls. A similar parameter, returned by **kern_loader_server_com_port**(), is required only for calls to functions that deal with a server's message logging.

Use **kern_loader_add_server**() to cause a loadable kernel server to be allocated. If the server starts automatically, then it will be added to the kernel; otherwise, you can call **kern_loader_load_server**() to load the server into the kernel. To remove a loadable kernel server from the kernel, use **kern_loader_unload_server**() (to leave the server in the allocated state) or **kern_loader_delete_server**() (to deallocate kernel-server loader resources for the server).

For each loadable kernel server, logging is off by default. To get log messages from a particular loadable kernel server, use **kern_loader_log_level**() to turn the server's logging on and **kern_loader_get_log**() to get the next log message. You might want to turn logging off (by again using **kern_loader_log_level**()) before you stop collecting log messages, since messages continue to be logged and take system space even when no one requests them.

You can get detailed information about the state of a particular server by calling **kern_loader_server_info**().

Use **kern_loader_status_port**() to register a port to which log messages from the kernel-server loader should be sent. These messages usually reflect changes in the state of one or more kernel servers. You can get a list of all the servers that the kernel-server loader knows about by calling **kern_loader_server_list**(). Use **kern_loader_abort**() to shut down or reconfigure the kernel-server loader. Use **kern_loader_ping**() to make sure either that the kernel-server loader is responding normally to messages or that all outstanding status messages have been sent.

Functions for Asynchronous Messages

Three of the kernel-server loader functions don't immediately return information. Instead, these three functions tell the kernel-server loader to send asynchronous reply messages that contain the information. Whenever you call one of these functions, you must supply the code necessary to handle the kernel-server loader's reply message. The following table shows the three asynchronous kernel-server loader functions and their corresponding user-written functions.

Asynchronous Function	**User-Written Function**
kern_loader_ping()	*ping_func()*
kern_loader_get_log()	*log_data_func()*
kern_loader_status_port()	*string_func()*

This section describes how to handle asynchronous reply messages from the kernel-server loader. First it describes the code that all three of the asynchronous functions require in your program. Then it describes how to implement the handler necessary for each of the functions.

Common Code for Handling Reply Messages

If your program calls a kernel-server loader function that sends an asynchronous reply message, then your program must follow these steps to handle reply messages:

1. Allocate a port on which to receive messages from the kernel-server loader.

2. Call the asynchronous function, passing as data the receiving port.

3. Listen to the receiving port (often in a separate thread).

4. After receiving a message on the port, call **kern_loader_reply_handler()**.

5. Take care of the reply message in a handling function, which is called by **kern_loader_reply_handler()**.

You must write the handling function that's called by **kern_loader_reply_handler()**. You must also create a structure that specifies which handling functions exist; you pass a pointer to this structure to **kern_loader_reply_handler()** every time you call it. The structure is of type **kern_loader_reply_t**, which is defined in the header file **kernserv/kern_loader_reply_handler.h** as the following:

```
typedef struct kern_loader_reply {
    void          *arg;          /* argument to pass to function */
    msg_timeout_t timeout;       /* timeout for RPC return msg_send */
    kern_return_t (*string)(    /* kern_loader_status_port() function */
                void          *arg,
                printf_data_t string,
                unsigned int  string_count,
                int           level);
    kern_return_t (*ping)(      /* kern_loader_ping() function */
                void       *arg,
                int         id);
    kern_return_t (*log_data)(/* kern_loader_get_log() function */
                void          *arg,
                printf_data_t log_data,
                unsigned int  log_data_count);
} kern_loader_reply_t;
```

The following example calls one of the asynchronous **kern_loader** functions, **kern_loader_status_port()**. The handler for the reply message is called **print_string()**, and is specified to the kernel-server loader using the structure **reply_handlers**.

```
#import <mach.h>
#import <kernserv/kern_loader_types.h>
#import <kernserv/kern_loader.h>
#import <kernserv/kern_loader_reply_handler.h>
#import <cthreads.h>

void receive_thread(port_name_t port);
kern_return_t print_string(void *arg, printf_data_t string,
    u_int string_count, int level);
```

```
main()
{
    kern_return_t   r;
    port_name_t     status_port, kl_port;

    r = kern_loader_look_up(&kl_port);
    if (r != KERN_SUCCESS) {
        mach_error("kl_util: can't find kernel loader", r);
        exit(1);
    }

    r = port_allocate(task_self(), &status_port);
    if (r != KERN_SUCCESS) {
        mach_error("kl_util: can't allocate reply port", r);
        exit(1);
    }

    /* Get generic status messages on this port. */
    r = kern_loader_status_port(kl_port, status_port);
    if (r != KERN_SUCCESS) {
        kern_loader_error("Couldn't specify status port", r);
            exit(1);
    }

    /* Create a thread to listen on status_port. */
    cthread_detach(cthread_fork((cthread_fn_t)receive_thread,
        (any_t)status_port));

    /*
     * Sleep for a while so we can enter kl_util commands at a shell
     * window.  The output of all commands (except status lines from
     * kl_util -s) will show up in both the window that's running this
     * program and in the window that's running kl_util.  (kl_util
     * also has a status port registered.)
     */
    sleep(30);
    exit(0);
}

kern_loader_reply_t reply_handlers = {
    0,                /* argument to pass to function */
    0,                /* timeout for rpc return msg_send */
    print_string,     /* string function */
    0,                /* reply_ping function */
    0                 /* log_data function */
};
```

```
void receive_thread(port_name_t port)
{
    char          msg_buf[KERN_LOADER_REPLY_INMSG_SIZE];
    msg_header_t *msg = (msg_header_t *)msg_buf;
    kern_return_t r;

    /* message handling loop */
    while (TRUE) {
        /* Receive the next message in the queue. */
        msg->msg_size = KERN_LOADER_REPLY_INMSG_SIZE;
        msg->msg_local_port = port;
        r = msg_receive(msg, MSG_OPTION_NONE, 0);
        if (r != KERN_SUCCESS)
            break;

        /* Handle the message we just received. */
        kern_loader_reply_handler(msg, &reply_handlers);
    }

    /* We get here only if msg_receive returned an error. */
    mach_error("receive_thread", r);
    exit(1);
}

/*
 * This function is called by kern_loader every time it has status to
 * report.
 */
kern_return_t print_string(void *arg, printf_data_t string,
    u_int string_count, int level)
{
    /* If the string is empty, return. */
    if (string_count == 0 || !string)
        return KERN_SUCCESS;

    /* Print the string we were passed, with our special prefix. */
    printf("print_string: %s", string);

    return KERN_SUCCESS;
}
```

Handling a Status Message

You can receive many reply messages as the result of just one call to **kern_loader_status_port()**. The function you must use to handle these reply messages is defined as follows:

kern_return_t *string_func*(void **arg*, printf_data_t *string*, u_int *string_count*, int *level*)

The first argument, *arg*, has the same value as the **arg** field in the **kern_loader_reply_t** structure. The string that the kernel-server loader is logging is returned in *string*, with the string's length returned in *string_count*. *level* is set to the priority of the log message, using the priorities defined in the header file **sys/syslog.h** (LOG_EMERG, LOG_ALERT, and so on).

Your function should return KERN_SUCCESS.

The following code is an example of a *string_func* named **print_string**().

```
/*
 * This function is called by kern_loader every time it has status to
 * report.
 */
kern_return_t print_string(void *arg, printf_data_t string,
    u_int string_count, int level)
{
    /* If the string is empty, return. */
    if (string_count == 0 || !string)
        return KERN_SUCCESS;

    /* Print the string we were passed, with our special prefix. */
    printf("print_string: %s", string);

    return KERN_SUCCESS;
}
```

Handling a Synchronization Message

A call to **kern_loader_ping**() results in a single reply message. Your handler for this reply message must have the following syntax:

kern_return_t *ping_func*(void **arg*, int *id*)

The first parameter, *arg*, is the value in the **arg** field of the **kern_loader_reply_t** structure. *id* is the same as the *id* value specified in the call to **kern_loader_ping**(). Your *ping_func* should return KERN_SUCCESS.

Here's an example of a *ping_func* that causes its task to shut down.

```
/* This function is called after a kern_loader_ping(). */
kern_return_t ping (void *arg, int id)
{
    exit(0);    /* Kill this process. */
}
```

Handling a Log Message

Each time you call **kern_loader_log_data()**, you receive a single reply message as soon as any log data from the specified driver is available. The function you write to handle this message must have the following syntax:

kern_return_t *log_func*(void **arg*, printf_data_t *log_data*,
    unsigned int *log_data_count*)

The first parameter has the same value as the *arg* field in the **kern_loader_reply_t** structure. The *log_data* parameter is a string containing the log entry from the loadable kernel server, preceded by a time stamp that indicates the relative time when the kernel-server loader received the log message. *log_data_count* is the size of *log_data* in bytes. You should call **vm_deallocate**() on *log_data* when it's no longer needed.

Your *log_func* should return KERN_SUCCESS.

Here's an example of a *log_func* called **log_data**. It prints out the log message it's passed, and then requests another log message.

```
kern_return_t log_data(void *arg, printf_data_t log_data,
    unsigned int log_data_count)
{
    kern_return_t r;

    /* Print the string we were passed, with our prefix. */
    printf("log_data: %s\n", log_data);
    vm_deallocate(task_self(), (vm_address_t)log_data,
        log_data_count*sizeof(*log_data));

    /* Request the next log message. */
    r = kern_loader_get_log(kl_port, server_com_port, reply_port);
    if (r != KERN_SUCCESS) {
        kern_loader_error("log_data:  Error calling
            kern_loader_get_log", r);
        exit(1);
    }

    return KERN_SUCCESS;
}
```

Chapter 4
C Functions

This chapter gives detailed descriptions of the C functions provided by the NeXT Mach operating system. It also describes some macros that behave like functions. For this chapter, the functions and macros are divided into five groups:

- C thread functions. Use these to implement multiple threads in an application.
- Mach kernel functions. Use these to get access to the Mach operating system.
- Bootstrap Server functions. Use these to set up communication between the task that provides a local service and the tasks that use the service.
- Network Name Server functions. Use these to set up communication between tasks that might not be on the same machine.
- Kernel-server loader functions. Use these to load and unload loadable kernel servers, to add and delete servers to and from the kernel-server loader, and to get information about servers.

Within each section, functions are subgrouped with other functions that perform related tasks. These subgroups are described in alphabetical order by the name of the first function listed in the subgroup. Functions within subgroups are also listed alphabetically, with a pointer to the subgroup's description.

For convenience, these functions are summarized in the *NeXT Technical Summaries* manual. The summary lists functions by the same subgroups used in this chapter and combines several related subgroups under a heading such as "Basic C Thread Functions" or "Task Functions." For each function, the summary shows the calling sequence.

C Thread Functions

These functions provide a C language interface to the low-level, language-independent primitives for manipulating threads of control.

In a multithreaded application, you should use the C thread functions whenever possible, rather than Mach kernel functions. If you need to call a Mach kernel function that requires a **thread_t** argument, you can find a C thread's corresponding Mach thread by calling **cthread_thread**().

condition_alloc(), mutex_alloc()

SUMMARY Create a condition or mutex object

SYNOPSIS

#include <cthreads.h>

condition_t **condition_alloc()**
mutex_t **mutex_alloc()**

DESCRIPTION

The macros **condition_alloc()** and **mutex_alloc()** provide dynamic allocation of condition and mutex objects. When you're finished using these objects, you can deallocate them using **condition_free**() and **mutex_free**().

EXAMPLE

```
my_condition = condition_alloc();
my_mutex = mutex_alloc();
```

SEE ALSO

condition_init(), **mutex_init**(), **condition_free**(), **mutex_free**()

condition_broadcast()

SUMMARY Broadcast a condition

SYNOPSIS

#include <cthreads.h>

void **condition_broadcast**(condition_t *c*)

DESCRIPTION

The macro **condition_broadcast**() wakes up all threads that are waiting (via **condition_wait**()) for the condition *c*. This macro is similar to **condition_signal**(), except that **condition_signal**() doesn't wake up every waiting thread.

EXAMPLE

```
any_t listen(any_t arg)
{
    mutex_lock(my_mutex);
    while(!data)
        condition_wait(my_condition, my_mutex);
    /* . . . */
    mutex_unlock(my_mutex);

    mutex_lock(printing);
    printf("Condition has been met\n");
    mutex_unlock(printing);
}

main()
{
    my_condition = condition_alloc();
    my_mutex = mutex_alloc();
    printing = mutex_alloc();

    cthread_detach(cthread_fork((cthread_fn_t)listen, (any_t)0));

    mutex_lock(my_mutex);
    data = 1;
    mutex_unlock(my_mutex);
    condition_broadcast(my_condition);
    /* . . . */
}
```

SEE ALSO

condition_signal(), **condition_wait**()

condition_clear(), mutex_clear()

SUMMARY Clear a condition or mutex object

SYNOPSIS

#include <cthreads.h>

void **condition_clear**(struct condition **c*)
void **mutex_clear**(struct mutex **m*)

DESCRIPTION

You must call one of these macros before freeing an object of type **struct condition** or **struct mutex**. For example, **mutex_free()** could be written in terms of **mutex_clear()** as follows:

```
void mutex_free(m)
    mutex_t m;
{
    mutex_clear(m);
    free((char *) m);
}
```

SEE ALSO

condition_init(), **mutex_init()**, **condition_free()**, **mutex_free()**

condition_free(), mutex_free()

SUMMARY Deallocate a condition or mutex object

SYNOPSIS

#include <cthreads.h>

void **condition_free**(condition_t *c*)
void **mutex_free**(mutex_t *m*)

DESCRIPTION

The macros **condition_free()** and **mutex_free()** let you deallocate condition and mutex objects that were allocated dynamically. Before deallocating such an object, you must guarantee that no other thread will reference it. In particular, a thread blocked in **mutex_lock()** or **condition_wait()** should be viewed as referencing the object continually, so freeing the object out from under such a thread is erroneous, and can result in bugs that are extremely difficult to track down.

SEE ALSO

condition_alloc(), **mutex_alloc()**, **condition_clear()**, **mutex_clear()**

condition_init(), mutex_init()

SUMMARY Initialize a condition variable or mutex

SYNOPSIS

#include <cthreads.h>

void **condition_init**(struct condition **c*)
void **mutex_init**(struct mutex **m*)

DESCRIPTION

The macros **condition_init()** and **mutex_init()** initialize an object of the **struct condition** or **struct mutex** referent type, so that its address can be used wherever an object of type **condition_t** or **mutex_t** is expected. For example, **mutex_alloc()** could be written in terms of **mutex_init()** as follows:

```
mutex_t
mutex_alloc()
{
    register mutex_t m;
    m = (mutex_t) malloc(sizeof(struct mutex));
    mutex_init(m);
    return m;
}
```

Initialization of the referent type is most often used when you have included the referent type itself (rather than a pointer) in a larger structure, for more efficient storage allocation.

For instance, a data structure might contain a component of type **struct mutex** to allow each instance of that structure to be locked independently. During initialization of the instance, you would call **mutex_init()** on the **struct mutex** component. The alternative of using a **mutex_t** component and initializing it using **mutex_alloc()** would be less efficient.

SEE ALSO

condition_alloc(), mutex_alloc(), condition_clear(), mutex_clear()

condition_name(), condition_set_name(), mutex_name(), mutex_set_name()

SUMMARY Associate a string with a condition or mutex variable

SYNOPSIS

#include <cthreads.h>

char ***condition_name**(condition_t *c*)
void **condition_set_name**(condition_t *c*, char **name*)
char ***mutex_name**(mutex_t *m*)
void **mutex_set_name**(mutex_t *m*, char **name*)

DESCRIPTION

These macros let you associate a name with a condition or a mutex object. The name is used when trace information is displayed. You can also use this name for your own application-dependent purposes.

EXAMPLE

```
/* Do something if this is a "TYPE 1" condition. */
if (strcmp(condition_name(c), "TYPE 1") == 0)
    /* Do something. */;
```

condition_set_name() → See condition_name()

condition_signal()

SUMMARY Signal a condition

SYNOPSIS

#include <cthreads.h>

void **condition_signal**(condition_t *c*)

DESCRIPTION

The macro **condition_signal()** should be called when one thread needs to indicate that the condition represented by the condition variable is now true. If any other threads are waiting (via **condition_wait()**), at least one of them will be awakened. If no threads are waiting, nothing happens. The macro **condition_broadcast()** is similar to this one, except that it wakes up *all* threads that are waiting.

EXAMPLE

```
any_t listen(any_t arg)
{
    mutex_lock(my_mutex);
    while(!data)
        condition_wait(my_condition, my_mutex);
    /* . . . */
    mutex_unlock(my_mutex);

    mutex_lock(printing);
    printf("Condition has been met\n");
    mutex_unlock(printing);
}

main()
{
    my_condition = condition_alloc();
    my_mutex = mutex_alloc();
    printing = mutex_alloc();

    cthread_detach(cthread_fork((cthread_fn_t)listen, (any_t)0));

    mutex_lock(my_mutex);
    data = 1;
    mutex_unlock(my_mutex);
    condition_signal(my_condition);
    /* . . . */
}
```

SEE ALSO

condition_broadcast(), **condition_wait()**

condition_wait()

SUMMARY Wait on a condition

SYNOPSIS

#include <cthreads.h>

void **condition_wait**(condition_t *c*, mutex_t *m*)

DESCRIPTION

The function **condition_wait**() unlocks the mutex it takes as a parameter, suspends the calling thread until the specified condition is likely to be true, and locks the mutex again when the thread resumes. There's no guarantee that the condition will be true when the thread resumes, so this function should always be used as follows:

```
mutex_t m;
condition_t c;

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

SEE ALSO

condition_broadcast(), **condition_signal**()

cthread_abort()

SUMMARY Interrupt a C thread

SYNOPSIS

#include <cthreads.h>

kern_return_t **cthread_abort**(cthread_t *t*)

DESCRIPTION

This function provides the functionality of **thread_abort()** to C threads. **cthread_abort**() interrupts system calls; it's usually used along with **thread_suspend**(), which stops a thread from executing any more user code. Calling **cthread_abort**() on a thread that's not suspended is risky, since it's difficult to know exactly what system trap, if any, the thread might be executing and whether an interrupt return would cause the thread to do something useful.

See **thread_abort**() for a full description of the use of this function.

cthread_count()

SUMMARY Get the number of threads in this task

SYNOPSIS

#include <cthreads.h>

int **cthread_count**()

DESCRIPTION

This function returns the number of threads that exist in the current task. You can use this function to help make sure that your task doesn't create too many threads (over two hundred or so). See **cthread_set_limit()** for information on restricting the number of threads in a task.

EXAMPLE

```
printf("C thread count should be 1, is %d\n", cthread_count());
cthread_detach(cthread_fork((cthread_fn_t)listen, (any_t)0));
printf("C thread count should be 2, is %d\n", cthread_count());
```

SEE ALSO

cthread_limit(), **cthread_set_limit**()

cthread_data(), cthread_set_data()

SUMMARY Associate data with a thread

SYNOPSIS

#include <cthreads.h>

any_t **cthread_data**(cthread_t *t*)
void **cthread_set_data**(cthread_t *t*, any_t *data*)

DESCRIPTION

The macros **cthread_data**() and **cthread_set_data**() let you associate arbitrary data with a thread, providing a simple form of thread-specific "global" variable. More elaborate mechanisms, such as per-thread property lists or hash tables, can then be built with these macros.

EXAMPLE

```
int listen(any_t arg)
{
    mutex_lock(printing);
    printf("This thread's data is: %d\n",
        (int)cthread_data(cthread_self()));
    mutex_unlock(printing);
    /* . . . */
}

main()
{
    cthread_t lthread;

    printing = mutex_alloc();

    lthread = cthread_fork((cthread_fn_t)listen, (any_t)0);
    cthread_set_data(lthread, (any_t)100);
    cthread_detach(lthread);
    /* . . . */
}
```

SEE ALSO

cthread_name(), cthread_set_name()

cthread_detach()

SUMMARY Detach a thread

SYNOPSIS

#include <cthreads.h>

void **cthread_detach**(cthread_t *t*)

DESCRIPTION

The function **cthread_detach()** is used to indicate that the given thread will never be joined. This is usually known at the time the thread is forked, so the most efficient usage is the following:

cthread_detach(cthread_fork(*function*, *argument***));**

A thread may, however, be detached at any time after it's forked, as long as no other attempt is made to join it or detach it.

EXAMPLE

```
cthread_detach(cthread_fork((cthread_fn_t)listen, (any_t)reply_port));
```

SEE ALSO

cthread_fork(), **cthread_join()**

cthread_errno()

SUMMARY Get a thread's errno value

SYNOPSIS

#include <cthreads.h>

int **cthread_errno()**

DESCRIPTION

Use the **cthread_errno()** function to get the errno value for the current thread. In the UNIX operating system, **errno** is a process-wide global variable that's set to an error number when a UNIX system call fails. However, because Mach has multiple threads per process, Mach keeps errno information on a per-thread basis as well as in **errno**.

Like the value of **errno**, the value returned by **cthread_errno()** is valid only if the last UNIX system call returned -1. Errno values are defined in the header file **sys/errno.h**.

EXAMPLE

```
int ret;

ret = chown(FILEPATH, newOwner, newGroup);
if (ret == -1) {
    if (cthread_errno() == ENAMETOOLONG)
        /* . . . */
}
```

SEE ALSO

cthread_set_errno(), **intro(2)** UNIX manual page

cthread_exit()

SUMMARY Exit a thread

SYNOPSIS

#include <cthreads.h>

void **cthread_exit**(any_t *result*)

DESCRIPTION

The function **cthread_exit()** terminates the calling thread. The result is passed to the thread that joins the caller, or is discarded if the caller is detached.

An implicit **cthread_exit()** occurs when the top-level function of a thread returns, but it may also be called explicitly.

EXAMPLE

```
cthread_exit(0);
```

SEE ALSO

cthread_detach(), **cthread_fork()**, **cthread_join()**

cthread_fork()

SUMMARY Fork a thread

SYNOPSIS

#include <cthreads.h>

cthread_t **cthread_fork**(any_t (**function*)(), any_t *arg*)

DESCRIPTION

The function **cthread_fork()** takes two parameters: a function for the new thread to execute, and a parameter to this function. **cthread_fork()** creates a new thread of control in which the specified function is executed concurrently with the caller's thread. This is the sole means of creating new threads.

The **any_t** type represents a pointer to any C type. The **cthread_t** type is an integer-size handle that uniquely identifies a thread of control. Values of type **cthread_t** will be referred to as thread identifiers. Arguments larger than a pointer must be passed by reference. Similarly, multiple arguments must be simulated by passing a pointer to a structure containing several components. The call to **cthread_fork()** returns a thread identifier that can be passed to **cthread_join()** or **cthread_detach()** (see the following example). Every thread must be either joined or detached exactly once.

EXAMPLE

```
cthread_detach(cthread_fork((cthread_fn_t)listen, (any_t)reply_port));
```

SEE ALSO

cthread_detach(), **cthread_exit()**, **cthread_join()**

cthread_join()

SUMMARY Join threads

SYNOPSIS

#include <cthreads.h>

any_t **cthread_join**(cthread_t *t*)

DESCRIPTION

The function **cthread_join**() suspends the caller until the specified thread *t* terminates via **cthread_exit**(). The caller receives either the result of *t*'s top-level function or the argument with which *t* explicitly called **cthread_exit**().

Attempting to join one's own thread results in deadlock.

EXAMPLE

```
cthread_join(cthread_fork((any_t (*)())listen, (any_t)reply_port));
```

SEE ALSO

cthread_detach(), **cthread_exit**(), **cthread_fork**()

cthread_limit(), cthread_set_limit()

SUMMARY Get or set the maximum number of threads in this task

SYNOPSIS

#include <cthreads.h>

int **cthread_limit**()
void **cthread_set_limit**(int *limit*)

ARGUMENTS

limit: The new maximum number of C threads per task. Specify zero if you want no limit.

DESCRIPTION

These functions can help you to avoid creating too many threads. The danger in creating a large number of threads is that the kernel might run out of resources and panic. Usually, a task should avoid creating more than about two hundred threads.

Use **cthread_set_limit()** to set a limit on the number of threads in the current task. When the limit is reached, new C threads will appear to fork successfully, but they will have no associated Mach thread so they won't do anything.

Use **cthread_limit()** to find out how many threads can exist in the current task. If the returned value is zero (the default), then no limit is currently being enforced.

Important: Use **cthread_count()** to determine when your task is approaching the maximum number of threads.

EXAMPLE

```
cthread_set_limit(LIMIT);

/* . . . */

/* Fork if we haven't reached the limit. */
if ( (LIMIT == 0) || (LIMIT > cthread_count()) )
    cthread_detach(cthread_fork((any_t (*)())a_thread,(any_t)0));
```

cthread_name(), cthread_set_name()

SUMMARY Associate a string with a thread

SYNOPSIS

#include <cthreads.h>

char ***cthread_name**(cthread_t *t*)
void **cthread_set_name**(cthread_t *t*, char **name*)

DESCRIPTION

The functions **cthread_name()** and **cthread_set_name()** let you associate an arbitrary name with a thread. The name is used when trace information is displayed. The name may also be used for application-specific diagnostics.

EXAMPLE

```
int listen(any_t arg)
{
    mutex_lock(printing);
    printf("This thread's name is: %s\n",
        cthread_name(cthread_self()));
    mutex_unlock(printing);
    /* . . . */
}
```

```
main()
{
    cthread_t lthread;

    printing = mutex_alloc();

    lthread = cthread_fork((cthread_fn_t)listen, (any_t)0);
    cthread_set_name(lthread, "lthread");
    cthread_detach(lthread);
    /* . . . */
}
```

SEE ALSO

cthread_data(), **cthread_set_data**()

cthread_priority(), cthread_max_priority()

SUMMARY Set the scheduling priority for a C thread

SYNOPSIS

#include <cthreads.h>

kern_return_t **cthread_priority**(cthread_t *t*, int *priority*, boolean_t *set_max*)
kern_return_t **cthread_max_priority**(cthread_t *t*, processor_set_t *processor_set*,
  int *max_priority*)

ARGUMENTS

t: The C thread whose priority is to be changed.

priority: The new priority to change it to.

set_max: Also set *thread*'s maximum priority if true.

processor_set: The privileged port for the processor set to which *thread* is currently assigned.

max_priority: The new maximum priority.

DESCRIPTION

These routines give C threads the functionality of **thread_priority**() and **thread_max_priority**(). See those functions for more details than are provided here.

cthread_priority() changes the base priority and (optionally) the maximum priority of *thread*. If the new base priority is higher than the scheduled priority of the currently executing thread, this thread might be preempted. The maximum priority of the thread is also set if *set_max* is true. This call fails if *priority* is greater than the current maximum priority of the thread. As a result, **cthread_priority**() can never raise—only lower—the value of a thread's maximum priority.

cthread_max_priority() changes the maximum priority of the thread. Because it requires the privileged port for the processor set, this call can reset the maximum priority to any legal value. If the new maximum priority is less than the thread's base priority, then the thread's base priority is set to the new maximum priority.

EXAMPLE

```
/* Get the privileged port for the default processor set. */
error=processor_set_default(host_self(), &default_set);
if (error!=KERN_SUCCESS) {
    mach_error("Error calling processor_set_default()", error);
    exit(1);
}

error=host_processor_set_priv(host_priv_self(), default_set,
    &default_set_priv);
if (error!=KERN_SUCCESS) {
    mach_error("Call to host_processor_set_priv() failed", error);
    exit(1);
}

/* Set the max priority. */
error=cthread_max_priority(cthread_self(), default_set_priv,
    priority);
if (error!=KERN_SUCCESS)
    mach_error("Call to cthread_max_priority() failed",error);

/* Set the thread's priority. */
error=cthread_priority(cthread_self(), priority, FALSE);
if (error!=KERN_SUCCESS)
    mach_error("Call to cthread_priority() failed",error);
```

RETURN

KERN_SUCCESS: Operation completed successfully

KERN_INVALID_ARGUMENT: *cthread* is not a C thread, *processor_set* is not a privileged port for a processor set, or *priority* is out of range (not in 0-31).

KERN_FAILURE: The requested operation would violate the thread's maximum priority (only for **cthread_priority**()) or the thread is not assigned to the processor set whose privileged port was presented.

SEE ALSO

thread_priority(), **thread_max_priority**(), **thread_policy**(), **task_priority**(), **processor_set_priority**()

cthread_self()

SUMMARY Return the caller's thread identifier

SYNOPSIS

#include <cthreads.h>

cthread_t **cthread_self()**

DESCRIPTION

The function **cthread_self**() returns the caller's own thread identifier, which is the same value that was returned by **cthread_fork**() to the creator of the thread. The thread identifier uniquely identifies the thread, and hence may be used as a key in data structures that associate user data with individual threads. Since thread identifiers may be reused by the underlying implementation, you should be careful to clean up such associations when threads exit.

EXAMPLE

```
printf("This thread's name is: %s\n",
    cthread_name(cthread_self()));
mutex_unlock(printing);
```

SEE ALSO

cthread_fork(), **cthread_thread**(), **thread_self**()

cthread_set_data() → See cthread_data()

cthread_set_errno_self()

SUMMARY Set the current thread's errno value

SYNOPSIS

#include <cthreads.h>

void **cthread_set_errno_self**(int *error*)

ARGUMENTS

error: The value to set the errno to. Errno values are defined in the header file **sys/errno.h**.

DESCRIPTION

Use this function to set the errno value for the current thread. In the UNIX operating system, **errno** is a process-wide global variable that's set to an error number when a UNIX system call fails. However, because Mach has multiple threads per process, Mach keeps errno information on a per-thread basis as well as in **errno**. This function has no effect on the value of **errno**.

The current thread's errno value can be obtained by calling **cthread_errno()**.

EXAMPLE

```
cthread_set_errno_self(EPERM);
```

SEE ALSO

cthread_errno(), **intro(2)** UNIX manual page

cthread_set_limit() → See cthread_limit()

cthread_set_name() → See cthread_name()

cthread_thread()

SUMMARY Return the caller's thread identifier

SYNOPSIS

#include <cthreads.h>

thread_t **cthread_thread**(cthread_t *t*)

DESCRIPTION

The macro **cthread_thread()** returns the Mach thread that corresponds to the specified C thread *t*.

EXAMPLE

```
/* Save the cthread and thread values for the forked thread. */
l_cthread = cthread_fork((cthread_fn_t)listen, (any_t)0);
cthread_detach(l_cthread);
l_realthread = cthread_thread(l_cthread);
```

SEE ALSO

cthread_fork(), **cthread_self()**

cthread_yield()

SUMMARY Yield the processor to other threads

SYNOPSIS

#include <cthreads.h>

void **cthread_yield**()

DESCRIPTION

The function **cthread_yield**() is a hint to the scheduler, suggesting that this would be a convenient point to schedule another thread to run on the current processor.

EXAMPLE

```
int i, n;

/* n is set previously */
for (i = 0; i < n; i += 1)
    cthread_yield();
```

SEE ALSO

cthread_priority(), **thread_switch**()

mutex_alloc() → See condition_alloc()

mutex_clear() → See condition_clear()

mutex_free() → See condition_free()

mutex_init() → See condition_init()

mutex_lock()

SUMMARY Lock a mutex variable

SYNOPSIS

#include <cthreads.h>

void **mutex_lock**(mutex_t *m*)

DESCRIPTION

The macro **mutex_lock()** attempts to lock the mutex *m* and blocks until it succeeds. If several threads attempt to lock the same mutex concurrently, one will succeed, and the others will block until *m* is unlocked. A deadlock occurs if a thread attempts to lock a mutex it has already locked.

EXAMPLE

```
/* Only one thread at a time should call printf. */
mutex_lock(printing);
printf("Condition has been met\n");
mutex_unlock(printing);
```

SEE ALSO

mutex_try_lock(), **mutex_unlock()**

mutex_name() → See condition_name()

mutex_set_name() → See condition_name()

mutex_try_lock()

SUMMARY Try to lock a mutex variable

SYNOPSIS

#include <cthreads.h>

int **mutex_try_lock**(mutex_t *m*)

DESCRIPTION

The function **mutex_try_lock()** attempts to lock the mutex *m*, like **mutex_lock()**, and returns true if it succeeds. If *m* is already locked, however, **mutex_try_lock()** immediately returns false rather than blocking. For example, a busy-waiting version of **mutex_lock()** could be written using **mutex_try_lock()**:

```
void mutex_lock(mutex_t m)
{
    for (;;)
        if (mutex_try_lock(m))
            return;
}
```

SEE ALSO

mutex_lock(), mutex_unlock()

mutex_unlock()

SUMMARY Unlock a mutex variable

SYNOPSIS

#include <cthreads.h>

void **mutex_unlock**(mutex_t *m*)

DESCRIPTION

The function **mutex_unlock()** unlocks *m*, giving other threads a chance to lock it.

EXAMPLE

```
/* Only one thread at a time should call printf. */
mutex_lock(printing);
printf("Condition has been met\n");
mutex_unlock(printing);
```

SEE ALSO

mutex_lock(), mutex_try_lock()

exc_server()

SUMMARY Dispatch a message received on an exception port

SYNOPSIS

#include <mach.h>
#include <sys/exception.h>

boolean_t **exc_server**(msg_header_t **in*, msg_header_t **out*)

ARGUMENTS

in: A message that was received on the exception port. This message structure should be at least 64 bytes long.

out: An empty message to be filled by **exc_server**() and then sent. This message buffer should be at least 32 bytes long.

DESCRIPTION

This function calls the appropriate exception handler. You should call this function after you've received a message on an exception port that you set up previously. Usually, this function is used along with a user-defined exception handler, which must have the following protocol:

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

To receive a message on an exception port, you must first create a new port and make it the task or thread exception port. (You can't use the default task exception port because you can't get receive rights for it.) Before calling **msg_receive**(), you must set the **local_port** field of the header to the appropriate exception port and the **msg_size** field to the size of the structure for the incoming message.

exc_server() returns true if it accepted the incoming message, false if it didn't recognize the message's type.

You should keep a global value that indicates whether your exception handler successfully handled the exception. If it couldn't, then you should forward the exception message to the old exception port.

EXAMPLE

```
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("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("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)) {
        /* 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;
        mach_NeXT_exception("Forwarding exception", exception,
            code, subcode);
        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_self();
    r = exception_raise(ports.exc_port, thread_reply(),
        ports.clear_port, task_self(), EXC_SOFTWARE, 0x20000, 6);

    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);
    }
}
```

SEE ALSO

exception_raise(), mach_NeXT_exception()

exception_raise()

SUMMARY Cause an exception to occur

SYNOPSIS

#include <mach.h>
#include <sys/exception.h>

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

ARGUMENTS

exception_port: The exception port of the affected thread. (If the thread doesn't have its own exception port, then this should be the task's exception port.)

clear_port: The port to which a reply message should be sent from the exception handler. If you don't care to see the reply, you can use **thread_reply**().

thread: The thread in which the exception condition occurred. If the exception isn't thread-specific, then specify THREAD_NULL.

task: The task in which the exception condition occurred.

exception: The type of exception that occurred; for example, EXC_SOFTWARE. Values for this variable are defined in the header file **sys/exception.h**.

code: The exception code. The meaning of this code depends on the value of *exception*.

subcode: The exception subcode. The meaning of this subcode depends on the values of *exception* and *code*.

DESCRIPTION

This function causes an exception message to be sent to *exception_port*, which results in a call to the exception handler. Usually this routine is used along with a user-defined exception handler. (See **exc_server()** and **mach_NeXT_exception()** for more information on user-defined exception handlers.)

You can obtain *exception_port* by calling **thread_get_exception_port()** or (if no thread exception port exists or the exception is a task-wide one) **task_get_exception_port**().

If you're defining your own type of exception, you must have *exception* equal to EXC_SOFTWARE and *code* equal to or greater than 0x20000.

EXAMPLE

```
/* Raise the exception. */
r = exception_raise(ports.exc_port, thread_reply(), thread_self(),
    task_self(), EXC_SOFTWARE, 0x20000, 6);
if (r != KERN_SUCCESS)
    mach_error("catch_exception_raise didn't handle exception", r);
else {
    /* Use mutex so only one thread at a time can call printf. */
    mutex_lock(printing);
    printf("Successfully called exception_raise\n");
    mutex_unlock(printing);
}
```

RETURN

KERN_SUCCESS: The call succeeded.

KERN_FAILURE: The exception handler didn't successfully deal with the exception.

KERN_INVALID_ARGUMENT: One of the arguments wasn't valid.

SEE ALSO

exc_server(), **mach_NeXT_exception()**, **task_get_exception_port()**, **thread_get_exception_port()**

host_info()

SUMMARY Get information about a host

SYNOPSIS

#include <mach.h>

kern_return_t **host_info**(host_t *host*, int *flavor*, host_info_t *host_info*, unsigned int **host_info_count*)

ARGUMENTS

host: The host for which information is to be obtained.

flavor: The type of statistics that are wanted. Currently HOST_BASIC_INFO, HOST_PROCESSOR_SLOTS, and HOST_SCHED_INFO are implemented.

host_info: Returns statistics about *host*.

host_info_count: The number of integers in the info structure; returns the number of integers that Mach tried to fill the info structure with. For HOST_BASIC_INFO, you should set *host_info_count* to HOST_BASIC_INFO_COUNT. For HOST_PROCESSOR_SLOTS, you should set it to the maximum number of CPUs (returned by HOST_BASIC_INFO). For HOST_SCHED_INFO, set it to HOST_SCHED_INFO_COUNT.

DESCRIPTION

Returns the selected information array for a host, as specified by *flavor*. *host_info* is an array of integers that's supplied by the caller and returned filled with specified information. *host_info_count* is supplied by the caller as the maximum number of integers in *host_info* (which can be larger than the space required for the information). On return, it contains the actual number of integers in *host_info*.

Warning: This replaces the old **host_info**() call. It isn't backwards compatible.

Basic information is defined by HOST_BASIC_INFO. Its size is defined by HOST_BASIC_INFO_COUNT. Possible values of the **cpu_type** and **cpu_subtype** fields are defined in the header file **sys/machine.h**, which is included in **mach.h**.

```
struct host_basic_info {
    int           max_cpus;      /* maximum possible cpus for
                                  * which kernel is configured */
    int           avail_cpus;    /* number of cpus now available */
    vm_size_t     memory_size;   /* size of memory in bytes */
    cpu_type_t    cpu_type;      /* cpu type */
    cpu_subtype_t cpu_subtype;   /* cpu subtype */
};
typedef struct host_basic_info *host_basic_info_t;
```

Processor slots of the active (available) processors are defined by HOST_PROCESSOR_SLOTS. The size of this information should be obtained from the **max_cpus** field of the structure returned by HOST_BASIC_INFO. HOST_PROCESSOR_SLOTS returns an array of integers, each of which is the slot number of a CPU.

Additional information of interest to schedulers is defined by HOST_SCHED_INFO. The size of this information is defined by HOST_SCHED_INFO_COUNT.

```
struct host_sched_info {
    int  min_timeout;  /* minimum timeout in milliseconds */
    int  min_quantum;  /* minimum quantum in milliseconds */
};

typedef struct host_sched_info *host_sched_info_t
```

EXAMPLE

An example of using HOST_BASIC_INFO:

```
kern_return_t           ret;
struct host_basic_info  basic_info;
unsigned int            count=HOST_BASIC_INFO_COUNT;

ret=host_info(host_self(), HOST_BASIC_INFO,
    (host_info_t)&basic_info, &count);
if (ret != KERN_SUCCESS)
    mach_error("host_info() call failed", ret);
else printf("This system has %d bytes of RAM.\n",
    basic_info.memory_size);
```

An example of using HOST_PROCESSOR_SLOTS (you also need to include the HOST_BASIC_INFO code above so you can get **max_cpus**):

```
host_info_t  slots;
unsigned int cpu_count, i;

cpu_count=basic_info.max_cpus;
slots=(host_info_t)malloc(cpu_count*sizeof(int));
ret=host_info(host_self(), HOST_PROCESSOR_SLOTS, slots,
    &cpu_count);
if (ret!=KERN_SUCCESS)
    mach_error("PROCESSOR host_info() call failed", ret);
else for (i=0; i<cpu_count; i++)
    printf("CPU %d is in slot %d.\n", i, *slots++);
```

An example of using HOST_SCHED_INFO:

```
kern_return_t           ret;
struct host_sched_info  sched_info;
unsigned int            sched_count=HOST_SCHED_INFO_COUNT;

ret=host_info(host_self(), HOST_SCHED_INFO,
    (host_info_t)&sched_info, &sched_count);
if (ret != KERN_SUCCESS)
    mach_error("SCHED host_info() call failed", ret);
else
    printf("The minimum quantum is %d milliseconds.\n",
        sched_info.min_quantum);
```

RETURN

KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *host* is not a host, *flavor* is not recognized, or (for HOST_PROCESSOR_SLOTS) **count* is less than **max_cpus**.

KERN_FAILURE: **count* is less than HOST_BASIC_INFO_COUNT (when *flavor* is HOST_BASIC_INFO) or HOST_SCHED_INFO_COUNT (for HOST_SCHED_INFO).

MIG_ARRAY_TOO_LARGE: Returned info array is too large for *host_info*. *host_info* is filled as much as possible. *host_info_count* is set to the number of elements that would be returned if there were enough room.

SEE ALSO

host_kernel_version(), **host_processors**(), **processor_info**()

host_kernel_version()

SUMMARY Get kernel version information

SYNOPSIS

#include <mach.h>

kern_return_t **host_kernel_version**(host_t *host*, kernel_version_t *version*)

ARGUMENTS

host: The host for which information is being requested.

version: Returns a character string describing the kernel version executing on *host*.

DESCRIPTION

host_kernel_version() returns the version string compiled into *host*'s kernel at the time it was built. If you don't use the **kernel_version_t** declaration, then you should allocate KERNEL_VERSION_MAX bytes for the version string.

EXAMPLE

```
kern_return_t       ret;
kernel_version_t    string;

ret=host_kernel_version(host_self(), string);
if (ret != KERN_SUCCESS)
    mach_error("host_kernel_version() call failed", ret);
else
    printf("Version string:  %s\n", string);
```

RETURN

KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *host* was not a host.

KERN_INVALID_ADDRESS: *version* points to inaccessible memory.

SEE ALSO

host_info(), host_processors(), processor_info()

host_priv_self() → See host_self()

host_processor_set_priv()

SUMMARY Get the privileged port of a processor set

SYNOPSIS

#include <mach.h>

kern_return_t **host_processor_set_priv**(host_priv_t *host_priv*,
processor_set_t *processor_set_name*, processor_set_t **processor_set*)

ARGUMENTS

host_priv: The privileged host port for the desired host.

processor_set_name: The name port of the processor set.

processor_set: Returns the privileged port of the processor set.

DESCRIPTION

host_processor_set_priv() returns send rights to the privileged port for the specified processor set. This port is used in calls that can affect other threads or tasks. For example, **processor_set_tasks**() requires the privileged port because it returns the port of every task on the system.

EXAMPLE

```
kern_return_t    error;
processor_set_t  processor_set;
processor_set_t  default_set;

error=processor_set_default(host_self(), &default_set);
if (error != KERN_SUCCESS)
    mach_error("Call to processor_set_default failed", error);

error=host_processor_set_priv(host_priv_self(), default_set,
    &processor_set);
if (error != KERN_SUCCESS)
    mach_error("Call to host_processor_set_priv failed; make sure
        you're superuser", error);
```

RETURN

KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *host_priv* was not a privileged host port, or *processor_set_name* didn't name a valid processor set.

host_processor_sets()

SUMMARY Get the name ports of all processor sets on a host

SYNOPSIS

#include <mach.h>

kern_return_t **host_processor_sets**(host_t *host*,
processor_set_name_array_t **processor_set_list*,
unsigned int **processor_set_count*)

ARGUMENTS

host: The host port for the desired host.

processor_set_list: Returns an array of processor sets currently existing on *host*; no particular ordering is guaranteed.

processor_set_ count: Returns the number of processor sets in the *processor_set_list*.

DESCRIPTION

host_processor_sets() gets send rights to the name port for each processor set currently assigned to *host*. **host_processor_set_priv**() can be used to obtain the privileged ports from these if desired. *processor_set_list* is an array that is created as a result of this call. You should call **vm_deallocate**() on this array when the data is no longer needed.

Note: In single-processor systems, you can get the same information by calling **processor_set_default**().

EXAMPLE

```
kern_return_t              ret;
processor_set_name_array_t  list;
unsigned int               count;

ret=host_processor_sets(host_self(), &list, &count);
if (ret!=KERN_SUCCESS)
    mach_error("error calling host_processor_sets", ret);
else {
    /* . . . */
    ret=vm_deallocate(task_self(), (vm_address_t)list,
        sizeof(list)*count);
    if (ret!=KERN_SUCCESS)
        mach_error("error calling vm_deallocate", ret);
}
```

RETURN

KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *host* is not a host.

SEE ALSO

host_processor_set_priv(), **processor_set_create()**, **processor_set_tasks()**, **processor_set_threads()**, **processor_set_default()**

host_processors()

SUMMARY Get the processor ports for a host

SYNOPSIS

#include <mach.h>

kern_return_t **host_processors**(host_priv_t *host_priv*,
processor_array_t **processor_list*, unsigned int **processor_count*)

ARGUMENTS

host_priv: Privileged host port for the desired host.

processor_list: Returns the processors existing on *host_priv*; no particular ordering is guaranteed.

processor_count: Returns the number of processors in *processor_list*.

DESCRIPTION

host_processors() gets send rights to the processor port for each processor existing on *host_priv*. *processor_list* is an array that is created as a result of this call. The caller may wish to call **vm_deallocate**() on this array when the data is no longer needed.

EXAMPLE

```
kern_return_t      error;
processor_array_t  list;
unsigned int       count;

error=host_processors(host_priv_self(), &list, &count);
if (error!=KERN_SUCCESS){
    mach_error("error calling host_processors", error);
    exit(1);
}
/* . . . */
vm_deallocate(task_self(), (vm_address_t)list, sizeof(list)*count);
if (error!=KERN_SUCCESS)
    mach_error("Trouble freeing list", error);
```

RETURN

KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *host_priv* is not a privileged host port.

SEE ALSO

processor_info(), **processor_start**(), **processor_exit**(), **processor_control**()

host_self(), host_priv_self()

SUMMARY Get the host port for this host

SYNOPSIS

#include <mach.h>

host_t **host_self()**
host_priv_t **host_priv_self()**

DESCRIPTION

host_self() returns send rights to the host port for the host on which the call is executed. This port can be used only to obtain information about the host, not to control the host.

host_priv_self() returns send rights to the privileged host port for the host on which the call is executed. This port is used to control physical resources on that host and is only available to privileged tasks. PORT_NULL is returned if the invoker is not the UNIX superuser.

EXAMPLE

```
/* Get the privileged port for the default processor set. */
error=processor_set_default(host_self(), &default_set);
if (error!=KERN_SUCCESS) {
    mach_error("Error calling processor_set_default()", error);
    exit(1);
}

error=host_processor_set_priv(host_priv_self(), default_set,
    &default_set_priv);
if (error!=KERN_SUCCESS) {
    mach_error("Call to host_processor_set_priv() failed", error);
    exit(1);
}
```

SEE ALSO

host_processors(), **host_info()**, **host_kernel_version()**

mach_error(), mach_error_string()

SUMMARY Display or get a Mach error string

SYNOPSIS

#include <mach.h>

void **mach_error**(char **string*, kern_return_t *errno*)
char ***mach_error_string**(kern_return_t *errno*)

ARGUMENTS

string: The string you want displayed before the Mach error string.

errno: The error value for which you want an error string.

DESCRIPTION

The function **mach_error()** displays a message on **stderr**. The message contains the string specified by *string*, the string returned by **mach_error_string()**, and the actual error value (*errno*). Since **mach_error()** isn't thread-safe, you might want to protect it with a mutex if you call it in a multiple-thread task.

The function **mach_error_string()** returns the string associated with *errno*.

Note that because the error value specified by *errno* is of type **kern_return_t**, these functions work only with Mach functions.

EXAMPLE

```
mutex_t         printing;

main()
{
    kern_return_t  error;
    port_t         result;

    printing = mutex_alloc();

    /* . . . */
    if ((error=port_allocate(task_self(), &result)) != KERN_SUCCESS) {
        mutex_lock(printing);
        mach_error("Error calling port_allocate", error);
        mutex_unlock(printing);
        exit(1);
    }
    /* . . . */
}
```

mach_NeXT_exception(), mach_NeXT_exception_string()

SUMMARY Display or get a Mach exception string

SYNOPSIS

#include <mach.h>

void **mach_NeXT_exception**(char **string*, int *exception*, int *code*, int *subcode*)
char ***mach_NeXT_exception_string**(int *exception*, int *code*, int *subcode*)

ARGUMENTS

string: The string you want displayed before the Mach exception string.

exception: The exception value for which you want a string.

code: The exception code. How this is used depends on the value of *exception.*

subcode: The exception subcode. How this is used depends on the value of *exception.*

DESCRIPTION

The function **mach_NeXT_exception()** displays a message on **stderr**. The message contains the string specified by *string*, then the string returned by **mach_NeXT_exception_string**(), and then the values of *exception*, *code*, and *subcode*. Since **mach_NeXT_exception()** isn't thread-safe, you might want to protect it with a mutex if you call it in a multiple-thread task.

The function **mach_NeXT_exception_string()** returns the string associated with *exception*, *code*, and *subcode.*

EXAMPLE

```
/*
 * 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)) {
        /* 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;
        mach_NeXT_exception("Forwarding exception", exception,
            code, subcode);
        return KERN_FAILURE;  /* Couldn't handle this exception. */
    }
}
```

SEE ALSO

exception_raise(), **exc_server()**

map_fd()

SUMMARY Map a file into virtual memory

SYNOPSIS

#include <mach.h>

kern_return_t **map_fd**(int *fd*, vm_offset_t *offset*, vm_offset_t **address*, boolean_t *find_space*, vm_size_t *size*)

ARGUMENTS

fd: An open UNIX file descriptor for the file that's to be mapped.

offset: The byte offset within the file, at which mapping is to begin.

address: A pointer to an address in the calling process at which the mapped file should start. This address, unlike the offset, must be page-aligned.

find_space: If true, the kernel will select an unused address range at which to map the file and return its value in *address*.

size: The number of bytes to be mapped.

DESCRIPTION

The function **map_fd()** is a UNIX extension that's technically not part of Mach. This function causes *size* bytes of data starting at *offset* in the file specified by *fd* to be mapped into the virtual memory at the address specified by *address*. If *find_space* is true, the input value of *address* can be null, and the kernel will find an unused piece of virtual memory to use. (You should free this space with **vm_deallocate()** when you no longer need it.) If you provide a value for *address*, it must be page-aligned and at least *size* bytes long. The sum of *offset* and *size* must not exceed the length of the file.

Memory mapping doesn't cause I/O to take place. When specific pages are first referenced, they cause page faults that bring in the data. The mapped memory is copy-on-write. Modified data is returned to the file only by a **write()** call.

EXAMPLE

```
kern_return_t r;
int           fd;
char         *memfile, *filename = "/tmp/myfile";

/* Open the file. */
fd = open(filename, O_RDONLY);

/* Map part of it into memory. */
r = map_fd(fd, (vm_offset_t)0, &(vm_offset_t)memfile, TRUE,
    (vm_size_t)5);
if (r != KERN_SUCCESS)
    mach_error("Error calling map_fd()", r);
else
    printf("Second character in %s is:  %c\n", filename, memfile[1]);
```

RETURN

KERN_SUCCESS: The data was mapped successfully.

KERN_INVALID_ADDRESS: *address* wasn't valid.

KERN_INVALID_ARGUMENT: An invalid argument was passed.

msg_receive()

SUMMARY Receive a message

SYNOPSIS

#include <mach.h>
#include <sys/message.h>

msg_return_t **msg_receive**(msg_header_t **header*, msg_option_t *option*, msg_timeout_t *timeout*)

ARGUMENTS

header: The address of a buffer in which the message is to be received. Two fields of the message header must be set before the call is made: **msg_local_port** must be set to the value of the port from which the message is to be received, and **msg_size** must be set to the maximum size of the message that may be received. This maximum size must be less than or equal to the size of the buffer.

option: The failure conditions under which **msg_receive()** should terminate; the value of this parameter is an ORed combination of the following options. Unless one of these values is explicitly specified, **msg_receive()** does not return until a message has been received.

RCV_TIMEOUT: Specifies that **msg_receive()** should return when the specified timeout elapses, if a message has not arrived by that time; if not specified, the timeout will be ignored (that is, it will be infinite).

RCV_INTERRUPT: Specifies that **msg_receive()** should return when a software interrupt occurs in this thread.

RCV_LARGE: Specifies that **msg_receive()** should return without dequeuing a message if the next message in the queue is larger than *header*.**msg_size**. (Normally, a message that is too large is dequeued and lost.) You can use this option to dynamically determine how large your message buffer must be.

Use MSG_OPTION_NONE to specify that none of the above options is desired.

timeout: If RCV_TIMEOUT is specified in *option*, then *timeout* is the maximum time in milliseconds to wait for a message before giving up.

DESCRIPTION

The function **msg_receive()** retrieves the next message from the port or port set specified in the **msg_local_port** field of *header*. If a port is specified, the port must not be a member of a port set.

If a port set is specified, then **msg_receive()** will retrieve messages sent to any of the set's member ports. Mach sets the **msg_local_port** field to the specific port on which the message was found. It's not an error for the port set to have no members, or for members to be added and removed from a port set while a **msg_receive()** on the port set is in progress.

The message consists of its header, followed by a variable amount of data; the message header supplied to **msg_receive()** must specify (in **msg_size**) the maximum size of the message that can be received into the buffer provided.

If no messages are present on the port(s) in question, **msg_receive()** will wait until a message arrives, or until one of the specified termination conditions is met (see the description of the *option* parameter above).

If the message is successfully received, then **msg_receive()** sets the **msg_size** field of the header to the size of the received message. If the RCV_LARGE option was set and **msg_receive()** returned RCV_TOO_LARGE, then the **msg_size** field is set to the size of the message that was too large.

If the received message contains out-of-line data (that is, data for which the **msg_type_inline** attribute was specified as false), the data will be returned in a newly allocated region of memory; the message body will contain a pointer to that new region. You should deallocate this memory when the data is no longer needed. See the **vm_allocate**() call for a description of the state of newly allocated memory.

See Chapter 2, "Using Mach Messages," for information on setting up messages and on writing Mach servers.

EXAMPLE

```
msg_header_t   *imsg, header;

/* Wait for messages. */
while (1) {
    /* Set up the message structure. */
    header.msg_size = sizeof header;
    header.msg_local_port = receive_port;

    /* Get the next message on the queue. */
    r = msg_receive(&header, RCV_LARGE, 0);

    /* If the message is too big ... */
    if (r==RCV_TOO_LARGE) {
        /* ... allocate a structure for it ... */
        imsg = (msg_header_t *)malloc(header.msg_size);
        /* ... initialize the structure ... */
        imsg->msg_size = header.msg_size;
        imsg->msg_local_port = receive_port;
        /* ... and get the message. */
        r = msg_receive(imsg, MSG_OPTION_NONE, 0);
    }

    if (r==RCV_SUCCESS) {
        /* Handle the message. */
    }
    else { /* msg_receive() returned an error. */
        mach_error("msg_receive", r);
        exit(3);
    }
}
```

RETURN

RCV_SUCCESS: The message has been received.

RCV_INVALID_MEMORY: The message specified was not writable by the calling task.

RCV_INVALID_PORT: An attempt was made to receive on a port to which the calling task does not have the proper access, or which was deallocated (see **port_deallocate**()) while waiting for a message.

RCV_TOO_LARGE: The message header and body combined are larger than the size specified by **msg_size**. Unless the RCV_LARGE option was set, the message has been dequeued and lost. If the RCV_LARGE option was specified, then Mach sets **msg_size** to the size of the too-large message and leaves the message at the head of the queue.

RCV_NOT_ENOUGH_MEMORY: The message to be received contains more out-of-line data than can be allocated in the receiving task.

RCV_TIMED_OUT: The message was not received after *timeout* milliseconds.

RCV_INTERRUPTED: A software interrupt occurred and the RCV_INTERRUPT option was specified.

RCV_PORT_CHANGE: The port specified was added to a port set during the duration of the **msg_receive**() call.

msg_rpc()

SUMMARY Send and receive a message

SYNOPSIS

#include <mach.h>
#include <sys/message.h>

msg_return_t **msg_rpc**(msg_header_t **header*, msg_option_t *option*, msg_size_t *rcv_size*, msg_timeout_t *send_timeout*, msg_timeout_t *rcv_timeout*)

ARGUMENTS

header: Address of a message buffer that will be used for both **msg_send**() and **msg_receive**(). This buffer contains a message header followed by the data for the message to be sent. The **msg_remote_port** field specifies the port to which the message is to be sent. The **msg_local_port** field specifies the port on which a message is then to be received; if this port is the special value PORT_DEFAULT, it gets replaced by the value PORT_NULL for the purposes of the **msg_send**() operation.

option: A union of the *option* parameters for the send and receive (see **msg_send()** and **msg_receive**()).

rcv_size: The maximum size allowed for the received message; this must be less than or equal to the size of the message buffer. The **msg_size** field in the header specifies the size of the message to be sent.

send_timeout, *rcv_timeout*: The timeout values to be applied to the component operations. These are used only if the option SEND_TIMEOUT or RCV_TIMEOUT is specified.

DESCRIPTION

The function **msg_rpc**() is a hybrid call that performs a **msg_send**() followed by a **msg_receive**(), using the same message buffer. Because of the order of the send and receive, this function is appropriate for clients of Mach servers. However, the **msg_rpc**() call to a Mach server is usually performed by MiG-generated code, not by handwritten code.

See Chapter 2, "Using Mach Messages," for information on setting up messages and on writing Mach servers.

RETURN

RPC_SUCCESS: The message was successfully sent and a reply was received.

Other possible values are the same as those for **msg_send**() and **msg_receive**(); any error during the **msg_send**() portion will terminate the call.

msg_send()

SUMMARY Send a message

SYNOPSIS

#include <mach.h>
#include <sys/message.h>

msg_return_t **msg_send**(msg_header_t **header*, msg_option_t *option*,
msg_timeout_t *timeout*)

ARGUMENTS

header: The address of the message to be sent. A message consists of a fixed-size header followed by a variable number of data descriptors and data items. See the header file **sys/message.h** for a definition of the message structure.

option: The failure conditions under which **msg_send**() should terminate; the value of this parameter is an ORed combination of the following options. Unless one of the following values is explicitly specified, **msg_send**() does not return until the message is successfully queued for the intended receiver.

SEND_TIMEOUT: Specifies that the **msg_send**() request should terminate after the timeout period has elapsed, even if the kernel has been unable to queue the message.

SEND_NOTIFY: Allows the sender to send exactly one message without being suspended even if the destination port is full. When that message can be posted to the receiving port's queue, this task receives a message that notifies it that another message can be sent. A second attempt to send a message with the notify option to the same port before the first notification arrives results in an error. If SEND_TIMEOUT is also specified, **msg_send()** will wait until the specified timeout has elapsed before invoking the SEND_NOTIFY option.

SEND_INTERRUPT: Specifies that **msg_send()** should return if a software interrupt occurs in this thread.

Use MSG_OPTION_NONE to specify that none of the above options is wanted.

timeout: If the destination port is full and the SEND_TIMEOUT option has been specified, this value specifies the maximum wait time (in milliseconds).

DESCRIPTION

The function **msg_send()** transmits a message from the current task to the port specified in the message header field. The message consists of its header, followed by a variable number of data descriptors and data items.

If the **msg_local_port** field isn't set to PORT_NULL, send rights to that port will be passed to the receiver of this message. The receiver task can use that port to send a reply to this message.

If the SEND_NOTIFY option is used and this call returns a SEND_WILL_NOTIFY code, you can expect to receive a notify message from the kernel. This message will be either a NOTIFY_MSG_ACCEPTED or a NOTIFY_PORT_DELETED message, depending on what happened to the queued message. The **notify_port** field in these messages is the port to which the original message was sent. The formats for these messages are defined in the header file **sys/notify.h**.

See Chapter 2, "Using Mach Messages," for information on setting up messages and on writing Mach servers.

EXAMPLE

```
/* From the handwritten part of a Mach server... */
while (TRUE)
{
    /* Receive a request from a client. */
    msg.head.msg_local_port = port;
    msg.head.msg_size = sizeof(struct message);
    ret = msg_receive(&msg.head, MSG_OPTION_NONE, 0);
    if (ret != RCV_SUCCESS) /* ignore errors */;

    /* Feed the request into the server. */
    (void)add_server(&msg, &reply);

    /* Send a reply to the client. */
    reply.head.msg_local_port = port;
    ret = msg_send(&reply.head, MSG_OPTION_NONE, 0);
    if (ret != SEND_SUCCESS) /* ignore errors */;
}
```

RETURN

SEND_SUCCESS: The message has been queued for the destination port.

SEND_INVALID_MEMORY: The message header or body was not readable by the calling task, or the message body specified out-of-line data that was not readable.

SEND_INVALID_PORT: The message refers either to a port for which the current task does not have access, or to which access was explicitly removed from the current task (see **port_deallocate**()) while waiting for the message to be posted, or a **msg_type_name** field in the message specifies rights that the name doesn't denote in the task (for example, specifying MSG_TYPE_SEND and supplying a port set's name).

SEND_TIMED_OUT: The message was not sent since the destination port was still full after *timeout* milliseconds.

SEND_WILL_NOTIFY: The destination port was full but the SEND_NOTIFY option was specified. A notification message will be sent when the message can be posted.

SEND_NOTIFY_IN_PROGRESS: The SEND_NOTIFY option was specified but a notification request is already outstanding for this thread and given destination port.

port_allocate()

SUMMARY Create a port

SYNOPSIS

#include <mach.h>

kern_return_t **port_allocate**(task_t *task*, port_name_t **port_name*)

ARGUMENTS

task: The task in which the new port is created (for example, use **task_self**() to specify the caller's task).

port_name: Returns the task's name for the new port.

DESCRIPTION

The function **port_allocate**() causes a port to be created for the specified task; the resulting port is returned in *port_name*. The target task initially has both send and receive rights to the port. The new port isn't a member of any port set.

EXAMPLE

```
port_t          myport;
kern_return_t   error;

if ((error=port_allocate(task_self(), &myport)) != KERN_SUCCESS) {
    mach_error("port_allocate failed", error);
    exit(1);
}
```

RETURN

KERN_SUCCESS: A port has been allocated.

KERN_INVALID_ARGUMENT: *task* was invalid.

KERN_RESOURCE_SHORTAGE: No more port slots are available for this task.

SEE ALSO

port_deallocate()

port_deallocate()

SUMMARY Deallocate a port

SYNOPSIS

#include <mach.h>

kern_return_t **port_deallocate**(task_t *task*, port_name_t *port_name*)

ARGUMENTS

task: The task that wants to relinquish rights to the port (for example, use **task_self()** to specify the caller's task).

port_name: *task*'s name for the port to be deallocated.

DESCRIPTION

The function **port_deallocate()** requests that the target task's access to a port be relinquished.

If *task* has receive rights for the port and the port doesn't have a backup port, these things happen:

- The port is destroyed.
- All other tasks with send access to the port are notified of the port's destruction.
- If the port is a member of a port set, it's removed from the port set.

If *task* has receive rights for the port and the port *does* have a backup port, then the following things happen:

- If the port is a member of a port set, it's removed from the port set.
- Send and receive rights for the port are sent to the backup port in a notification message (see **port_set_backup**()).

EXAMPLE

```
port_t         my_port;
kern_return_t  error;

/* . . . */

error=port_deallocate(task_self(), my_port);
if (error != KERN_SUCCESS) {
    mach_error("port_deallocate failed", error);
    exit(1);
}
```

RETURN

KERN_SUCCESS: The port has been deallocated.

KERN_INVALID_ARGUMENT: *task* was invalid or *port_name* doesn't name a valid port.

SEE ALSO

port_allocate()

port_extract_receive(), port_extract_send()

SUMMARY Remove a task's rights for a port and return them to the caller

SYNOPSIS

#include <mach.h>

kern_return_t **port_extract_receive**(task_t *task*, port_name_t *its_name*, port_t **its_port*)
kern_return_t **port_extract_send**(task_t *task*, port_name_t *its_name*, port_t **its_port*)

ARGUMENTS

task: The task whose rights the caller takes.

its_name: The name by which *task* knows the port.

its_port: Returns the receive or send rights.

DESCRIPTION

The functions **port_extract_receive**() and **port_extract_send**() remove *task*'s rights for a port and return the rights to the caller. *task* is left with no rights for the port.

port_extract_send() extracts send rights; *task* can't have receive rights for the named port. **port_extract_receive**() extracts receive rights.

RETURN

KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *task* was invalid or *its_name* doesn't name a port for which *task* has the required rights.

SEE ALSO

port_insert_send(), **port_insert_receive**()

port_extract_send() → See port_extract_receive()

port_insert_receive(), port_insert_send()

SUMMARY Give a task rights with a specific name

SYNOPSIS

#include <mach.h>

kern_return_t **port_insert_receive**(task_t *task*, port_t *my_port*, port_name_t *its_name*)
kern_return_t **port_insert_send**(task_t *task*, port_t *my_port*, port_name_t *its_name*)

ARGUMENTS

task: The task getting the new rights.

my_port: Rights supplied by the caller.

its_name: The name by which task will know the new rights.

DESCRIPTION

The functions **port_insert_receive**() and **port_insert_send**() give a task rights with a specific name. If *task* already has rights named *its_name*, or has some other name for *my_port*, the operation will fail. *its_name* can't be a predefined port, such as PORT_NULL.

port_insert_send() inserts send rights, and **port_insert_receive**() inserts receive rights.

RETURN

KERN_SUCCESS: The call succeeded.

KERN_NAME_EXISTS: *task* already has a right named *its_name*.

KERN_FAILURE: *task* already has rights to *my_port*.

KERN_INVALID_ARGUMENT: *task* was invalid or *its_name* was an invalid name.

SEE ALSO

port_extract_send(), **port_extract_receive()**

port_insert_send() → See port_insert_receive()

port_names()

SUMMARY Get information about a task's port name space

SYNOPSIS

#include <mach.h>

kern_return_t **port_names**(task_t *task*, port_name_array_t **port_names*,
unsigned int **port_names_count*, port_type_array_t **port_types*,
unsigned int **port_types_count*)

ARGUMENTS

task: The task whose port name space is queried.

port_names: Returns the names of the ports and port sets in the task's port name space, in no particular order.

port_names_count: Returns the number of names returned.

port_types: Returns the type of each corresponding name. This indicates what kind of rights the task holds for the port, or whether the name refers to a port set. The type is one of the following: PORT_TYPE_SEND (send rights only), PORT_TYPE_RECEIVE_OWN (send and receive rights), PORT_TYPE_SET (the port is a port set).

port_types_count: Returns the same value as *port_names_count*.

DESCRIPTION

The function **port_names()** returns information about *task*'s port name space. It returns *task*'s currently valid port and port set names. For each name, it also returns what type of rights *task* holds.

port_names and *port_types* are arrays that are automatically allocated when the reply message is received. You should use **vm_deallocate**() on them when the data is no longer needed.

EXAMPLE

```
kern_return_t      error;
port_name_array_t  names;
unsigned int       names_count, types_count;
port_type_array_t  types;

error=port_names(task_self(), &names, &names_count, &types,
    &types_count);
if (error != KERN_SUCCESS) {
    mach_error("port_rename returned value of ", error);
    exit(1);
}
/* . . . */
error=vm_deallocate(task_self(), (vm_address_t)names,
    sizeof(names)*names_count);
if (error != KERN_SUCCESS)
    mach_error("Trouble freeing names", error);

error=vm_deallocate(task_self(), (vm_address_t)types,
    sizeof(names)*types_count);
if (error != KERN_SUCCESS)
    mach_error("Trouble freeing types", error);
```

RETURN

KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *task* was invalid.

SEE ALSO

port_type(), **port_status**(), **port_set_status**()

port_rename()

SUMMARY Change the name by which a port or port set is known to a task

SYNOPSIS

#include <mach.h>

kern_return_t **port_rename**(task_t *task*, port_name_t *old_name*,
port_name_t *new_name*)

ARGUMENTS

task: The task whose port name space is changed.

old_name: The current name of the port or port set.

new_name: The new name for the port or port set.

DESCRIPTION

The function **port_rename()** changes the name by which a port or port set is known to *task*. *new_name* must not already be in use, and it can't be a predefined port, such as PORT_NULL. Currently, a name is a small integer.

One way to guarantee that a name isn't already in use is to deallocate a port and then use its name as *new_name*. Another way is to check all the existing names, using **port_names()**, before you call **port_rename()**. If you choose another naming scheme, you should be prepared to try another name if **port_rename()** returns a KERN_NAME_EXISTS error.

EXAMPLE

```
#define MY_PORT  (port_name_t)99

port_name_t        my_port;
kern_return_t      error;

error=port_allocate(task_self(),&my_port);
if (error != KERN_SUCCESS) {
    mach_error("port_allocate failed", error);
    exit(1);
}

error=port_rename(task_self(), my_port, MY_PORT);
if (error == KERN_NAME_EXISTS)
    /* try again with a different name */;
else if (error != KERN_SUCCESS) {
        mach_error("port_rename failed", error);
        exit(1);
    }
```

RETURN

KERN_SUCCESS: The call succeeded.

KERN_NAME_EXISTS: *task* already has a port or port set named *new_name*.

KERN_INVALID_ARGUMENT: *task* was invalid, or *task* didn't know any ports or port sets named *old_name*, or *new_name* was an invalid name.

SEE ALSO

port_names()

port_set_add()

SUMMARY Move the named port into the named port set

SYNOPSIS

#include <mach.h>

kern_return_t **port_set_add**(task_t *task*, port_set_name_t *set_name*,
port_name_t *port_name*)

ARGUMENTS

task: The task that has receive rights for the port set and port.

set_name: *task*'s name for the port set.

port_name: *task*'s name for the port.

DESCRIPTION

The function **port_set_add()** moves the named port into the named port set. *task* must have receive rights for the port. If the port is already a member of another port set, it's removed from that set first.

EXAMPLE

```
kern_return_t    error;
port_set_name_t  set_name;
port_t           my_port;

error=port_set_allocate(task_self(),&set_name);
if (error != KERN_SUCCESS) {
    mach_error("port_set_allocate failed", error);
    exit(1);
}

error=port_allocate(task_self(),&my_port);
if (error != KERN_SUCCESS) {
    mach_error("port_allocate failed", error);
    exit(1);
}

error=port_set_add(task_self(), set_name, my_port);
if (error != KERN_SUCCESS) {
    mach_error("port_allocate failed", error);
    exit(1);
}
```

RETURN

KERN_SUCCESS: The call succeeded.

KERN_NOT_RECEIVER: *task* doesn't have receive rights for the port.

KERN_INVALID_ARGUMENT: *task* was invalid, or *set_name* doesn't name a valid port set, or *port_name* doesn't name a valid port.

SEE ALSO

port_set_remove()

port_set_allocate()

SUMMARY Create a port set

SYNOPSIS

#include <mach.h>

kern_return_t **port_set_allocate**(task_t *task*, port_set_name_t **set_name*)

ARGUMENTS

task: The task in which the new port set is created.

set_name: Returns the task's name for the new port set.

DESCRIPTION

The function **port_set_allocate()** causes a port set to be created for the specified task; the resulting set's name is returned in *set_name.* The new port set is empty.

EXAMPLE

```
kern_return_t    error;
port_set_name_t  set_name;

error=port_set_allocate(task_self(),&set_name);
if (error != KERN_SUCCESS) {
    mach_error("port_set_allocate failed", error);
    exit(1);
}
```

RETURN

KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *task* was invalid.

KERN_RESOURCE_SHORTAGE: The kernel ran out of memory.

SEE ALSO

port_set_deallocate(), port_set_add()

port_set_backlog()

SUMMARY Set the size of the port queue

SYNOPSIS

#include <mach.h>

kern_return_t **port_set_backlog**(task_t *task*, port_name_t *port_name*, int *backlog*)

ARGUMENTS

task: The task that has receive rights for the named port (for example, use **task_self()** to specify the caller's task).

port_name: *task*'s name for the port.

backlog: The new backlog to be set.

DESCRIPTION

The function **port_set_backlog()** changes the backlog value on the specified port (the port's backlog value is the number of unreceived messages that are allowed in its message queue before the kernel will refuse to accept any more sends to that port).

task must have receive rights for the named port.

The current value of a port's backlog can be found by the **port_status()** call. The maximum backlog value is the constant PORT_BACKLOG_MAX. You can get the current backlog by calling **port_status()**.

EXAMPLE

```
#define MY_BACKLOG  10

kern_return_t   error;
port_t          my_port;

error=port_allocate(task_self(),&my_port);
if (error != KERN_SUCCESS) {
    mach_error("port_allocate failed", error);
    exit(1);
}

error=port_set_backlog(task_self(), my_port, MY_BACKLOG);
if (error!=KERN_SUCCESS)
    mach_error("Call to port_set_backlog failed", error);
```

RETURN

KERN_SUCCESS: The backlog value has been changed.

KERN_NOT_RECEIVER: *task* doesn't have receive rights for the port.

KERN_INVALID_ARGUMENT: *task* was invalid, or *port_name* doesn't name a valid port, or the desired backlog wasn't greater than 0, or the desired backlog was greater than PORT_BACKLOG_MAX.

SEE ALSO

msg_send(), **port_status**()

port_set_backup()

SUMMARY Set the backup port for a port

SYNOPSIS

#include <mach.h>

kern_return_t **port_set_backup**(task_t *task*, port_name_t *port_name*, port_t *backup*, port_t **previous*)

ARGUMENTS

task: The task that has receive rights for the named port (for example, use **task_self**() to specify the caller's task).

port_name: *task*'s name for the port right.

backup: The new backup port. If you want to disable the current backup port without setting a new one, set this to PORT_NULL.

previous: Returns the previous backup port.

DESCRIPTION

Use this function to keep a port alive despite its being deallocated by its receiver. If the call to **port_set_backup**() is successful, then whenever *port_name* is deallocated by its receiver, *backup* will receive a notification message with receive and send rights for *port_name*. As far as *task* is concerned, the port will be deleted; however, as far as senders to the port are concerned, the port will continue to exist.

To let a port die naturally after its backup port has been set, call **port_set_backup**() on it with *backup* set to PORT_NULL.

EXAMPLE

```
kern_return_t    error;
port_t           my_port, backup_port, previous_port;

error=port_allocate(task_self(),&my_port);
if (error != KERN_SUCCESS) {
    mach_error("port_allocate failed", error);
    exit(1);
}

error=port_allocate(task_self(),&backup_port);
if (error != KERN_SUCCESS) {
    mach_error("port_allocate failed", error);
    exit(1);
}

error=port_set_backup(task_self(), my_port, backup_port,
    &previous_port);
if (error!=KERN_SUCCESS)
    mach_error("Call to port_set_backlog failed", error);
```

RETURN

KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *task* was invalid, or *port_name* doesn't name a valid port.

KERN_NOT_RECEIVER: *task* doesn't have receive rights for *port_name*.

port_set_deallocate()

SUMMARY Destroy a port set

SYNOPSIS

#include <mach.h>

kern_return_t **port_set_deallocate**(task_t *task*, port_set_name_t *set_name*)

ARGUMENTS

task: The task that has receive rights for the port set to be destroyed.

set_name: *task*'s name for the doomed port set.

DESCRIPTION

The function **port_set_deallocate()** requests that the target task's port set be destroyed. If the port set isn't empty, any members are first removed.

EXAMPLE

```
kern_return_t   error;
port_set_name_t set_name;

error=port_set_deallocate(task_self(),set_name);
if (error != KERN_SUCCESS) {
    mach_error("port_set_deallocate failed", error);
    exit(1);
}
```

RETURN

KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *task* was invalid or *set_name* doesn't name a valid port set.

SEE ALSO

port_set_allocate()

port_set_remove()

SUMMARY Remove the named port from a port set

SYNOPSIS

#include <mach.h>

kern_return_t **port_set_remove**(task_t *task*, port_name_t *port_name*)

ARGUMENTS

task: The task that has receive rights for the port and port set.

port_name: *task*'s name for the receive rights to be removed.

DESCRIPTION

The function **port_set_remove()** removes the named port from a port set. *task* must have receive rights for the port, and the port must be a member of a port set.

EXAMPLE

```
error=port_set_remove(task_self(), my_port);
if (error != KERN_SUCCESS) {
    mach_error("port_set_remove failed", error);
    exit(1);
}
```

RETURN

KERN_SUCCESS: The call succeeded.

KERN_NOT_RECEIVER: *task* doesn't have receive rights for the port.

KERN_NOT_IN_SET: The port isn't a member of a set.

KERN_INVALID_ARGUMENT: *task* was invalid or *port_name* doesn't name a valid port.

SEE ALSO

port_set_add()

port_set_status()

SUMMARY Get the members of a port set

SYNOPSIS

#include <mach.h>

kern_return_t **port_set_status**(task_t *task*, port_set_name_t *set_name*,
port_name_array_t **members*, unsigned int **members_count*)

ARGUMENTS

task: The task whose port set is queried.

set_name: *task*'s name for the port set.

members: Returns *task*'s names for the port set's members.

members_count: Returns the number of port names in *members*.

DESCRIPTION

The function **port_set_status()** returns a list of the ports in a port set. *members* is an array that's automatically allocated when the reply message is received. You should use **vm_deallocate()** on it when the data is no longer needed.

EXAMPLE

```
error=port_set_status(task_self(), set_name, &members,
   &members_count);
if (error != KERN_SUCCESS) {
    mach_error("port_set_status failed", error);
    exit(1);
}

/* . . . */
error=vm_deallocate(task_self(), (vm_address_t)members,
    sizeof(members)*members_count);
if (error != KERN_SUCCESS)
    mach_error("Trouble freeing members", error);
```

RETURN

KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *task* was invalid or *set_name* doesn't name a valid port set.

SEE ALSO

port_status()

port_status()

SUMMARY Examine a port's current status

SYNOPSIS

#include <mach.h>

kern_return_t **port_status**(task_t *task*, port_name_t *port_name*, port_set_name_t **port_set_name*, int **num_msgs*, int **backlog*, boolean_t **owner*, boolean_t **receiver*)

ARGUMENTS

task: The task that has receive rights for the port in question (for example, use **task_self**() to specify the caller's task).

port_name: *task*'s name for the port right.

port_set_name: Returns *task*'s name for the port set that the named port belongs to, or PORT_NULL if it isn't in a set.

num_msgs: Returns the number of messages queued on this port. If *task* isn't the port's receiver, the number of messages will be returned as negative.

backlog: Returns the number of messages that can be queued to this port without causing the sender to block.

owner: Returns the same value as *receiver*, since ownership rights and receive rights aren't separable.

receiver: Returns true if *task* has receive rights to *port_name*; otherwise, returns false.

DESCRIPTION

The function **port_status**() returns the current port status associated with *port_name*.

EXAMPLE

```
error=port_status(task_self(), my_port, &port_set_name, &num_msgs,
    &backlog, &owner, &receiver);
if (error!=KERN_SUCCESS)
    mach_error("Call to port_status failed", error);
```

RETURN

KERN_SUCCESS: The data has been retrieved.

KERN_INVALID_ARGUMENT: *task* was invalid or *port_name* doesn't name a valid port.

SEE ALSO

port_set_backlog(), **port_set_status**()

port_type()

SUMMARY Get a task's rights for a specific name in its port name space

SYNOPSIS

#include <mach.h>

kern_return_t **port_type**(task_t *task*, port_name_t *port_name*, port_type_t **port_type*)

ARGUMENTS

task: The task whose port name space is queried.

port_name: The name being queried.

port_type: Returns a value that indicates what kind of rights the task holds for the port, or whether the name refers to a port set. *port_type* is one of the following: PORT_TYPE_SEND (send rights only), PORT_TYPE_RECEIVE_OWN (send and receive rights), PORT_TYPE_SET (the port is a port set).

DESCRIPTION

The function **port_type**() returns information about *task*'s rights for a specific name in its port name space.

EXAMPLE

```
error=port_type(task_self(), port, &type);
if (error != KERN_SUCCESS)
    mach_error("Couldn't get type of port", error);
```

RETURN

KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *task* was invalid or *task* didn't have any rights named *port_name*.

SEE ALSO

port_names(), **port_status()**, **port_set_status()**

processor_assign(), processor_control(), processor_exit(), processor_get_assignment(), processor_start()

SUMMARY Start up a processor

SYNOPSIS

#include <mach.h>

kern_return_t **processor_assign**(processor_t *processor*,
processor_set_t *new_processor_set*, boolean_t *wait*)
kern_return_t **processor_control**(processor_t *processor*, processor_info_t *info*,
long **count*)
kern_return_t **processor_exit**(processor_t *processor*)
kern_return_t **processor_get_assignment**(processor_t *processor*,
processor_set_t **processor_set*)
kern_return_t **processor_start**(processor_t *processor*)

DESCRIPTION

processor_assign() changes the processor set to which *processor* is assigned. **processor_control**() returns information about *processor*. **processor_exit**() shuts down *processor*. **processor_get_assignment**() returns the processor set to which *processor* is assigned. **processor_start**() starts up *processor*.

Note: These functions are useful only on multiprocessor systems.

processor_info()

SUMMARY Get information about a processor

SYNOPSIS

#include <mach.h>

kern_return_t **processor_info**(processor_t *processor*, int *flavor*, host_t **host*,
processor_info_t *processor_info*, unsigned int **processor_info_count*)

ARGUMENTS

processor: The processor for which information is to be obtained.

flavor: The type of information that is wanted. Currently only PROCESSOR_BASIC_INFO is implemented.

host: Returns the non-privileged host port for the host on which the processor resides.

processor_info: Returns information about the processor specified by *processor*.

processor_info_count: Size of the info structure. Should be PROCESSOR_BASIC_INFO_COUNT for flavor PROCESSOR_BASIC_INFO.

DESCRIPTION

Returns the selected information array for a processor, as specified by *flavor*. *processor_info* is an array of integers that is supplied by the caller and filled with specified information. *processor_info_count* is supplied as the maximum number of integers in *processor_info*. On return, it contains the actual number of integers in *processor_info*.

Basic information is defined by PROCESSOR_BASIC_INFO. The size of this information is defined by PROCESSOR_BASIC_INFO_COUNT. The data structures used by PROCESSOR_BASIC_INFO are defined in the header file **sys/processor_info.h**. Possible values of the **cpu_type** and **cpu_subtype** fields are defined in the header file **sys/machine.h**.

```
typedef int *processor_info_t;  /* variable length array of int */

/* one interpretation of info is */
struct processor_basic_info {
    cpu_type_t     cpu_type;     /* cpu type */
    cpu_subtype_t  cpu_subtype;  /* cpu subtype */
    boolean_t      running;      /* is processor running? */
    int            slot_num;     /* slot number */
    boolean_t      is_master;    /* is this the master processor */
};

typedef struct processor_basic_info *processor_basic_info_t;
```

EXAMPLE

```
kern_return_t                  error;
host_t                         host;
unsigned int                   list_size, info_count;
struct processor_basic_info    info;
processor_array_t              list;

/* Get the processor port. */
error=host_processors(host_priv_self(), &list, &list_size);
if ((error!=KERN_SUCCESS) || (list_size < 1)){
    mach_error("Error calling host_processors (are you root?)",
        error);
    exit(1);
}

/* Get information about the processor. */
info_count=PROCESSOR_BASIC_INFO_COUNT;
error=processor_info(list[0], PROCESSOR_BASIC_INFO, &host,
    (processor_info_t)&info, &info_count);
if (error != KERN_SUCCESS)
    mach_error("Error calling processor_info", error);

/* Now that we're done with the processor port, free it. */
vm_deallocate(task_self(), (vm_address_t)list,
    sizeof(list)*list_size);
if (error!=KERN_SUCCESS)
    mach_error("Trouble freeing list", error);
```

RETURN

KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *processor* isn't a known processor.

MIG_ARRAY_TOO_LARGE: Returned info array is too large for *processor_info*. *processor_info* is filled as much as possible. *processor_info_count* is set to the number of elements that would be returned if there were enough room.

KERN_FAILURE: *flavor* isn't recognized or *processor_info_count* is too small.

SEE ALSO

processor_start(), processor_exit(), processor_control(), host_processors()

processor_set_create()

SUMMARY Create a new processor set

SYNOPSIS

#include <mach.h>

kern_return_t **processor_set_create**(host_t *host*, port_t **new_set*, port_t **new_name*)

DESCRIPTION

This function creates a new processor set on *host*.

Note: This function is useful only on multiprocessor systems.

processor_set_default()

SUMMARY Get the port of the default processor set

SYNOPSIS

#include <mach.h>

kern_return_t **processor_set_default**(host_t *host*, processor_set_t **default_set*)

ARGUMENTS

host: The host whose default processor set is requested.

default_set: Returns the name (non-privileged) port for the default processor set.

DESCRIPTION

The default processor set is used by all threads, tasks, and processors that aren't explicitly assigned to other sets. **processor_set_default()** returns a port that can be used to obtain information about this set (for example, how many threads are assigned to it). This port can't be used to perform operations on that set (call **host_processor_set_priv**() after **processor_set_default()** to get the privileged port).

EXAMPLE

```
error=processor_set_default(host_self(), &default_set);
if (error!=KERN_SUCCESS){
    mach_error("Error calling processor_set_default", error);
    exit(1);
}
```

RETURN

KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *host* is not a host.

SEE ALSO

processor_set_info(), task_assign(), thread_assign()

processor_set_destroy()

SUMMARY Delete a processor set

SYNOPSIS

#include <mach.h>

kern_return_t **processor_set_destroy**(processor_set_t *processor_set*)

DESCRIPTION

This function destroys *processor_set*, reassigning all of its tasks, threads, and processors to the default processor set.

Note: This function is useful only on multiprocessor systems.

processor_set_info()

SUMMARY Get information about a processor set

SYNOPSIS

#include <mach.h>

kern_return_t **processor_set_info**(processor_set_t *processor_set*, int *flavor*,
host_t **host*, processor_set_info_t *processor_set_info*,
unsigned int **processor_set_info_count*)

ARGUMENTS

processor_set: The processor set for which information is to be obtained.

flavor: The type of information that is wanted. Should be PROCESSOR_SET_BASIC_INFO or PROCESSOR_SET_SCHED_INFO.

host: Returns the non-privileged host port for the host on which the processor set resides.

processor_set_info: Returns information about the processor set specified by *processor_set*.

processor_set_info_count: Size of the info structure. Should be PROCESSOR_SET_BASIC_INFO_COUNT for flavor PROCESSOR_SET_BASIC_INFO, and PROCESSOR_SET_SCHED_INFO_COUNT for flavor PROCESSOR_SET_SCHED_INFO.

DESCRIPTION

Returns the selected information array for a processor set, as specified by *flavor*. *processor_set_info* is an array of integers that is supplied by the caller, and filled with specified information. *processor_set_info_count* is supplied as the maximum number of integers in *processor_set_info*. On return, it contains the actual number of integers in *processor_set_info*.

Basic information is defined by PROCESSOR_SET_BASIC_INFO. The size of this information is defined by PROCESSOR_SET_BASIC_INFO_COUNT. The **load_average** and **mach_factor** fields are scaled by the constant LOAD_SCALE (that is, the integer value returned is the load average or Mach factor multiplied by LOAD_SCALE).

The *Mach factor*, like the UNIX load average, is a measurement of how busy the system is. Unlike the load average, higher Mach factors mean that the system is less busy. The Mach factor tells you how much of a CPU you have available for running an application. For example, on a single-processor system with one job running, the Mach factor is 0.5; this means if another job starts running it will get half of the CPU. (Two jobs will be running, each getting half the CPU.) On a single-processor system, the Mach factor is between zero and one. On a multiprocessor system, the Mach factor can go over one. For example, a three-processor system with one job running has a Mach factor of 2.0, since two processors are available to new jobs.

```
struct processor_set_basic_info {
    int  processor_count;   /* number of processors */
    int  task_count;        /* number of tasks */
    int  thread_count;      /* number of threads */
    int  load_average;      /* scaled load average */
    int  mach_factor;       /* scaled mach factor */
};
typedef struct processor_set_basic_info
*processor_set_basic_info_t;
```

Scheduling information is defined by PROCESSOR_SET_SCHED_INFO. The size of this information is given by PROCESSOR_SET_SCHED_INFO_COUNT.

```
struct processor_set_sched_info {
    int  policies;          /* allowed policies */
    int  max_priority;      /* max priority for new threads */
};
typedef struct processor_set_sched_info
*processor_set_sched_info_t;
```

EXAMPLE

```
kern_return_t                  error;
host_t                         host;
unsigned int                   info_count;
struct processor_set_basic_info  info;
processor_set_t                default_set;

error=processor_set_default(host_self(), &default_set);
if (error!=KERN_SUCCESS){
    mach_error("Error calling processor_set_default", error);
    exit(1);
}

info_count=PROCESSOR_SET_BASIC_INFO_COUNT;
error=processor_set_info(default_set, PROCESSOR_SET_BASIC_INFO,
   &host, (processor_set_info_t)&info, &info_count);
if (error != KERN_SUCCESS)
    mach_error("Error calling processor_set_info", error);

printf("The UNIX load average is %f\n",
    (float)info.load_average/LOAD_SCALE);
printf("The Mach factor is %f\n", (float)info.mach_factor/LOAD_SCALE);
```

RETURN

KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *processor_set* is not a processor set, or *flavor* is not recognized.

KERN_FAILURE: *processor_set_info_count* is less than what it should be.

MIG_ARRAY_TOO_LARGE: Returned info array is too large for *processor_set_info*.

SEE ALSO

processor_set_create(), **processor_set_default()**, **processor_assign()**, **task_assign()**, **thread_assign()**

processor_set_max_priority()

SUMMARY Set the maximum priority permitted on a processor set

SYNOPSIS

#include <mach.h>

kern_return_t **processor_set_max_priority**(processor_set_t *processor_set*, int *max_priority*, boolean_t *change_threads*)

DESCRIPTION

This function affects only newly-created or newly-assigned threads unless you specify *change_threads* as true.

Note: This function is useful only on multiprocessor systems.

processor_set_policy_enable(), processor_set_policy_disable()

SUMMARY Enable or disable a scheduling policy on a processor set

SYNOPSIS

#include <mach.h>

kern_return_t **processor_set_policy_enable**(processor_set_t *processor_set*,
int *policy*)
kern_return_t **processor_set_policy_disable**(processor_set_t *processor_set*,
int *policy*, boolean_t *change_threads*)

ARGUMENTS

processor_set: The processor set whose allowed policies are to be changed. This must be the privileged processor set port, which is returned by **host_processor_set_priv**().

policy: The policy to enable or disable. Currently, the only valid policies are POLICY_TIMESHARE, POLICY_INTERACTIVE, and POLICY_FIXEDPRI. You can't disable timesharing.

change_threads: Specify true if you want to reset to timesharing the policies of any threads with the newly-disallowed policy. Otherwise, specify false.

DESCRIPTION

Processor sets may restrict the scheduling policies to be used for threads assigned to them. These two calls provide the mechanism for designating permitted and forbidden policies. The current set of permitted policies can be obtained from **processor_set_info**(). Timesharing can't be forbidden by any processor set. This is a compromise to reduce the complexity of the assign operation; any thread whose policy is forbidden by the target processor set has its policy reset to timesharing. If the *change_threads* argument to **processor_set_policy_disable**() is true, threads currently assigned to this processor set and using the newly disabled policy will have their policy reset to timesharing.

Warning: Don't use POLICY_FIXEDPRI unless you're familiar with the consequences of fixed-priority scheduling. Using fixed-priority scheduling in a process can keep other processes from getting any CPU time. If processes that are essential to the functioning of the system don't get CPU time, you might have to reboot your system to make it work normally.

EXAMPLE

```
kern_return_t    error;
processor_set_t  default_set, default_set_priv;

error=processor_set_default(host_self(), &default_set);
if (error!=KERN_SUCCESS) {
    mach_error("Error calling processor_set_default()", error);
    exit(1);
}

error=host_processor_set_priv(host_priv_self(), default_set,
    &default_set_priv);
if (error != KERN_SUCCESS) {
    mach_error("Call to host_processor_set_priv() failed", error);
    exit(1);
}

error=processor_set_policy_enable(default_set_priv, POLICY_FIXEDPRI);
if (error != KERN_SUCCESS)
    mach_error("Error calling processor_set_policy_enable", error);
```

RETURN

KERN_SUCCESS: Operation completed successfully.

KERN_INVALID_ARGUMENT: *processor_set* isn't the privileged port of a processor set, *policy* isn't a valid policy, or an attempt was made to disable timesharing.

SEE ALSO

thread_policy(), **thread_switch**()

processor_set_tasks()

SUMMARY Get kernel ports for tasks assigned to a processor set

SYNOPSIS

#include <mach.h>

kern_return_t **processor_set_tasks**(processor_set_t *processor_set*, task_array_t **task_list*, unsigned int **task_count*)

ARGUMENTS

processor_set: The processor set to be affected. This must be the privileged processor set port, which is returned by **host_processor_set_priv**().

task_list: Returns the set of tasks currently assigned to *processor_set*; no particular ordering is guaranteed.

task_count: Returns the number of tasks in *task_list*.

DESCRIPTION

processor_set_tasks() gets send rights to the kernel port for each task currently assigned to *processor_set*. *task_list* is an array that is created as a result of this call. You should call **vm_deallocate()** on this array when you no longer need the data.

EXAMPLE

```
task_array_t    task_list;
unsigned int    task_count;
processor_set_t default_set, default_set_priv;
kern_return_t   error;

error=processor_set_default(host_self(), &default_set);
if (error!=KERN_SUCCESS) {
    mach_error("Error calling processor_set_default()", error);
    exit(1);
}

error=host_processor_set_priv(host_priv_self(), default_set,
    &default_set_priv);
if (error != KERN_SUCCESS) {
    mach_error("Call to host_processor_set_priv() failed", error);
    exit(1);
}

error=processor_set_tasks(default_set_priv, &task_list, &task_count);
if (error != KERN_SUCCESS) {
    mach_error("Call to processor_set_tasks() failed", error);
    exit(1);
}

/* . . . */
error=vm_deallocate(task_self(), (vm_address_t)task_list,
    sizeof(task_list)*task_count);
if (error != KERN_SUCCESS)
    mach_error("Trouble freeing task_list", error);
```

RETURN

KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *processor_set* isn't a privileged processor set.

SEE ALSO

task_assign(), thread_assign(), processor_set_threads()

processor_set_threads()

SUMMARY Get kernel ports for threads assigned to a processor set

SYNOPSIS

#include <mach.h>

kern_return_t **processor_set_threads**(processor_set_t *processor_set*, thread_array_t **thread_list*, unsigned int **thread_count*)

ARGUMENTS

processor_set: The processor set to be affected. This must be the privileged processor set port, which is returned by **host_processor_set_priv**().

thread_list: Returns the set of threads currently assigned to *processor_set*; no particular ordering is guaranteed.

thread_count: Returns the number of threads in *thread_list*.

DESCRIPTION

processor_set_threads() gets send rights to the kernel port for each thread currently assigned to *processor_set*. *thread_list* is an array that is created as a result of this call. You should call **vm_deallocate**() on *thread_list* when you no longer need the data.

EXAMPLE

```
thread_array_t  thread_list;
unsigned int    thread_count;
processor_set_t default_set, default_set_priv;
kern_return_t   error;

error=processor_set_default(host_self(), &default_set);
if (error!=KERN_SUCCESS) {
    mach_error("Error calling processor_set_default()", error);
    exit(1);
}

error=host_processor_set_priv(host_priv_self(), default_set,
    &default_set_priv);
if (error != KERN_SUCCESS) {
    mach_error("Call to host_processor_set_priv() failed", error);
    exit(1);
}

error=processor_set_threads(default_set_priv, &thread_list,
&thread_count);
if (error != KERN_SUCCESS) {
    mach_error("Call to processor_set_threads() failed", error);
    exit(1);
}
```

```
/* . . . */
error=vm_deallocate(task_self(), (vm_address_t)thread_list,
    sizeof(thread_list)*thread_count);
if (error != KERN_SUCCESS)
    mach_error("Trouble freeing thread_list", error);
```

RETURN

KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *processor_set* isn't a privileged processor set.

SEE ALSO

task_assign(), **thread_assign()**, **processor_set_tasks()**

processor_start() → See processor_assign()

task_assign(), task_assign_default()

SUMMARY Assign a task to a processor set

SYNOPSIS

#include <mach.h>

kern_return_t **task_assign**(task_t *task*, processor_set_t *new_processor_set*,
    boolean_t *assign_threads*)
kern_return_t **task_assign_default**(task_t *task*, boolean_t *assign_threads*)

DESCRIPTION

task_assign() assigns *task* to *new_processor_set*; **task_assign_default()** assigns *task* to the default processor set.

Note: These functions are useful only on multiprocessor systems.

task_by_unix_pid()

SUMMARY Get the task port for a UNIX process on the same host

SYNOPSIS

#include <mach.h>

kern_return_t **task_by_unix_pid**(task_t *task*, int *pid*, task_t **result_task*)

ARGUMENTS

task: A task that is used to check permission (usually **task_self**()).

pid: The process ID of the desired process.

result_task: Returns send rights to the task port of the process specified by *pid*.

DESCRIPTION

Returns the task port for another process, named by its process ID, on the same host as *task*. This call succeeds only if the caller is the superuser or *task* has the same user ID as the process specified by *pid*. If the call fails, *result_task* is set to TASK_NULL.

EXAMPLE

```
pid=fork();

if (pid==0) /* We're in the child. */ {
    /* do childish things */
}
else /* We're in the parent */ {
    result=task_by_unix_pid(task_self(), pid, &child_task);
    if (result != KERN_SUCCESS)
        mach_error("task_by_unix_pid", result);
    /* . . . */
}
```

RETURN

KERN_SUCCESS: The call succeeded.

KERN_FAILURE: *target_task* has a different user ID from *pid*'s process and the caller isn't the superuser, or *pid* didn't refer to a valid process, or *target_task* wasn't a valid task.

SEE ALSO

unix_pid()

task_create()

SUMMARY Create a task

SYNOPSIS

#include <mach.h>

kern_return_t **task_create**(task_t *parent_task*, boolean_t *inherit_memory*,
task_t **child_task*)

ARGUMENTS

parent_task: The task from which the child's capabilities are drawn.

inherit_memory: If set, the child task's address space is built from the parent task according to its memory inheritance values; otherwise, the child task is given an empty address space.

child_task: Returns the new task.

DESCRIPTION

Important: Normally, you should use the UNIX **fork()** system call instead of **task_create**().

The function **task_create()** creates a new task from *parent_task*; the resulting task (*child_task*) acquires shared or copied parts of the parent's address space (see **vm_inherit**()). The child task initially has no threads; you put threads in it using **thread_create**().

The child task gets the four special ports initialized for it at task creation. The kernel port (task port) is created and send rights for it are given to the child and returned to the caller in *child_task*. The notify port is initialized to null. The child inherits its bootstrap and exception ports from the parent task. The new task can get send rights to these ports with the call **task_get_special_port**().

EXAMPLE

```
error=task_create(task_self(), TRUE, &child_task);
if(error!=KERN_SUCCESS)
    mach_error("Call to task_create() failed", error);
```

RETURN

KERN_SUCCESS: A new task has been created.

KERN_INVALID_ARGUMENT: *parent_task* is not a valid task port.

KERN_RESOURCE_SHORTAGE: Some critical kernel resource is unavailable.

SEE ALSO

task_terminate(), **task_suspend**(), **task_resume**(), **task_get_special_port**(), **task_set_special_port**(), **task_self**(), **task_threads**(), **thread_create**(), **thread_resume**(), **vm_inherit**()

task_get_assignment()

SUMMARY Get the name of the processor set that a task is assigned to

SYNOPSIS

#include <mach.h>

kern_return_t **task_get_assignment**(task_t *task*, processor_set_t **processor_set*)

Note: This function is useful only on multiprocessor systems.

task_get_special_port(), task_set_special_port(), task_self(), task_notify()

SUMMARY Access a task's special ports

SYNOPSIS

#include <mach.h>

kern_return_t **task_get_special_port**(task_t *task*, int *which_port*,
port_t **special_port*)
kern_return_t **task_set_special_port**(task_t *task*, int *which_port*, port_t *special_port*)
task_t **task_self()**
port_t **task_notify**()

ARGUMENTS

task: The task for which to get the port.

which_port: The port that's requested. This is one of:

TASK_NOTIFY_PORT
TASK_BOOTSTRAP_PORT
TASK_EXCEPTION_PORT

special_port: The value of the port that's being requested or set.

DESCRIPTION

The function **task_get_special_port**() returns send rights to one of a set of special ports for the task specified by *task*. In the case of the task's own notify port, the task also gets receive rights.

The function **task_set_special_port**() sets one of a set of special ports for the task specified by *task*.

The function **task_self()** returns the port to which kernel calls for the currently executing thread should be directed. Currently **task_self()** returns the task kernel port, which is a port for which the kernel has receive rights and which it uses to identify a task. In the future it may be possible for one task to interpose a port as another task's kernel port. At that time **task_self()** will still return the port to which the executing thread should direct kernel calls, but it may no longer be a port for which the kernel has receive rights.

If a controller task has send access to the kernel port of a subject task, then the controller task can perform kernel operations for the subject task. Normally only the task itself and the task that created it will have access to the task kernel port, but any task may pass rights to its kernel port to any other task.

The function **task_notify**() returns receive and send rights to the notify port associated with the task to which the executing thread belongs. The notify port is a port on which the task should receive notification of such kernel events as the destruction of a port to which it has send rights.

The other special ports associated with a task are the bootstrap port and the exception port. The bootstrap port is a port to which a thread may send a message requesting other system service ports. This port isn't used by the kernel. The task's exception port is the port to which messages are sent by the kernel, when an exception occurs and the thread causing the exception has no exception port of its own.

Important: If you set your task's bootstrap port, you should also set the global variable **bootstrap_port** to *special_port*. **bootstrap_port** is a task-wide variable that's used by **mach_init** and other processes to determine your task's bootstrap port. Since you can't change the **bootstrap_port** variable's value in another task, you should use care when changing the bootstrap port of another task.

MACRO EQUIVALENTS

The following macros are defined in the header file **sys/task_special_ports.h**:

task_get_notify_port(*task*, *port*)
task_set_notify_port(*task*, *port*)

task_get_exception_port(*task*, *port*)
task_set_exception_port(*task*, *port*)

task_get_bootstrap_port(*task*, *port*)
task_set_bootstrap_port(*task*, *port*)

EXAMPLE

```
/* 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);
}
```

RETURN

KERN_SUCCESS: The port was returned or set.

KERN_INVALID_ARGUMENT: Either *task* is not a task or *which_port* is an invalid port selector.

SEE ALSO

thread_special_ports(), task_create()

task_info()

SUMMARY Get information about a task

SYNOPSIS

#include <mach.h>

kern_return_t **task_info(**task_t *target_task*, int *flavor*, task_info_t *task_info*, unsigned int **task_info_count***)**

ARGUMENTS

target_task: The task to be affected (for example, use **task_self()** to specify the caller's task).

flavor: The type of statistics that are wanted. Currently only TASK_BASIC_INFO is implemented.

task_info: Returns statistics about *target_task*.

task_info_count: Size of the info structure. Currently this must be TASK_BASIC_INFO_COUNT.

DESCRIPTION

The function **task_info()** returns the information specified by *flavor* about a task. *task_info* is an array of integers that's supplied by the caller and returned filled with information. *task_info_count* is supplied as the maximum number of integers in *task_info*. On return, it contains the actual number of integers in *task_info*.

Currently there's only one flavor of information, defined by TASK_BASIC_INFO. Its size is defined by TASK_BASIC_INFO_COUNT. The definition of the information structure returned by TASK_BASIC_INFO is:

```
struct task_basic_info {
    int           suspend_count;   /* suspend count for task */
    int           base_priority;   /* base scheduling priority */
    vm_size_t     virtual_size;    /* number of virtual pages */
    vm_size_t     resident_size;   /* number of resident pages */
    time_value_t  user_time;       /* total user run time for
                                      terminated threads */
    time_value_t  system_time;     /* total system run time for
                                      terminated threads */
};
typedef struct task_basic_info  *task_basic_info_t;
```

EXAMPLE

```
kern_return_t            error;
struct task_basic_info   info;
unsigned int             info_count=TASK_BASIC_INFO_COUNT;

error=task_info(task_self(), TASK_BASIC_INFO,
    (task_info_t)&info, &info_count);
if (error!=KERN_SUCCESS)
    mach_error("Error calling task_info()", error);
else
    printf("Base priority is %d\n", info.base_priority);
```

RETURN

KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *target_task* isn't a task or *flavor* isn't recognized.

MIG_ARRAY_TOO_LARGE: The returned info array is too large for *task_info*. *task_info* is filled as much as possible, and *task_info_count* is set to the number of elements that would be returned if there were enough room.

SEE ALSO

task_threads(), thread_info(), thread_state()

task_notify() → See task_get_special_port()

task_priority()

SUMMARY Set the scheduling priority for a task

SYNOPSIS

#include <mach.h>

kern_return_t **task_priority**(task_t *task*, int *priority*, boolean_t *change_threads*)

ARGUMENTS

task: Task to set priority for.

priority: New priority.

change_threads: Change priority of existing threads if true.

DESCRIPTION

The priority of a task is used only for creation of new threads; a new thread's priority is set to its task's priority. **task_priority**() changes this task priority. It also sets the priorities of all threads in the task to this new priority if *change_threads* is true. Existing threads are not affected otherwise. If this priority change violates the maximum priority of some threads, as many threads as possible will be changed and an error code will be returned.

Priorities range from 0 to 31, where higher numbers denote higher priorities. The maximum user priority is defined in the header file **kern/sched.h** as MAXPRI_USER. You can retrieve the current scheduling priority using **thread_info**().

EXAMPLE

```
kern_return_t            error;
struct task_basic_info   info;
unsigned int             info_count=TASK_BASIC_INFO_COUNT;

error=task_info(task_self(), TASK_BASIC_INFO,
    (task_info_t)&info, &info_count);
if (error!=KERN_SUCCESS)
    mach_error("Error calling task_info()", error);
else {
    /* Set this task's base priority to be much lower than normal */
    error = task_priority(task_self(), info.base_priority - 4, TRUE);
    if (error != KERN_SUCCESS)
        mach_error("Call to task_priority() failed", error);
}
```

RETURN

KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *task* is not a task or *priority* is not a valid priority.

KERN_FAILURE: *change_threads* was true and the attempt to change the priority of at least one existing thread failed because the new priority would have exceeded that thread's maximum priority.

SEE ALSO

thread_priority(), **processor_set_max_priority**(), **thread_switch**()

task_resume()

SUMMARY Resume a task

SYNOPSIS

#include <mach.h>

kern_return_t **task_resume**(task_t *target_task*)

ARGUMENTS

target_task: The task to be resumed.

DESCRIPTION

The function **task_resume**() decrements the task's suspend count. If the suspend count becomes 0, all threads with 0 suspend counts in the task are resumed. If the suspend count is already 0, it's not decremented (it never becomes negative).

RETURN

KERN_SUCCESS: The task has been resumed.

KERN_FAILURE: The suspend count is already 0.

KERN_INVALID_ARGUMENT: *target_task* isn't a task.

SEE ALSO

task_create(), **task_terminate**(), **task_suspend**(), **task_info**(), **thread_suspend**(), **thread_resume**(), **thread_info**()

task_self() → See task_get_special_port()

task_set_special_port() → See task_get_special_port()

task_suspend()

SUMMARY Suspend a task

SYNOPSIS

#include <mach.h>

kern_return_t **task_suspend**(task_t *target_task*)

ARGUMENTS

target_task: The task to be suspended (for example, use **task_self**() to specify the caller's task).

DESCRIPTION

The function **task_suspend**() increments the task's suspend count and stops all threads in the task. As long as the suspend count is positive, newly created threads will not run. This call doesn't return until all threads are suspended.

If the count becomes greater than 1, it will take more than one **task_resume**() call to restart the task.

RETURN

KERN_SUCCESS: The task has been suspended.

KERN_INVALID_ARGUMENT: *target_task* isn't a task.

SEE ALSO

task_create(), **task_terminate**(), **task_resume**(), **task_info**(), **thread_suspend**()

task_terminate()

SUMMARY Terminate a task

SYNOPSIS

#include <mach.h>

kern_return_t **task_terminate**(task_t *target_task*)

ARGUMENTS

target_task: The task to be destroyed (for example, use **task_self**() to specify the caller's task).

DESCRIPTION

The function **task_terminate**() destroys the task specified by *target_task* and all its threads. All resources that are used only by this task are freed. Any port to which this task has receive rights is destroyed.

RETURN

KERN_SUCCESS: The task has been destroyed.

KERN_INVALID_ARGUMENT: *target_task* isn't a task.

SEE ALSO

task_create(), **task_suspend**(), **task_resume**(), **thread_terminate**(), **thread_suspend**()

task_threads()

SUMMARY Get a task's threads

SYNOPSIS

#include <mach.h>

kern_return_t **task_threads**(task_t *target_task*, thread_array_t **thread_list*, unsigned int **thread_count*)

ARGUMENTS

target_task: The task to be affected (for example, use **task_self**() to specify the caller's task).

thread_list: Returns the set of threads contained within *target_task*; no particular ordering is guaranteed.

thread_count: Returns the number of threads in *thread_list*.

DESCRIPTION

The function **task_threads**() gets send rights to the kernel port for each thread contained in *target_task*.

The array *thread_list* is created as a result of this call. You should call **vm_deallocate**() on this array when the data is no longer needed.

EXAMPLE

```
r = task_threads(task_self(), &thread_list, &thread_count);
if (r != KERN_SUCCESS)
    mach_error("Error calling task_threads", r);
else {
    if (thread_count == 1)
        printf ("There's 1 thread in this task\n");
    else
        printf("There are %d threads in this task\n", thread_count);

/* Deallocate the list of threads. */
    r = vm_deallocate(task_self(), (vm_address_t)thread_list,
        sizeof(thread_list)*thread_count);
    if (r != KERN_SUCCESS)
        mach_error("Trouble freeing thread_list", r);
}
```

RETURN

KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *target_task* isn't a task.

SEE ALSO

thread_create(), **thread_terminate()**, **thread_suspend()**

thread_abort()

SUMMARY Interrupt a thread

SYNOPSIS

#include <mach.h>

kern_return_t **thread_abort**(thread_t *target_thread*)

ARGUMENTS

target_thread: The thread to be interrupted.

DESCRIPTION

The function **thread_abort**() aborts the kernel functions **msg_send()**, **msg_receive()**, and **msg_rpc()** and page faults, making the call return a code indicating that it was interrupted. The call is interrupted whether or not the thread (or task containing it) is currently suspended. If it's suspended, the thread receives the interrupt when it resumes.

A thread will retry an aborted page fault if its state isn't modified before it resumes. The function **msg_send()** returns SEND_INTERRUPTED; **msg_receive()** returns RCV_INTERRUPTED; **msg_rpc()** returns either SEND_INTERRUPTED or RCV_INTERRUPTED, depending on which half of the RPC was interrupted.

This function lets one thread stop another thread cleanly, thereby allowing the future execution of the target thread to be controlled in a predictable way. **thread_suspend()** keeps the target thread from executing any further instructions at the user level, including the return from a system call. **thread_get_state()** and **thread_set_state()** let you examine or modify the user state of a target thread. However, if a suspended thread was executing within a system call, it also has associated with it a kernel state. This kernel state can't be modified by **thread_set_state()**; therefore, when the thread is resumed the system call may return, changing the user state and possibly user memory.

thread_abort() aborts the kernel call from the target thread's point of view by resetting the kernel state so that the thread will resume execution just after the system call. The system call will return one of the interrupted codes described above. The system call will either be entirely completed or entirely aborted, depending on the precise moment at which the abort was received. Thus if the thread's user state has been changed by **thread_set_state()**, it won't be modified by any unexpected system call side effects.

For example, to simulate a UNIX signal, the following sequence of calls may be used:

1. **thread_suspend()**: Stops the thread.
2. **thread_abort()**: Interrupts any system call in progress, setting the return value to "interrupted". Since the thread is stopped, it won't return to user code.
3. **thread_set_state()**: Alters the thread's state to simulate a procedure call to the signal handler.
4. **thread_resume()**: Resumes execution at the signal handler. If the thread's stack has been correctly set up, the thread can return to the interrupted system call.

Calling **thread_abort()** on a thread that's not suspended is risky, since it's difficult to know exactly what system trap, if any, the thread might be executing and whether an interrupt return would cause the thread to do something useful.

RETURN

KERN_SUCCESS: The thread received an interrupt.

KERN_INVALID_ARGUMENT: *target_thread* isn't a thread.

SEE ALSO

thread_info(), thread_state(), thread_terminate(), thread_suspend()

thread_assign(), thread_assign_default()

SUMMARY Assign a thread to a processor set

SYNOPSIS

#include <mach.h>

kern_return_t **thread_assign**(thread_t *thread*, processor_set_t *new_processor_set*)
kern_return_t **thread_assign_default**(thread_t *thread*)

DESCRIPTION

thread_assign() assigns *thread* to *new_processor_set*; **thread_assign_default**() assigns *thread* to the default processor set.

Note: These functions are useful only on multiprocessor systems.

thread_create()

SUMMARY Create a thread

SYNOPSIS

#include <mach.h>

kern_return_t **thread_create**(task_t *parent_task*, thread_t **child_thread*)

ARGUMENTS

parent_task: The task that's to contain the new thread.

child_thread: Returns the new thread.

DESCRIPTION

Important: Don't use this function unless you're writing a loadable kernel server or implementing a new thread package, such as the C thread functions. For normal, user-level programming, use **cthread_fork**() instead. You can then use **cthread_thread**() if you need to get the Mach thread that corresponds to the new C thread.

The function **thread_create**() creates a new thread within *parent_task.* The new thread has no processor state, and has a suspend count of 1. To get a new thread to run, first call **thread_create**() to get the new thread's identifier, *child_thread.* Then call **thread_set_state**() to set a processor state. Finally, call **thread_resume()** to schedule the thread to execute.

When the thread is created, send rights to its thread kernel port are given to it and returned to the caller in *child_thread.* The new thread's exception port is set to PORT_NULL.

RETURN

KERN_SUCCESS: A new thread has been created.

KERN_INVALID_ARGUMENT: *parent_task* isn't a valid task.

KERN_RESOURCE_SHORTAGE: Some critical kernel resource isn't available.

SEE ALSO

task_create(), **task_threads**(), **thread_terminate**(), **thread_suspend**(), **thread_resume**(), **thread_special_ports**(), **thread_set_state**()

thread_get_assignment()

SUMMARY Get the name of the processor set to which a thread is assigned

SYNOPSIS

#include <mach.h>

kern_return_t **thread_get_assignment**(thread_t *thread*, processor_set_t **processor_set*)

Note: This function is useful only in multiprocessor systems.

thread_get_special_port(), thread_set_special_port(), thread_self(), thread_reply()

SUMMARY Access a thread's special ports

SYNOPSIS

#include <mach.h>

kern_return_t **thread_get_special_port**(thread_t *thread*, int *which_port*, port_t **special_port*)
kern_return_t **thread_set_special_port**(thread_t *thread*, int *which_port*, port_t *special_port*)
thread_t **thread_self**()
port_t **thread_reply**()

ARGUMENTS

thread: The thread for which to get the port.

which_port: The port that's requested. This is one of:

THREAD_REPLY_PORT
THREAD_EXCEPTION_PORT

special_port: The value of the port that's being requested or set.

DESCRIPTION

The function **thread_get_special_port**() returns send rights to one of a set of special ports for the thread specified by *thread.* In the case of getting the thread's own reply port, receive rights are also given to the thread.

The function **thread_set_special_port**() sets one of a set of special ports for the thread specified by *thread.*

The function **thread_self**() returns the port to which kernel calls for the currently executing thread should be directed. Currently **thread_self**() returns the thread kernel port, which is a port for which the kernel has receive rights and which it uses to identify a thread. In the future it may be possible for one thread to interpose a port as another thread's kernel port. At that time **thread_self**() will still return the port to which the executing thread should direct kernel calls, but it may no longer be a port for which the kernel has receive rights.

If a controller thread has send access to the kernel port of a subject thread, the controller thread can perform kernel operations for the subject thread. Normally only the thread itself and its parent task will have access to the thread kernel port, but any thread may pass rights to its kernel port to any other thread.

The function **thread_reply**() returns receive and send rights to the reply port of the calling thread. The reply port is a port to which the thread has receive rights. It's used to receive any initialization messages and as a reply port for early remote procedure calls.

A thread also has access to its task's special ports.

MACRO EQUIVALENTS

The following macros are defined in the header file **sys/thread_special_ports.h**:

thread_get_reply_port(*thread*, *port*)
thread_set_reply_port(*thread*, *port*)

thread_get_exception_port(*thread*, *port*)
thread_set_exception_port(*thread*, *port*)

RETURN

KERN_SUCCESS: The port was returned or set.

KERN_INVALID_ARGUMENT: *thread* isn't a thread or *which_port* is an invalid port selector.

SEE ALSO

task_get_special_port(), **task_set_special_port()**, **task_self()**, **thread_create()**

thread_get_state(), thread_set_state()

SUMMARY Access a thread's state

SYNOPSIS

#include <mach.h>

kern_return_t **thread_get_state**(thread_t *target_thread*, int *flavor*, thread_state_data_t *old_state*, unsigned int **old_state_count*)
kern_return_t **thread_set_state**(thread_t *target_thread*, int *flavor*, thread_state_data_t *new_state*, unsigned int *new_state_count*)

ARGUMENTS

target_thread: Thread to get or set the state for.

flavor: The type of state that's to be manipulated. This may be any one of the following:

NeXT_THREAD_STATE_REGS
NeXT_THREAD_STATE_68882
NeXT_THREAD_STATE_USER_REG

old_state: Returns an array of state information.

new_state: An array of state information.

old_state_count: The size of the state information array. This may be any one of the following:

NeXT_THREAD_STATE_REGS_COUNT
NeXT_THREAD_STATE_68882_COUNT
NeXT_THREAD_STATE_USER_REG_COUNT

new_state_count: Same as *old_state_count*.

DESCRIPTION

The function **thread_get_state()** returns the state component (that is, the machine registers) of *target_thread* as specified by *flavor*. The *old_state* is an array of integers that's provided by the caller and returned filled with the specified information. You should set *old_state_count* to the maximum number of integers in *old_state*. On return, *old_state_count* is equal to the actual number of integers in *old_state*.

The function **thread_set_state()** sets the state component of *target_thread* as specified by *flavor.* The *new_state* is an array of integers that the caller fills. You should set *new_state_count* to the number of elements in *new_state*. The entire set of registers is reset.

target_thread must not be **thread_self()** for either of these calls.

The state structures are defined in the header file **machine/thread_status.h**.

RETURN

KERN_SUCCESS: The state has been set or returned.

MIG_ARRAY_TOO_LARGE: The returned state is too large for the *new_state*. *new_state* is filled in as much as possible and *new_state_count* is set to the number of elements that would be returned if there were enough room.

KERN_INVALID_ARGUMENT: Either *target_thread* isn't a thread, *target_thread* is **thread_self()**, or *flavor* is unrecognized for this machine.

SEE ALSO

task_info(), thread_info()

thread_info()

SUMMARY Get information about a thread

SYNOPSIS

#include <mach.h>

kern_return_t **thread_info**(thread_t *target_thread*, int *flavor*,
thread_info_t *thread_info*, unsigned int **thread_info_count*)

ARGUMENTS

target_thread: The thread to be affected.

flavor: The type of statistics wanted. This can be THREAD_BASIC_INFO or THREAD_SCHED_INFO.

thread_info: Returns statistics about *target_thread*.

thread_info_count: Size of the info structure. This can be THREAD_BASIC_INFO_COUNT or THREAD_SCHED_INFO_COUNT.

DESCRIPTION

The function **thread_info()** returns the selected information array for a thread, as specified by *flavor. thread_info* is an array of integers that's supplied by the caller and returned filled with specified information. *thread_info_count* is supplied as the maximum number of integers in *thread_info*. On return, it contains the actual number of integers in *thread_info*.

The size of the information returned by THREAD_BASIC_INFO is defined by THREAD_BASIC_INFO_COUNT. The definition of the information structure returned by THREAD_BASIC_INFO is:

```
struct thread_basic_info {
    time_value_t  user_time;      /* user run time */
    time_value_t  system_time;    /* system run time */
    int           cpu_usage;      /* scaled cpu usage percentage */
    int           base_priority;  /* base scheduling priority */
    int           cur_priority;   /* current scheduling priority */
    int           run_state;      /* run state */
    int           flags;          /* various flags */
    int           suspend_count;  /* suspend count for thread */
    long          sleep_time;     /* number of seconds that thread
                                     has been sleeping */
};
typedef struct thread_basic_info *thread_basic_info_t;
```

The **run_state** field has one of the following values:

TH_STATE_RUNNING: The thread is running normally.

TH_STATE_STOPPED: The thread is suspended. This happens when the thread or task suspend count is greater than zero.

TH_STATE_WAITING: The thread is sleeping normally.

TH_STATE_UNINTERRUPTIBLE: The thread is in an uninterruptible sleep. This should happen only for very short times during some system calls.

TH_STATE_HALTED: The thread is halted at a clean point. This state happens only after a call to **thread_abort()**.

Possible values of the **flags** field are:

TH_FLAGS_SWAPPED: The thread is swapped out. This happens when the thread hasn't run in a long time, and the kernel stack for the thread has been swapped out.

TH_FLAGS_IDLE: The thread is the idle thread for the CPU. This means that the CPU runs this thread whenever it has nothing else to do.

The **sleep_time** field is useful only when **run_state** is TH_STATE_STOPPED. (Currently **sleep_time** is always set to zero, no matter how long the thread has been sleeping.)

The size of the information returned by THREAD_SCHED_INFO is defined by THREAD_SCHED_INFO_COUNT. The definition of the information structure returned by THREAD_SCHED_INFO is:

```
struct thread_sched_info {
    int        policy;           /* scheduling policy */
    int        data;             /* associated data */
    int        base_priority;    /* base priority */
    int        max_priority;     /* max priority */
    int        cur_priority;     /* current priority */
    boolean_t  depressed;        /* depressed ? */
    int        depress_priority; /* priority depressed from */
};
typedef struct thread_sched_info    *thread_sched_info_t;
```

The **policy** field has one of the following values: POLICY_FIXEDPRI, POLICY_TIMESHARE, or POLICY_INTERACTIVE. If **policy** is POLICY_FIXEDPRI, then **data** is the quantum (in milliseconds). Otherwise, **data** is meaningless.

EXAMPLE

Example of using THREAD_BASIC_INFO:

```
kern_return_t             error;
struct thread_basic_info  info;
unsigned int              info_count=THREAD_BASIC_INFO_COUNT;

error=thread_info(thread_self(), THREAD_BASIC_INFO,
    (thread_info_t)&info, &info_count);
if (error!=KERN_SUCCESS)
    mach_error("Error calling thread_info()", error);
else {
    printf("User time is %d seconds, %d microseconds\n",
        info.user_time.seconds, info.user_time.microseconds);
    printf("System time is %d seconds, %d microseconds\n",
        info.system_time.seconds, info.system_time.microseconds);
}
```

Example of using THREAD_SCHED_INFO:

```
kern_return_t             error;
struct thread_sched_info  info;
unsigned int              info_count=THREAD_SCHED_INFO_COUNT;

error=thread_info(thread_self(), THREAD_SCHED_INFO,
    (thread_info_t)&info, &info_count);
if (error!=KERN_SUCCESS)
    mach_error("Error calling thread_info()", error);
else {
    printf("Base priority is %d\n", info.base_priority);
    printf("Max priority is %d\n", info.max_priority);
}
```

RETURN

KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *target_thread* isn't a thread, or *flavor* isn't recognized, or *thread_info_count* is smaller than it's supposed to be.

MIG_ARRAY_TOO_LARGE: The returned info array is too large for *thread_info*. *thread_info* is filled as much as possible. *thread_info_count* is set to the number of elements that would have been returned if there were enough room.

SEE ALSO

thread_get_special_port(), **task_threads()**, **task_info()**, **thread_state()**

thread_max_priority() → See thread_priority()

thread_policy()

SUMMARY Set scheduling policy for a thread

SYNOPSIS

#include <mach.h>

kern_return_t **thread_policy**(thread_t *thread*, int *policy*, int *data*)

ARGUMENTS

thread: Thread to set policy for.

policy: Policy to set. This must be POLICY_TIMESHARE, POLICY_INTERACTIVE, or POLICY_FIXEDPRI.

data: Policy-specific data.

DESCRIPTION

thread_policy() changes the scheduling policy for *thread* to *policy.*

data is meaningless for the timesharing and interactive policies; for the fixed priority policy, it's the quantum to be used (in milliseconds). The system will always round the quantum up to the next multiple of the basic system quantum (**min_quantum**, which can be obtained from **host_info**()). You can find the current quantum using **thread_info**().

Processor sets can restrict the allowed policies, so this call will fail if the processor set to which *thread* is currently assigned doesn't permit *policy.*

EXAMPLE

```
kern_return_t           error;
struct host_sched_info  sched_info;
unsigned int            sched_count=HOST_SCHED_INFO_COUNT;
int                     quantum;
processor_set_t         default_set, default_set_priv;

/* Set quantum to a reasonable value. */
error=host_info(host_self(), HOST_SCHED_INFO,
    (host_info_t)&sched_info, &sched_count);
if (error != KERN_SUCCESS) {
    mach_error("SCHED host_info() call failed", error);
    exit(1);
}
else
    quantum = sched_info.min_quantum;

/*
 * Fix the default processor set to take a fixed priority thread.
 */
error=processor_set_default(host_self(), &default_set);
if (error!=KERN_SUCCESS) {
    mach_error("Error calling processor_set_default()", error);
    exit(1);
}

error=host_processor_set_priv(host_priv_self(), default_set,
    &default_set_priv);
if (error != KERN_SUCCESS) {
    mach_error("Call to host_processor_set_priv() failed", error);
    exit(1);
}

error=processor_set_policy_enable(default_set_priv, POLICY_FIXEDPRI);
if (error != KERN_SUCCESS)
    mach_error("Error calling processor_set_policy_enable", error);
```

```
/*
 * Change the thread's scheduling policy to fixed priority.
 */
error=thread_policy(thread_self(), POLICY_FIXEDPRI, quantum);
if (error != KERN_SUCCESS)
    mach_error("thread_policy() call failed", error);
```

RETURN

KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *thread* is not a thread, or *policy* is not a recognized policy.

KERN_FAILURE: The processor set to which *thread* is currently assigned doesn't permit *policy.*

SEE ALSO

processor_set_policy(), **thread_switch()**

thread_priority(), thread_max_priority()

SUMMARY Set scheduling priority for thread

SYNOPSIS

#include <mach.h>

kern_return_t **thread_priority**(thread_t *thread*, int *priority*, boolean_t *set_max*)
kern_return_t **thread_max_priority**(thread_t *thread*, processor_set_t *processor_set*, int *priority*)

ARGUMENTS

thread: The thread whose priority is to be changed.

priority: The new priority to change it to.

set_max: Also set *thread*'s maximum priority if true.

processor_set: The privileged port for the processor set to which *thread* is currently assigned.

DESCRIPTION

Threads have three priorities associated with them by the system: a *base priority*, a *maximum priority*, and a *scheduled priority.*

The scheduled priority is used to make scheduling decisions about the thread. It is determined from the base priority by the policy. (For the timesharing and interactive policies, this means adding an increment derived from CPU usage). The base priority can be set under user control, but can never exceed the maximum priority. Raising the maximum priority requires presentation of the privileged port for the thread's processor set; since the privileged port for the default processor set is available only to the superuser, users cannot raise their maximum priority to unfairly compete with other users on that set. Newly created threads obtain their base priority from the task and their maximum priority from the thread.

Priorities range from 0 to 31, where higher numbers denote higher priorities. The maximum user priority is defined in the header file **kern/sched.h** as MAXPRI_USER. You can obtain the base, scheduled, and maximum priorities using **thread_info**().

thread_priority() changes the base priority and optionally the maximum priority of *thread*. If the new base priority is higher than the scheduled priority of the currently executing thread, preemption may occur as a result of this call. The maximum priority of the thread is also set if *set_max* is true. This call fails if *priority* is greater than the current maximum priority of the thread. As a result, **thread_priority**() can never raise —only lower—the value of a thread's maximum priority.

thread_max_priority() changes the maximum priority of the thread. Because it requires the privileged port for the processor set, this call can reset the maximum priority to any legal value. If the new maximum priority is less than the thread's base priority, then the thread's base priority is set to the new maximum priority.

EXAMPLE

```
/* Get the privileged port for the default processor set. */
error=processor_set_default(host_self(), &default_set);
if (error!=KERN_SUCCESS) {
    mach_error("Error calling processor_set_default()", error);
    exit(1);
}

error=host_processor_set_priv(host_priv_self(), default_set,
    &default_set_priv);
if (error!=KERN_SUCCESS) {
    mach_error("Call to host_processor_set_priv() failed", error);
    exit(1);
}

/* Set the max priority. */
error=thread_max_priority(thread_self(), default_set_priv, priority);
if (error!=KERN_SUCCESS)
    mach_error("Call to thread_max_priority() failed",error);

/* Set the thread's priority. */
error=thread_priority(thread_self(), priority, FALSE);
if (error!=KERN_SUCCESS)
    mach_error("Call to thread_priority() failed",error);
```

RETURN

KERN_SUCCESS: Operation completed successfully

KERN_INVALID_ARGUMENT: *thread* is not a thread, *processor_set* is not a privileged port for a processor set, or *priority* is out of range (not in 0-31).

KERN_FAILURE: The requested operation would violate the thread's maximum (only for **thread_priority**()) or the thread is not assigned to the processor set whose privileged port was presented.

SEE ALSO

thread_policy(), **task_priority**(), **processor_set_priority**(), **thread_switch**()

thread_reply() → See thread_get_special_port()

thread_resume()

SUMMARY Resume a thread

SYNOPSIS

#include <mach.h>

kern_return_t **thread_resume**(thread_t *target_thread*)

ARGUMENTS

target_thread: The thread to be resumed.

DESCRIPTION

The function **thread_resume**() decrements the thread's suspend count. If the count becomes 0, the thread is resumed. If it's still positive, the thread is left suspended. The suspend count never becomes negative.

RETURN

KERN_SUCCESS: The thread has been resumed.

KERN_FAILURE: The suspend count is already 0.

KERN_INVALID_ARGUMENT: *target_thread* isn't a thread.

SEE ALSO

task_suspend(), **task_resume**(), **thread_info**(), **thread_create**(), **thread_terminate**(), **thread_suspend**()

thread_self() → See thread_get_special_port()

thread_set_special_port() → See thread_get_special_port()

thread_set_state() → See thread_get_state()

thread_suspend()

SUMMARY Suspend a thread

SYNOPSIS

#include <mach.h>

kern_return_t **thread_suspend**(thread_t *target_thread*)

ARGUMENTS

target_thread: The thread to be suspended.

DESCRIPTION

The function **thread_suspend**() increments the thread's suspend count and prevents the thread from executing any more user-level instructions. In this context, a user-level instruction is either a machine instruction executed in user mode or a system trap instruction (including page faults).

Thus, if a thread is currently executing within a system trap, the kernel code may continue to execute until it reaches the system return code or it may suspend within the kernel code. In either case, when the thread is resumed the system trap will return. This could cause unpredictable results if you did a suspend and then altered the user state of the thread in order to change its direction upon a resume. The function **thread_abort**() lets you abort any currently executing system call in a predictable way.

If the suspend count becomes greater than 1, it will take more than one **thread_resume**() call to restart the thread.

RETURN

KERN_SUCCESS: The thread has been suspended.

KERN_INVALID_ARGUMENT: *target_thread* isn't a thread.

SEE ALSO

task_suspend(), **task_resume**(), **thread_info**(), **thread_state**(), **thread_resume**(), **thread_terminate**(), **thread_abort**()

thread_switch()

SUMMARY Cause context switch with options

SYNOPSIS

#include <mach.h>

kern_return_t **thread_switch**(thread_t *new_thread*, int *option*, int *time*)

ARGUMENTS

new_thread: Thread to context switch to. If you specify THREAD_NULL, be sure to specify the *option* parameter to be either SWITCH_OPTION_WAIT or SWITCH_OPTION_DEPRESS.

option: Specifies options associated with context switch. Three options are recognized:

SWITCH_OPTION_NONE: No options, the *time* argument is ignored. (You must set *new_thread* to a valid thread.)

SWITCH_OPTION_WAIT: This thread is blocked for the specified time. The block can be aborted by **thread_abort**().

SWITCH_OPTION_DEPRESS: This thread's priority is depressed to the lowest possible value until one of the following happens: *time* milliseconds pass, this thread is scheduled again, or **thread_abort**() is called on this thread (whichever happens first). Priority depression is independent of operations that change this thread's priority; for example, **thread_priority**() will not abort the depression.

time: Time duration (in milliseconds) for options. The minimum time can be obtained as the **min_timeout** value from **host_info**().

DESCRIPTION

thread_switch() provides low-level access to the scheduler's context switching code. *new_thread* is a hint that implements handoff scheduling. The operating system will attempt to switch directly to *new_thread* (bypassing the normal logic that selects the next thread to run) if possible. If *new_thread* isn't valid or THREAD_NULL, **thread_switch**() returns an error.

thread_switch() is often called when the current thread can proceed no further for some reason; the various options and arguments allow information about this reason to be transmitted to the kernel. The *new_thread* argument (handoff scheduling) is useful when the identity of the thread that must make progress before the current thread runs again is known. The SWITCH_OPTION_WAIT option is used when the amount of time that the current thread must wait before it can do anything useful can be estimated and is fairly long. The SWITCH_OPTION_DEPRESS option is used when the amount of time that must be waited is fairly short, especially when the identity of the thread that is being waited for is not known.

Users should beware of calling **thread_switch**() with an invalid *new_thread* (for example, THREAD_NULL) and no *option*. Because the time-sharing and interactive schedulers vary the priority of threads based on usage, this may result in a waste of CPU time if the thread that must be run is of lower priority. The use of the SWITCH_OPTION_DEPRESS option in this situation is highly recommended.

When a thread that's depressed is scheduled, it regains its old priority. The code should recheck the conditions to see if it wants to depress again. If **thread_abort**() is called on a depressed thread, the thread's priority is restored.

Users relying on the preemption semantics of a fixed time policy should be aware that **thread_switch**() ignores these semantics; it will run the specified *new_thread* independent of its priority and the priority of any other threads that could be run instead.

RETURN

KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: *new_thread* is not a thread, or *option* is not a recognized option.

thread_terminate()

SUMMARY Terminate a thread

SYNOPSIS

#include <mach.h>

kern_return_t **thread_terminate**(thread_t *target_thread*)

ARGUMENTS

target_thread: The thread to be destroyed.

DESCRIPTION

The function **thread_terminate**() destroys the thread specified by *target_thread.*

Warning: Don't use this function on threads that were created using the C thread functions.

RETURN

KERN_SUCCESS: The thread has been destroyed.

KERN_INVALID_ARGUMENT: *target_thread* isn't a thread.

SEE ALSO

task_terminate(), **task_threads()**, **thread_create()**, **thread_resume()**, **thread_suspend()**

unix_pid()

SUMMARY Get the process ID of a task

SYNOPSIS

#include <mach.h>

kern_return_t **unix_pid(**task_t *target_task*, int **pid*)

ARGUMENTS

target_task: The task for which you want the process ID.

pid: Returns the process ID of *target_task*.

DESCRIPTION

Returns the process ID of *target_task*. If the call doesn't succeed, *pid* is set to -1.

EXAMPLE

```
result=unix_pid(task_self(), &my_pid);
if (result!=KERN_SUCCESS) {
    mach_error("Call to unix_pid failed", result);
    exit(1);
}

printf("My process ID is %d\n", my_pid);
```

RETURN

KERN_SUCCESS: The call succeeded.

KERN_FAILURE: *target_task* isn't a valid task. This might be because *target_task* is a pure Mach task (one created using **task_create()**).

SEE ALSO

task_by_unix_pid()

vm_allocate()

SUMMARY Allocate virtual memory

SYNOPSIS

#include <mach.h>

kern_return_t **vm_allocate**(vm_task_t *target_task*, vm_address_t **address*,
vm_size_t *size*, boolean_t *anywhere*)

ARGUMENTS

target_task: Task whose virtual memory is to be affected. Use **task_self()** to allocate memory in the caller's address space.

address: Starting address. If *anywhere* is true, the input value of this address will be ignored, and the space will be allocated wherever it's available. If *anywhere* is false, an attempt is made to allocate virtual memory starting at this virtual address. If this address isn't at the beginning of a virtual page, it gets rounded down so that it is. If there isn't enough space at this address, no memory will be allocated. No matter what the value of *anywhere* is, the address at which memory is actually allocated is returned in *address*.

size: Number of bytes to allocate (rounded up by the system to an integral number of virtual pages).

anywhere: If true, the kernel should find and allocate any region of the specified size. If false, virtual memory is allocated starting at *address* (rounded down to a virtual page boundary) if there's sufficient space.

DESCRIPTION

The function **vm_allocate()** allocates a region of virtual memory, placing it in the specified task's address space. The physical memory isn't actually allocated until the new virtual memory is referenced. By default, the kernel rounds all addresses down to the nearest page boundary and all memory sizes up to the nearest page size. The global variable **vm_page_size** contains the page size. For languages other than C, the value of **vm_page_size** can be obtained by calling **vm_statistics()**.

Initially, the pages of allocated memory are protected to allow all forms of access, and are inherited in child tasks as a copy. Subsequent calls to **vm_protect()** and **vm_inherit()** may be used to change these properties. The allocated region is always zero-filled.

Note: Unless you have a special reason for calling **vm_allocate()** (such as a need for page-aligned memory), you should usually call **malloc()** or a similar C library function instead. The C library functions don't necessarily make UNIX or Mach system calls, so they're generally faster than using a Mach function such as **vm_allocate()**.

EXAMPLE

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

RETURN

KERN_SUCCESS: Memory allocated.

KERN_INVALID_ADDRESS: Illegal address specified.

KERN_NO_SPACE: Not enough space left to satisfy this request.

SEE ALSO

vm_deallocate(), **vm_inherit()**, **vm_protect()**, **vm_region()**, **vm_statistics()**

vm_copy()

SUMMARY Copy virtual memory

SYNOPSIS

#include <mach.h>

kern_return_t **vm_copy**(vm_task_t *target_task*, vm_address_t *source_address*, vm_size_t *size*, vm_address_t *dest_address*)

ARGUMENTS

target_task: The task whose virtual memory is to be affected.

source_address: The address in *target_task* of the start of the source range (must be a page boundary).

size: The number of bytes to copy (must be a multiple of **vm_page_size**).

dest_address: The address in *target_task* of the start of the destination range (must be a page boundary).

DESCRIPTION

The function **vm_copy()** causes the source memory range to be copied to the destination address; the destination region must not overlap the source region. The destination address range must already be allocated and writable; the source range must be readable.

For languages other than C, the value of **vm_page_size** can be obtained by calling **vm_statistics()**.

EXAMPLE

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

RETURN

KERN_SUCCESS: Memory copied.

KERN_INVALID_ARGUMENT: The address doesn't start on a page boundary or the size isn't a multiple of **vm_page_size**.

KERN_PROTECTION_FAILURE: The destination region isn't writable or the source region isn't readable.

KERN_INVALID_ADDRESS: An illegal or nonallocated address was specified, or insufficient memory was allocated at one of the addresses.

SEE ALSO

vm_allocate(), vm_protect(), vm_write(), vm_statistics()

vm_deallocate()

SUMMARY Deallocate virtual memory

SYNOPSIS

#include <mach.h>

kern_return_t **vm_deallocate**(vm_task_t *target_task*, vm_address_t *address*,
vm_size_t *size*)

ARGUMENTS

target_task: Task whose virtual memory is to be affected.

address: Starting address (this gets rounded down to a page boundary).

size: Number of bytes to deallocate (this gets rounded up to a page boundary).

DESCRIPTION

The function **vm_deallocate**() relinquishes access to a region of a task's address space, causing further access to that memory to fail. This address range will be available for reallocation. Note that because of the rounding to virtual page boundaries, more than *size* bytes may be deallocated. Use **vm_statistics**() or the global variable **vm_page_size** to find out the current virtual page size.

This function may be used to deallocate memory that was passed to a task in a message (via out-of-line data). In that case, the rounding should cause no trouble, since the region of memory was allocated as a set of pages.

The function **vm_deallocate**() affects only the task specified by *target_task*. Other tasks that may have access to this memory can continue to reference it.

EXAMPLE

```
r = vm_deallocate(task_self(), (vm_address_t)thread_list,
    sizeof(thread_list)*thread_count);
if (r != KERN_SUCCESS)
    mach_error("Trouble freeing thread_list", r);
```

RETURN

KERN_SUCCESS: Memory deallocated.

KERN_INVALID_ADDRESS: Illegal or nonallocated address specified.

SEE ALSO

vm_allocate(), **vm_statistics**(), **msg_receive**()

vm_inherit()

SUMMARY Inherit virtual memory

SYNOPSIS

#include <mach.h>

kern_return_t **vm_inherit**(vm_task_t *target_task*, vm_address_t *address*, vm_size_t *size*, vm_inherit_t *new_inheritance*)

ARGUMENTS

target_task: Task whose virtual memory is to be affected.

address: Starting address (this gets rounded down to a page boundary).

size: Size in bytes of the region for which inheritance is to change (this gets rounded up to a page boundary).

new_inheritance: How this memory is to be inherited in child tasks. Inheritance is specified by using one of these following three values:

VM_INHERIT_SHARE: Child tasks will share this memory with this task.
VM_INHERIT_COPY: Child tasks will receive a copy of this region.
VM_INHERIT_NONE: This region will be absent from child tasks.

DESCRIPTION

The function **vm_inherit()** specifies how a region of a task's address space is to be passed to child tasks at the time of task creation. Inheritance is an attribute of virtual pages; thus the addresses and size of memory to be set will be rounded to refer to whole pages.

Setting **vm_inherit()** to VM_INHERIT_SHARE and forking a child task is the only way two Mach tasks can share physical memory. However, all the threads of a given task share all the same memory.

EXAMPLE

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

RETURN

KERN_SUCCESS: The inheritance has been set.

KERN_INVALID_ADDRESS: Illegal address specified.

SEE ALSO

task_create(), vm_region()

vm_protect()

SUMMARY Protect virtual memory

SYNOPSIS

#include <mach.h>

kern_return_t **vm_protect**(vm_task_t *target_task*, vm_address_t *address*, vm_size_t *size*, boolean_t *set_maximum*, vm_prot_t *new_protection*)

ARGUMENTS

target_task: Task whose virtual memory is to be affected.

address: Starting address (this gets rounded down to a page boundary).

size: Size in bytes of the region for which protection is to change (this gets rounded up to a page boundary).

set_maximum: If set, make the protection change apply to the maximum protection associated with this address range; otherwise, change the current protection on this range. If the maximum protection is reduced below the current protection, both will be changed to reflect the new maximum.

new_protection: A new protection value for this region; some combination of VM_PROT_READ, VM_PROT_WRITE, and VM_PROT_EXECUTE.

DESCRIPTION

The function **vm_protect()** changes the protection of some pages of allocated memory in a task's address space. In general a protection value permits the named operation. When memory is first allocated it has all protection bits on. The exact interpretation of a protection value is machine-dependent. On a NeXT computer, three levels of memory protection are provided:

- No access
- Read and execute access
- Read, execute, and write access

VM_PROT_WRITE permits read, execute, and write access; VM_PROT_READ or VM_PROT_EXECUTE permits read and execute access, but not write access.

EXAMPLE

```
vm_address_t    addr = (vm_address_t)mlock;

r = vm_protect(task_self(), addr, vm_page_size, FALSE, 0);
if (r != KERN_SUCCESS) {
    mach_error("vm_protect 0", r);
    exit(1);
}
printf("protect on\n");
```

RETURN

KERN_SUCCESS: The memory has been protected.

KERN_PROTECTION_FAILURE: An attempt was made to increase the current or maximum protection beyond the existing maximum protection value.

KERN_INVALID_ADDRESS: An illegal or nonallocated address was specified.

vm_read()

SUMMARY Read virtual memory

SYNOPSIS

#include <mach.h>

kern_return_t **vm_read**(vm_task_t *target_task*, vm_address_t *address*, vm_size_t *size*, pointer_t **data*, unsigned int **data_count*)

ARGUMENTS

target_task: Task whose memory is to be read.

address: The first address to be read (must be on a page boundary).

size: The number of bytes of data to be read (must be a multiple of **vm_page_size**).

data: The array of data copied from the given task.

data_count: Returns the size of the data array in bytes (will be an integral number of pages).

DESCRIPTION

The function **vm_read**() allows one task's virtual memory to be read by another task. The data array is returned in a newly allocated region; the task reading the data should call **vm_deallocate**() on this region when it's done with the data.

For languages other than C, the value of **vm_page_size** can be obtained by calling **vm_statistics**().

EXAMPLE

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

RETURN

KERN_SUCCESS: The memory has been read.

KERN_INVALID_ARGUMENT: Either *address* does not start on a page boundary or *size* isn't an integral number of pages.

KERN_NO_SPACE: There isn't enough room in the caller's virtual memory to allocate space for the data to be returned.

KERN_PROTECTION_FAILURE: The address region in the target task is protected against reading.

KERN_INVALID_ADDRESS: An illegal or nonallocated address was specified, or there were not *size* bytes of data following that address.

SEE ALSO

vm_write(), vm_copy(), vm_deallocate()

vm_region()

SUMMARY Get information about virtual memory regions

SYNOPSIS

#include <mach.h>

kern_return_t **vm_region**(vm_task_t *target_task*, vm_address_t **address*, vm_size_t **size*, vm_prot_t **protection*, vm_prot_t **max_protection*, vm_inherit_t **inheritance*, boolean_t **shared*, port_t **object_name*, vm_offset_t **offset*)

ARGUMENTS

target_task: The task for which an address space description is requested.

address: The address at which to start looking for a region. On return, *address* will contain the start of the region (therefore, the value returned will be different from the value that was passed in if the specified region is part of a larger region).

size: Returns the size (in bytes) of the located region.

protection: Returns the current protection of the region.

max_protection: Returns the maximum allowable protection for this region.

inheritance: Returns the inheritance attribute for this region.

shared: Returns true if this region is shared, false if it isn't.

object_name: Returns the port identifying the region's memory object.

offset: Returns the offset into the pager object at which this region begins.

DESCRIPTION

The function **vm_region()** returns a description of the specified region of the target task's virtual address space. **vm_region()** begins at *address*, looking forward through memory until it comes to an allocated region. (If *address* is in a region, that region is used.) If *address* isn't in a region, it's set to the start of the first region that follows the incoming value. In this way an entire address space can be scanned. You can set *address* to the constant VM_MIN_ADDRESS (defined in the header file **machine/vm_param.h**) to specify the first address in the address space.

EXAMPLE

```
char            data;
kern_return_t   r;
vm_size_t       size;
vm_prot_t       protection, max_protection;
vm_inherit_t    inheritance;
boolean_t       shared;
port_t          object_name;
vm_offset_t     offset;
```

```
/* . . . */
/* Check the inheritance of "data". */
r = vm_region(task_self(), &(vm_address_t)size, &size, &protection,
    &max_protection, &inheritance, &shared, &object_name, &offset);

if (r != KERN_SUCCESS)
    mach_error("Error calling vm_region", r);
else {
    printf("Protection is:  ");
    switch (inheritance) {
        case VM_INHERIT_SHARE:
            printf("Share with child\n");
            break;
        case VM_INHERIT_COPY:
            printf("Copy into child\n");
            break;
        case VM_INHERIT_NONE:
            printf("Absent from child\n");
            break;
        case VM_INHERIT_DONATE_COPY:
            printf("Copy and delete\n");
            break;
    }
}
```

RETURN

KERN_SUCCESS: The region was located and information has been returned.

KERN_NO_SPACE: The task contains no region at or above *address*.

SEE ALSO

vm_allocate(), **vm_deallocate()**, **vm_protect()**, **vm_inherit()**

vm_statistics()

SUMMARY Examine virtual memory statistics

SYNOPSIS

#include <mach.h>

kern_return_t **vm_statistics**(vm_task_t *target_task*, vm_statistics_data_t **vm_stats*)

ARGUMENTS

target_task: The task that's requesting the statistics.

vm_stats: Returns the statistics.

DESCRIPTION

The function **vm_statistics**() returns statistics about the kernel's use of virtual memory since the kernel was booted. **pagesize** can also be found through the global variable **vm_page_size**, which is set at task initialization and remains constant for the life of the task.

```
struct vm_statistics {
    long  pagesize;         /* page size in bytes */
    long  free_count;       /* number of pages free */
    long  active_count;     /* number of pages active */
    long  inactive_count;   /* number of pages inactive */
    long  wire_count;       /* number of pages wired down */
    long  zero_fill_count;  /* number of zero-fill pages */
    long  reactivations;    /* number of pages reactivated */
    long  pageins;          /* number of pageins */
    long  pageouts;         /* number of pageouts */
    long  faults;           /* number of faults */
    long  cow_faults;       /* number of copy-on-writes */
    long  lookups;          /* object cache lookups */
    long  hits;             /* object cache hits */
};
typedef struct vm_statistics   vm_statistics_data_t;
```

EXAMPLE

```
result=vm_statistics(task_self(), &vm_stats);
if (result != KERN_SUCCESS)
    mach_error("An error calling vm_statistics()!", result);
else
    printf("%d bytes of RAM are free\n",
        vm_stats.free_count * vm_stats.pagesize);
```

RETURN

KERN_SUCCESS: The operation was successful.

vm_write()

SUMMARY Write virtual memory

SYNOPSIS

#include <mach.h>

kern_return_t **vm_write**(vm_task_t *target_task*, vm_address_t *address*, pointer_t *data*, unsigned int *data_count*)

ARGUMENTS

target_task: Task whose memory is to be written.

address: Starting address in task to be affected (must be a page boundary).

data: An array of bytes to be written.

data_count: The size in bytes of the data array (must be a multiple of **vm_page_size**).

DESCRIPTION

The function **vm_write()** allows a task's virtual memory to be written by another task. For languages other than C, the value of **vm_page_size** can be obtained by calling **vm_statistics()**.

RETURN

KERN_SUCCESS: Memory written.

KERN_INVALID_ARGUMENT: The address doesn't start on a page boundary or the size isn't an integral number of pages.

KERN_PROTECTION_FAILURE: The address region in the target task is protected against writing.

KERN_INVALID_ADDRESS: An illegal or nonallocated address was specified or the amount of allocated memory starting at *address* was less than *data_count*.

SEE ALSO

vm_copy(), **vm_protect()**, **vm_read()**, **vm_statistics()**

Bootstrap Server Functions

See **/usr/include/servers/bootstrap.defs** for documentation of how the Bootstrap Server works.

The Bootstrap Server was created by NeXT, so these functions aren't in other versions of Mach.

bootstrap_check_in()

SUMMARY Get receive rights to a service port

SYNOPSIS

#include <mach.h>
#include <servers/bootstrap.h>

kern_return_t **bootstrap_check_in**(port_t *bootstrap_port*, name_t *service_name*, port_all_t **service_port*)

ARGUMENTS

bootstrap_port: A bootstrap port.

service_name: The string that names the service.

service_port: Returns receive rights to the service's port.

DESCRIPTION

Use this routine in a server to start providing a service. The service must already be defined, either by the appropriate line in **/etc/bootstrap.conf** or by a call to **bootstrap_create_service**(). Calling **bootstrap_check_in**() makes the service active.

EXAMPLE

```
extern port_t bootstrap_port;
port_all_t    my_service_port;

/* Get receive rights for our service. */
result=bootstrap_check_in(bootstrap_port, MYNAME, &my_service_port);
if (result != BOOTSTRAP_SUCCESS)
    mach_error("Couldn't create service", result);
```

RETURN

BOOTSTRAP_SUCCESS: The call succeeded.

BOOTSTRAP_NOT_PRIVILEGED: *bootstrap_port* is an unprivileged bootstrap port.

BOOTSTRAP_UNKNOWN_SERVICE: The service doesn't exist. It might be defined in a subset (see **bootstrap_subset**()).

BOOTSTRAP_SERVICE_ACTIVE: The service has already been registered or checked in and the server hasn't died.

Returns appropriate kernel errors on RPC failure.

bootstrap_create_service()

SUMMARY Create a service and service port

SYNOPSIS

#include <mach.h>
#include <servers/bootstrap.h>

kern_return_t **bootstrap_create_service**(port_t *bootstrap_port*, name_t *service_name*, port_t **service_port*)

ARGUMENTS

bootstrap_port: A bootstrap port.

service_name: The string that specifies the service.

service_port: Returns send rights for the service.

DESCRIPTION

Creates a service named *service_name* and returns send rights to that port in *service_port*. The port may later be checked in as if this port were configured in the bootstrap configuration file. (At that time **bootstrap_check_in**() will return receive rights to *service_port* and will make the service active.)

This function is often used to create services that are available only to a subset of tasks (see **bootstrap_subset**()). Any task can call this routine—it doesn't have to be the server—as long as the task's bootstrap port isn't unprivileged.

EXAMPLE

```
/* Tell the bootstrap server about a service. */
result=bootstrap_create_service(bootstrap_port, SERVICENAME,
    &service_port);
if (result!=BOOTSTRAP_SUCCESS)
    mach_error("Couldn't create service", result);
```

RETURN

BOOTSTRAP_SUCCESS: The call succeeded.

BOOTSTRAP_NOT_PRIVILEGED: *bootstrap_port* is an unprivileged bootstrap port.

BOOTSTRAP_SERVICE_ACTIVE: The service already exists.

Returns appropriate kernel errors on RPC failure.

bootstrap_get_unpriv_port()

SUMMARY Get an unprivileged bootstrap port

SYNOPSIS

#include <mach.h>
#include <servers/bootstrap.h>

kern_return_t **bootstrap_get_unpriv_port**(port_t *bootstrap_port*,
port_t **unpriv_port*)

ARGUMENT

bootstrap_port: A bootstrap port.

unpriv_port: Returns an unprivileged bootstrap port.

DESCRIPTION

Returns an unprivileged bootstrap port for the Bootstrap Server. Unprivileged ports are used just like regular bootstrap ports, except that they can't be used for **bootstrap_check_in**(), **bootstrap_create_service**(), or **bootstrap_register**() requests.

One use of unprivileged bootstrap ports is to ensure that remote processes aren't able to create or provide services. In this case, the task that spawns the remote processes sets its own bootstrap port to *unpriv_port*; the remote processes then inherit this unprivileged port as their bootstrap port.

EXAMPLE

```
extern port_t  bootstrap_port;
port_t         unpriv_port;

result=bootstrap_get_unpriv_port(bootstrap_port, &unpriv_port);
if (result != BOOTSTRAP_SUCCESS)
    printf("Couldn’t get the unprivileged port (%d)\n", result);
else {
    /*
     * Set our bootstrap port so that tasks we create inherit the
     * unprivileged port.
     */
    result=task_set_bootstrap_port(task_self(), unpriv_port);
    if (result != KERN_SUCCESS)
        mach_error(“task_set_bootstrap_port() failed”, result);
    bootstrap_port=unpriv_port;
```

RETURN

BOOTSTRAP_SUCCESS: The call succeeded.

Returns appropriate kernel errors on RPC failure.

bootstrap_info()

SUMMARY Get information about all known services

SYNOPSIS

#include <mach.h>
#include <servers/bootstrap.h>

kern_return_t **bootstrap_info**(port_t *bootstrap_port*, name_array_t **service_names*, unsigned int **service_names_count*, name_array_t **server_names*, unsigned int **server_names_count*, bool_array_t **service_active*, unsigned int **service_active_count*)

ARGUMENTS

bootstrap_port: A bootstrap port.

service_names: Returns the names of all known services.

service_names_count: Returns the number of service names.

server_names: Returns the name, if known, of the server that provides the corresponding service. Except for the **mach_init** server, this name isn’t known unless the bootstrap configuration file has a **server** line for this server.

server_names_count: Returns the number of server names.

service_active: Returns an array of booleans that correspond to the *service_names* array. For each item, the boolean value is true if the service is receiving messages sent to its port, otherwise false.

service_active_count: Returns the number of items in the *service_active* array.

DESCRIPTION

This routine returns information about all services that are known. Note that it won't return information on services that are defined only in subsets, unless the subset port is an ancestor of *bootstrap_port*. (See **bootstrap_subset()** for information on subsets.)

EXAMPLE

```
result = bootstrap_info(bootstrap_port, &service_names, &service_cnt,
  &server_names, &server_cnt, &service_active, &service_active_cnt);
if (result != BOOTSTRAP_SUCCESS)
    printf("ERROR:  info failed: %d", result);
else {
    for (i = 0; i < service_cnt; i++)
        printf("Name: %-15s   Server: %-15s    Active: %-4s",
            service_names[i],
            server_names[i][0] == '\0' ? "Unknown" : server_names[i],
            service_active[i] ? "Yes\n" : "No\n");
}
```

RETURN

BOOTSTRAP_SUCCESS: The call succeeded.

BOOTSTRAP_NO_MEMORY: The Bootstrap Server couldn't allocate enough memory to return the information.

Returns appropriate kernel errors on RPC failure.

bootstrap_look_up()

SUMMARY Get the service port of a particular service

SYNOPSIS

#include <mach.h>
#include <servers/bootstrap.h>

kern_return_t **bootstrap_look_up**(port_t *bootstrap_port*, name_t *service_name*, port_t **service_port*)

ARGUMENTS

bootstrap_port: A bootstrap port.

service_name: The string that identifies the service.

service_port: Returns send rights for the service port.

DESCRIPTION

Returns send rights for the service port of the specified service. The service isn't guaranteed to be active. (To check whether the service is active, use **bootstrap_status**().)

EXAMPLE

```
result=bootstrap_look_up(bootstrap_port, "FreeService2", &srvc_port);
if (result!=BOOTSTRAP_SUCCESS)
    printf("lookup failed: %d\n", result);
else {
    /* Access the service by sending messages to srvc_port. */
}
```

RETURN

BOOTSTRAP_SUCCESS: The call succeeded.

BOOTSTRAP_UNKNOWN_SERVICE: The service doesn't exist. It might be defined in a subset (see **bootstrap_subset**()).

Returns appropriate kernel errors on RPC failure.

bootstrap_look_up_array()

SUMMARY Get the service ports for an array of services

SYNOPSIS

#include <mach.h>
#include <servers/bootstrap.h>

kern_return_t **bootstrap_look_up_array**(port_t *bootstrap_port*,
name_array_t *service_names*, unsigned int *service_names_count*,
port_array_t **service_ports*, unsigned int **service_ports_count*,
boolean_t **all_services_known*)

ARGUMENTS

bootstrap_port: A bootstrap port.

service_names: An array of service names.

service_names_count: The number of service names.

service_ports: Returns an array of service ports.

service_ports_count: Returns the number of service ports. This should be equal to *service_names_count*.

all_services_known: Returns true if every service name was recognized; otherwise returns false.

DESCRIPTION

Returns port send rights in corresponding entries of the array *service_ports* for all services named in the array *service_names*. You should call **vm_deallocate**() on *service_ports* when you no longer need it.

Unknown service names have the corresponding service port set to PORT_NULL. Note that these services might be available in a subset (see **bootstrap_subset**()).

EXAMPLE

```
kern_return_t   result;
port_t          my_bootstrap_port;
unsigned int    port_cnt;
boolean_t       all_known;
name_t          name_array[2]={"Service", "NetMessage"};
port_array_t    ports;

result = task_get_bootstrap_port(task_self(), &my_bootstrap_port);
if (result != KERN_SUCCESS) {
    mach_error("Couldn't get bootstrap port", result);
    exit(1);
}

result=bootstrap_look_up_array(my_bootstrap_port, name_array, 2,
    &ports, &port_cnt, &all_known);
if (result!=BOOTSTRAP_SUCCESS)
    mach_error("Lookup array failed", result);
else
    printf("Port count = %d, all known = %d\n", port_cnt, all_known);

/* . . . */
result=vm_deallocate(task_self(), (vm_address_t)ports,
    sizeof(ports)*port_cnt);
if (result != KERN_SUCCESS)
    mach_error("Trouble freeing ports", result);
```

RETURN

BOOTSTRAP_SUCCESS: The call succeeded.

BOOTSTRAP_BAD_COUNT: *service_names_count* was too large (greater than BOOTSTRAP_MAX_LOOKUP_COUNT, which is defined in the header file **server/bootstrap_defs.h**).

Returns appropriate kernel errors on RPC failure.

bootstrap_register()

SUMMARY Register send rights for a service port

SYNOPSIS

#include <mach.h>
#include <servers/bootstrap.h>

kern_return_t **bootstrap_register**(port_t *bootstrap_port*, name_t *service_name*, port_t *service_port*)

ARGUMENTS

bootstrap_port: A bootstrap port.

service_name: The string that identifies the service.

service_port: The service port for the service.

DESCRIPTION

You can use this function to create a server that hasn't been defined in the bootstrap configuration file. This function specifies to the Bootstrap Server exactly which port should be the service port.

You can't register a service if an active binding already exists. However, you can register a service if the existing binding is inactive (that is, the Bootstrap Server currently holds receive rights for the service port); in this case the previous service port will be deallocated.

A service that is restarting can resume service for previous clients by setting *service_port* to the previous service port. You can get this port by calling **bootstrap_check_in**().

EXAMPLE

```
/* Create a port to use as the service port. */
result=port_allocate(task_self(), &myport);
if (result != KERN_SUCCESS) {
    mach_error("Couldn't allocate a service port", result);
    exit(1);
}

/* Tell the bootstrap server about my service. */
result=bootstrap_register(bootstrap_port, MYNAME, myport);
if (result != BOOTSTRAP_SUCCESS)
    printf("Call to bootstrap_register failed: %d", result);
```

RETURN

BOOTSTRAP_SUCCESS: The call succeeded.

BOOTSTRAP_NOT_PRIVILEGED: *bootstrap_port* is an unprivileged bootstrap port.

BOOTSTRAP_NAME_IN_USE: The service is already active.

Returns appropriate kernel errors on RPC failure.

bootstrap_status()

SUMMARY Check whether a service is available

SYNOPSIS

#include <mach.h>
#include <servers/bootstrap.h>

kern_return_t **bootstrap_status**(port_t *bootstrap_port*, name_t *service_name*, boolean_t **service_active*)

ARGUMENTS

bootstrap_port: A bootstrap port.

service_name: The string that specifies a particular service.

service_active: Returns true if the service is active, false otherwise.

DESCRIPTION

This function tells you whether a service is known to users of *bootstrap_port*, and whether it's active. A service is active if a server is able to receive messages on its service port. If a service isn't active, the Bootstrap Server holds receive rights for the service port.

EXAMPLE

```
result=bootstrap_status(bootstrap_port, MYNAME, &service_active);
if (result!=BOOTSTRAP_SUCCESS)
    printf("status check failed\n");
else {
    if (service_active)
        printf("Server %s is active\n", MYNAME);
    else
        printf ("Server %s is NOT active\n", MYNAME);
}
```

RETURN

BOOTSTRAP_SUCCESS: The call succeeded.

BOOTSTRAP_UNKNOWN_SERVICE: The service doesn't exist. It might be defined in a subset (see **bootstrap_subset**()).

Returns appropriate kernel errors on RPC failure.

bootstrap_subset()

SUMMARY Get a new port to use as a bootstrap port

SYNOPSIS

#include <mach.h>
#include <servers/bootstrap.h>

kern_return_t **bootstrap_subset**(port_t *bootstrap_port*, port_t *requestor_port*, port_t **subset_port*)

ARGUMENTS

bootstrap_port: A bootstrap port.

requestor_port: A port that determines the life-span of the subset.

subset_port: Returns the subset port.

DESCRIPTION

Returns a new port to use as a bootstrap port. This port behaves exactly like the previous *bootstrap_port*, with one exception: When you register a port by calling **bootstrap_register**() using *subset_port* as the bootstrap port, the registered port is available only to users of *subset_port* and its descendants. Lookups on the *subset_port* will return ports registered with this port specifically, and will also return ports registered with ancestors of this *subset_port*. (The ancestors of *subset_port* are *bootstrap_port* and, if *bootstrap_port* is itself a subset port, any ancestors of *bootstrap_port*.)

You can override a service already registered with an ancestor port by registering it with the subset port. Any thread that looks up the service using the subset port will see only the version of the service that's registered with the subset port. This is one way to transparently provide services such as monitor programs or individualized spell checkers, while the rest of the system still uses the default service.

When it's detected that *requestor_port* is destroyed, the subset port and all services advertised by it are destroyed as well.

EXAMPLE

```
/*
 * Get a subset port.
 */
print("Subset port test");
result = bootstrap_subset(bootstrap_port, task_self(), &subset_port);
if (result != BOOTSTRAP_SUCCESS)
    mach_error("Couldn't get unpriv port", result);
```

RETURN

BOOTSTRAP_SUCCESS: The call succeeded.

BOOTSTRAP_NOT_PRIVILEGED: *bootstrap_port* is an unprivileged bootstrap port.

Returns appropriate kernel errors on RPC failure.

Network Name Server Functions

netname_check_in()

SUMMARY Check a name into the local namespace

SYNOPSIS

#include <mach.h>
#include <servers/netname.h>

kern_return_t **netname_check_in**(port_t *server_port*, netname_name_t *port_name*, port_t *signature*, port_t *port_id*)

ARGUMENTS

server_port: The port to the Network Name Server.

port_name: The name of the port that's to be checked in.

signature: The port that's used to protect the right to remove a name.

port_id: The port that's to be checked in.

DESCRIPTION

The function **netname_check_in**() enters a port with the name *port_name* into the namespace of the local network server. *signature* is a port that's used to protect this name. This same port must be presented on a **netname_check_out**() call for that call to be able to remove the name from the namespace.

Note that the *server_port* parameter should be set to **name_server_port** in order to use the system Network Name Server.

RETURN

NETNAME_SUCCESS: The operation succeeded.

SEE ALSO

netname_check_out(), **netname_look_up**()

netname_check_out()

SUMMARY Remove a name from the local namespace

SYNOPSIS

#include <mach.h>
#include <servers/netname.h>

kern_return_t **netname_check_out**(port_t *server_port*, netname_name_t *port_name*, port_t *signature*)

ARGUMENTS

server_port: The port to the Network Name Server.

port_name: The name of the port that's to be checked out.

signature: The port that's used to protect the right to remove a name.

DESCRIPTION

The function **netname_check_out**() removes a port with the name *port_name* from the namespace of the local network server. *signature* must be the same port as the signature port passed to **netname_check_in**() when this name was checked in.

Note that the *server_port* parameter should be set to **name_server_port** in order to use the system Network Name Server.

RETURN

NETNAME_SUCCESS: The operation succeeded.

NAME_NOT_YOURS: The signature given to **netname_check_out**() did not match the signature with which the port was checked in.

SEE ALSO

netname_check_in(), **netname_look_up**()

netname_look_up()

SUMMARY Look up a name on a specific host

SYNOPSIS

#include <mach.h>
#include <servers/netname.h>

kern_return_t **netname_look_up**(port_t *server_port*, netname_name_t *host_name*, netname_name_t *port_name*, port_t **port_id*)

ARGUMENTS

server_port: The port to the Network Name Server.

host_name: The name of the host to query. This can't be a null pointer.

port_name: The name of port to be looked up.

port_id: The port that was looked up.

DESCRIPTION

The function **netname_look_up()** returns the value of the port named by *port_name* by questioning the host named by the *host_name* argument. Thus this call is a directed name lookup. The *host_name* may be any of the host's official nicknames. If it's an empty string, the local host is assumed. If *host_name* is "*", a broadcast lookup is performed.

The *server_port* parameter should be set to **name_server_port** in order to use the system Network Name Server.

Important: Use **NXPortNameLookup()** instead of **netname_look_up()** in all NeXTstep applications. (In the future, Listener instances might register with the Bootstrap Server instead of the Network Name Server.)

RETURN

NETNAME_SUCCESS: The operation succeeded.

NAME_NOT_CHECKED_IN: **netname_look_up()** could not find the name at the given host.

NETNAME_NO_SUCH_HOST: The *host_name* argument to **netname_look_up()** does not name a valid host.

NETNAME_HOST_NOT_FOUND: **netname_look_up()** could not reach the host named by *host_name* (for instance, because it's down).

SEE ALSO

netname_check_in(), **netname_check_out()**

Kernel-Server Loader Functions

To use these functions, you must compile with the kernload library. For example:

```
cc myprog.c -lkernload
```

kern_loader_abort()

SUMMARY Shut down or reconfigure **kern_loader**

SYNOPSIS

#include <mach.h>
#include <kernserv/kern_loader.h>

kern_return_t **kern_loader_abort**(port_t *loader_port*, port_t *priv_port*, boolean_t *restart*)

ARGUMENTS

loader_port: **kern_loader**'s port, obtained from **kern_loader_look_up()**.

priv_port: The privileged port for this host, returned by **host_priv_self()**.

restart: If true, reconfigure **kern_loader**.

DESCRIPTION

This function unloads and deallocates all loadable kernel servers and then, depending on the value of *restart*, kills or reconfigures the kernel-server loader. If *restart* is true, then **kern_loader** rereads its configuration file (**etc/kern_loader.conf**) to determine which servers it should allocate and load.

EXAMPLE

```
/* Get kern_loader's port. */
error=kern_loader_look_up(&loader_port);
if (error != KERN_SUCCESS) {
    kern_loader_error("Couldn't find kern_loader's port", error);
    exit(1);
}

/* Reconfigure kern_loader. */
error=kern_loader_abort(loader_port, host_priv_self(), TRUE);
if (error != KERN_SUCCESS)
    kern_loader_error("Couldn't stop kern_loader", error);
```

RETURN

KERN_SUCCESS: The call was successful.

KERN_LOADER_NO_PERMISSION: *priv_port* isn't the host's privileged port. (Make sure **host_priv_self**() is called by a process with superuser permission.)

SEE ALSO

kern_loader_delete_server(), **kern_loader_look_up**(), **kern_loader_unload_server**()

kern_loader_add_server()

SUMMARY Allocate a loadable kernel server

SYNOPSIS

#include <mach.h>
#include <kernserv/kern_loader.h>

kern_return_t **kern_loader_add_server**(port_t *loader_port*, port_t *task_port*, server_reloc_t *server_reloc*)

ARGUMENTS

loader_port: **kern_loader**'s port, obtained from calling **kern_loader_look_up**().

task_port: The kernel's task port, obtained using **task_by_unix_pid**().

server_reloc: The server's relocatable object file. For example, the MIDI driver's relocatable object file is "/usr/lib/kern_loader/Midi/midi_reloc".

DESCRIPTION

This function prepares the loadable kernel server to be loaded into the kernel. The server isn't loaded unless it automatically loads when allocated.

If the server is already loaded or allocated, then the server is unloaded (if necessary) and allocated again from scratch.

EXAMPLE

```
/* Get kern_loader's port. */
r = kern_loader_look_up(&loader_port);
if (r != KERN_SUCCESS) {
    kern_loader_error("Couldn't get loader_port", r);
    exit(1);
}
```

```
/* Get the kernel's task port. */
r = task_by_unix_pid(task_self(), 0, &kernel_task);
if (r != KERN_SUCCESS) {
    kern_loader_error("Couldn't get kernel_task", r);
    exit(2);
}

/* Add the server. */
r = kern_loader_add_server(loader_port, kernel_task,
    "/usr/lib/kern_loader/Midi/midi_reloc");
if (r != KERN_SUCCESS) {
    kern_loader_error("Call to kern_loader_abort failed", r);
    exit(3);
}
```

RETURN

KERN_SUCCESS: The server has been successfully allocated.

KERN_LOADER_NO_PERMISSION: *task_port* wasn't the kernel's task port. (Make sure **task_by_unix_pid()** is called by a process with superuser permission.)

KERN_LOADER_SERVER_WONT_LOAD: The kernel-server loader couldn't use *server_reloc* to build an loadable object file, or it couldn't understand the load or unload commands, or it couldn't link the loadable object file against **/mach**.

SEE ALSO

kern_loader_delete_server(), **kern_loader_look_up()**

kern_loader_delete_server()

SUMMARY Delete a loadable kernel server

SYNOPSIS

#include <mach.h>
#include <kernserv/kern_loader.h>

kern_return_t **kern_loader_delete_server**(port_t *loader_port*, port_t *task_port*, server_name_t *server_name*)

ARGUMENTS

loader_port: **kern_loader**'s port, obtained from calling **kern_loader_look_up()**.

task_port: The kernel's task port, obtained using **task_by_unix_pid()**.

server_name: The string associated with the server. For example, the MIDI driver's name is "midi".

DESCRIPTION

This function removes the loadable kernel server from **kern_loader**'s control. If the server is currently loaded, then it's unloaded.

EXAMPLE

```
/* Get kern_loader's port. */
error=kern_loader_look_up(&loader_port);
if (error != KERN_SUCCESS) {
    kern_loader_error("Couldn't find kern_loader's port", error);
    exit(1);
}

/* Get the kernel's task port. */
error=task_by_unix_pid(task_self(), 0, &kern_port);
if (error != KERN_SUCCESS) {
    mach_error("Error looking up kernel port", error);
    exit(2);
}

/* Delete the server. */
error=kern_loader_delete_server(loader_port, kern_port, "midi");
if (error != KERN_SUCCESS) {
    kern_loader_error("Couldn't delete midi", error);
    exit(3);
}
```

RETURN

KERN_SUCCESS: The call succeeded.

KERN_LOADER_NO_PERMISSION: *task_port* wasn't the kernel's task port.
(Make sure **task_by_unix_pid**() is called by a process with superuser permission.)

KERN_LOADER_UNKNOWN_SERVER: *server_name* wasn't recognized.

SEE ALSO

kern_loader_add_server(), **kern_loader_look_up**()

kern_loader_error(), kern_loader_error_string()

SUMMARY

Display or return an error message

SYNOPSIS

#include <mach.h>
#include <kernserv/kern_loader_error.h>

void **kern_loader_error**(const char **string*, kern_return_t *errno*)
const char ***kern_loader_error_string**(kern_return_t *errno*)

ARGUMENTS

string: The string to be printed along with the error message.

errno: The value returned by a Mach function.

DESCRIPTION

These functions act like **mach_error()** and **mach_error_string()**, except that they also understand errors from the kernel-server loader functions.

kern_loader_error() prints to **stderr** the *string*, followed by the string corresponding to *errno*, followed by *errno* in parentheses. **kern_loader_error_string()** returns the string that corresponds to *errno*.

EXAMPLE

```
error=kern_loader_delete_server(loader_port, kern_port, "midi");
if (error != KERN_SUCCESS) {
    kern_loader_error("Couldn't delete midi", error);
    exit(3);
}
```

SEE ALSO

mach_error(), mach_error_string()

kern_loader_get_log()

SUMMARY Request a message containing kernel log data

SYNOPSIS

#include <mach.h>
#include <kernserv/kern_loader_types.h>

kern_return_t **kern_loader_get_log**(port_t *loader_port*, port_t *server_com_port*, port_t *reply_port*)

ARGUMENTS

loader_port: **kern_loader**'s port, obtained from **kern_loader_look_up()**.

server_com_port: The loadable kernel server's communication port, obtained from **kern_loader_server_com_port()**.

reply_port: The port to which **kern_loader** should send the reply message.

DESCRIPTION

This function requests a reply message containing data logged by a loadable kernel server. Before calling this function for the first time on a server, you should turn the server's logging on by calling **kern_loader_log_level**().

You must supply the implementation of the reply message, as described in Chapter 3, "Using Loadable Kernel Servers."

Each item of logged data is preceded by a time stamp. The time stamp is a relative indicator of when the data was logged by the loadable kernel server.

EXAMPLE

```
r = kern_loader_look_up(&kl_port);
if (r != KERN_SUCCESS) {
    mach_error("Can't find kernel loader", r);
    exit(1);
}

r = port_allocate(task_self(), &reply_port);
if (r != KERN_SUCCESS) {
    mach_error("Can't allocate reply port", r);
    exit(1);
}

/* Get the server's communication port. */
r = task_by_unix_pid(task_self(), 0, &kern_port);
if (r != KERN_SUCCESS) {
    mach_error("Error looking up kernel's port", r);
    exit(1);
}
r = kern_loader_server_com_port(kl_port, kern_port, MYDRIVER_NAME,
    &server_com_port);
if (r != KERN_SUCCESS) {
    kern_loader_error("Error looking up server com port", r);
    exit(1);
}

/* Set the log level so we'll get log messages. */
r = kern_loader_log_level(kl_port, server_com_port, LOG_NOTICE);
if (r != KERN_SUCCESS) {
    kern_loader_error("Can't change log level", r);
    exit(1);
}
```

```
    /* Get the first log message. */
    r = kern_loader_get_log(kl_port, server_com_port, reply_port);
    if (r != KERN_SUCCESS) {
        kern_loader_error("Error calling kern_loader_get_log", r);
        exit(1);
    }

    /* Listen for the asynchronous reply message. */
    listen(reply_port);
}

kern_loader_reply_t kern_loader_reply = {
    0,             /* argument to pass to function */
    0,             /* timeout for rpc return msg_send */
    0,             /* string function */
    0,             /* ping function */
    log_data       /* log_data function */
};

void listen(port_name_t port)
{
    char           msg_buf[KERN_LOADER_REPLY_INMSG_SIZE];
    msg_header_t *msg = (msg_header_t *)msg_buf;
    kern_return_t r;

    while (1) {
        /* Receive the next message in the queue. */
        msg->msg_size = KERN_LOADER_REPLY_INMSG_SIZE;
        msg->msg_local_port = port;
        r = msg_receive(msg, MSG_OPTION_NONE, 0);

        if (r != KERN_SUCCESS) {
            mach_error("listen msg_receive", r);
            exit(1);
        }

        /* Handle the message we just received. */
        kern_loader_reply_handler(msg, &kern_loader_reply);
    }
}

kern_return_t log_data(void *arg, printf_data_t log_data, unsigned
int log_data_count)
{
    kern_return_t r;

    /* Print the string we were passed, with our prefix. */
    printf("log_data: %s", log_data);

    /* Deallocate the memory used for the string. */
    vm_deallocate(task_self(), (vm_address_t)log_data,
        log_data_count*sizeof(*log_data));
```

```
        /* Get another log message. */
        r = kern_loader_get_log(kl_port, server_com_port, reply_port);
        if (r != KERN_SUCCESS) {
            kern_loader_error("Error calling kern_loader_get_log", r);
            exit(1);
        }
        return KERN_SUCCESS;
    }
```

RETURN

KERN_SUCCESS: The call succeeded.

KERN_LOADER_UNKNOWN_SERVER: The server is either unknown or has been deallocated.

KERN_LOADER_SERVER_UNLOADED: The server is only allocated, not loaded.

KERN_LOADER_PORT_EXISTS: Someone is already receiving log messages for this server.

SEE ALSO

kern_loader_log_level(), **kern_loader_look_up**(), **kern_loader_reply_handler**(), **kern_loader_server_com_port**()

kern_loader_load_server()

SUMMARY Load a loadable kernel server

SYNOPSIS

#include <mach.h>
#include <kernserv/kern_loader.h>

kern_return_t **kern_loader_load_server**(port_t *loader_port*, server_name_t *server_name*)

ARGUMENTS

loader_port: **kern_loader**'s port, obtained from **kern_loader_look_up**().

server_name: The string associated with the server. For example, the MIDI driver's name is "midi".

DESCRIPTION

This function loads a loadable kernel server that has already been allocated. If the server's relocatable object file has changed since allocation, then the server is allocated again from scratch. This function has no effect on servers that are already loaded; it simply returns KERN_SUCCESS.

EXAMPLE

```
/* Get kern_loader's port. */
error=kern_loader_look_up(&loader_port);
if (error != KERN_SUCCESS) {
    kern_loader_error("Couldn't find kern_loader's port", error);
    exit(1);
}

/* Load the server. */
error=kern_loader_load_server(loader_port, "midi");
if (error != KERN_SUCCESS) {
    kern_loader_error("Couldn't load the server", error);
    exit(2);
}
```

RETURN

KERN_SUCCESS: The server was successfully loaded.

KERN_LOADER_UNKNOWN_SERVER: *server_name* wasn't recognized.

KERN_LOADER_SERVER_WONT_LOAD: The server couldn't be loaded.

SEE ALSO

kern_loader_look_up(), **kern_loader_unload_server()**

kern_loader_log_level()

SUMMARY Set the level of data being logged by a kernel server

SYNOPSIS

#include <mach.h>
#include <kernserv/kern_loader.h>

kern_return_t **kern_loader_log_level**(port_t *loader_port*, port_t *server_com_port*, int *log_level*)

ARGUMENTS

loader_port: **kern_loader**'s port, obtained from **kern_loader_look_up()**.

server_com_port: The loadable kernel server's communication port, obtained from **kern_loader_server_com_port()**.

log_level: An integer indicating the minimum priority of data to be logged. A value of zero turns logging off.

DESCRIPTION

This function determines which data logged by a kernel server gets kept. When a loadable kernel server is first loaded, none of its log messages are kept since its log level is initialized to zero. If you set *log_level* to a value greater than zero, then messages logged at a priority equal to or higher than *log_level* are kept. If you reset the log level to zero, no more log messages are kept until the log level is once again set to a positive value.

Each kernel server can have its own conventions for log priorities; the values defined in the header file **sys/syslog.h** are one possible convention.

EXAMPLE

```
r = kern_loader_look_up(&kl_port);
if (r != KERN_SUCCESS) {
    mach_error("Can't find kernel loader", r);
    exit(1);
}

/* Get the server's communication port. */
r = task_by_unix_pid(task_self(), 0, &kern_port);
if (r != KERN_SUCCESS) {
    mach_error("Error looking up kernel's port", r);
    exit(1);
}
r = kern_loader_server_com_port(kl_port, kern_port, MYDRIVER_NAME,
    &server_com_port);
if (r != KERN_SUCCESS) {
    kern_loader_error("Error looking up server com port", r);
    exit(1);
}

/* Set the log level so we'll get log messages. */
r = kern_loader_log_level(kl_port, server_com_port, LOG_NOTICE);
if (r != KERN_SUCCESS) {
    kern_loader_error("Can't change log level", r);
    exit(1);
}
```

RETURN

KERN_SUCCESS: The log level was successfully set.

KERN_LOADER_UNKNOWN_SERVER: *server_com_port* wasn't valid.

KERN_LOADER_SERVER_UNLOADED: The server isn't currently loaded.

SEE ALSO

kern_loader_look_up(), kern_loader_server_com_port(), kern_loader_log_level()

kern_loader_look_up()

SUMMARY Get **kern_loader**'s port

SYNOPSIS

#include <mach.h>
#include <kernserv/kern_loader_types.h>

kern_return_t **kern_loader_look_up**(port_t **loader_port*)

ARGUMENTS

loader_port: Returns the port on which **kern_loader** receives our messages.

DESCRIPTION

This function returns the service port for the kernel-server loader.

EXAMPLE

```
/* Get kern_loader's port. */
error=kern_loader_look_up(&loader_port);
if (error != KERN_SUCCESS) {
    kern_loader_error("Couldn't find kern_loader's port", error);
    exit(1);
}
```

RETURN

KERN_SUCCESS: The call succeeded.

SEE ALSO

kern_loader_server_com_port()

kern_loader_ping()

SUMMARY Request a synchronization message

SYNOPSIS

#include <mach.h>
#include <kernserv/kern_loader_types.h>

kern_return_t **kern_loader_ping**(port_t *loader_port*, port_t *ping_port*, int *id*)

ARGUMENTS

loader_port: **kern_loader**'s port, obtained from calling **kern_loader_look_up**().

ping_port: The port to which the ping message should be sent.

id: A value to be sent in the message. You can use this as you wish.

DESCRIPTION

You can use this function to make sure that all outstanding status messages have been sent to your program. For example, if you call **kern_loader_status_port**(), you might want to call **kern_loader_ping**() at some point afterward. When you receive the ping message, you'll know that you have received all status messages that were queued before you called **kern_loader_ping**().

Another reason to call **kern_loader_ping**() is to check whether **kern_loader** has fallen into an unresponsive state.

You must implement the ping message yourself, as described in Chapter 3. **kern_loader_ping**() returns a value indicating whether the message was successfully sent to *ping_port*.

EXAMPLE

```
        r = kern_loader_look_up(&kl_port);
        if (r != KERN_SUCCESS) {
            mach_error("kl_util: can't find kernel loader", r);
            exit(1);
        }

        r = port_allocate(task_self(), &reply_port);
        if (r != KERN_SUCCESS) {
            mach_error("kl_util: can't allocate reply port", r);
            exit(1);
        }

        /* Create a thread to listen on reply_port. */
        cthread_detach(cthread_fork((cthread_fn_t)ping_thread,
            (any_t)reply_port));

        /* . . . */

        /* Get a ping message sent to the reply port. */
        r=kern_loader_ping(kl_port, reply_port, 0);

        /* Wait for ping() to kill us.  Exit if we receive a signal. */
        pause();
        exit(0);
}
```

```
kern_loader_reply_t kern_loader_reply = {
    0,        /* argument to pass to function */
    0,        /* timeout for rpc return msg_send */
    0,        /* string function */
    ping,     /* ping function */
    0         /* log_data function */
};

void ping_thread(port_name_t port)
{
    char          msg_buf[KERN_LOADER_REPLY_INMSG_SIZE];
    msg_header_t *msg = (msg_header_t *)msg_buf;
    kern_return_t r;

    /* message handling loop */
    while (TRUE) {
        /* Receive the next message in the queue. */
        msg->msg_size = KERN_LOADER_REPLY_INMSG_SIZE;
        msg->msg_local_port = port;
        r = msg_receive(msg, MSG_OPTION_NONE, 0);
        if (r != KERN_SUCCESS)
            break;

        /* Handle the message we just received. */
        kern_loader_reply_handler(msg, &kern_loader_reply);
    }

    /* We get here only if msg_receive returned an error. */
    mach_error("ping_thread receive", r);
    exit(1);
}

/* This function is called after a kern_loader_ping. */
kern_return_t ping (void *arg, int id)
{
    exit(0);    /* Kill this process. */
}
```

SEE ALSO

kern_loader_reply_handler()

kern_loader_reply_handler()

SUMMARY Handle a message from the kernel-server loader

SYNOPSIS

#include <mach.h>
#include <kernserv/kern_loader_reply_handler.h>

kern_return_t **kern_loader_reply_handler**(msg_header_t **msg*,
kern_loader_reply_t **kern_loader_reply*)

ARGUMENTS

msg: The message you just received from the kernel-server loader.

kern_loader_reply: A pointer to the structure that specifies which of your functions handle each type of reply from the kernel-server loader.

DESCRIPTION

You must use this function if you use **kern_loader_ping**(), **kern_loader_get_log**(), or **kern_loader_status_port**(). Those routines cause an asynchronous reply message from the kernel-server loader; this reply message must be passed to **kern_loader_reply_handler**(), which forwards the message's data to your *ping_func()*, *log_func()*, or *string_func()* function.

This function returns the value that is returned by your *ping_func()*, *log_func()*, or *string_func()* function. See Chapter 3 for more information on implementing these functions.

EXAMPLE

```
kern_loader_reply_t kern_loader_reply = {
    0,            /* argument to pass to function */
    0,            /* timeout for rpc return msg_send */
    0,            /* string function */
    0,            /* ping function */
    log_data      /* log_data function */
};

void listen(port_name_t port)
{
    char          msg_buf[KERN_LOADER_REPLY_INMSG_SIZE];
   msg_header_t *msg = (msg_header_t *)msg_buf;
    kern_return_t r;
```

```
    while (1) {
        /* Receive the next message in the queue. */
        msg->msg_size = KERN_LOADER_REPLY_INMSG_SIZE;
        msg->msg_local_port = port;
        r = msg_receive(msg, MSG_OPTION_NONE, 0);

        if (r != KERN_SUCCESS) {
            mach_error("listen msg_receive", r);
            exit(1);
        }

        /* Handle the message we just received. */
        kern_loader_reply_handler(msg, &kern_loader_reply);
    }
}
```

SEE ALSO

kern_loader_get_log(), **kern_loader_ping()**, **kern_loader_status_port()**

kern_loader_server_com_port()

SUMMARY Get a loadable kernel server's communication port

SYNOPSIS

#include <mach.h>
#include <kernserv/kern_loader.h>

kern_return_t **kern_loader_server_com_port**(port_t *loader_port*, port_t *task_port*, server_name_t *server_name*, port_t **server_com_port*)

ARGUMENTS

loader_port: **kern_loader**'s port, obtained from calling **kern_loader_look_up()**.

task_port: The kernel port for the task in which the loadable kernel server is executing. This is returned by **kern_loader_server_task_port()**.

server_name: The string associated with the server. For example, the MIDI driver's name is "midi".

server_com_port: Returns the server's communication port.

DESCRIPTION

A loadable kernel server's communication port is used for logging-related functions, such as **kern_loader_get_log()** and **kern_loader_log_level()**.

EXAMPLE

```
/* Get kern_loader's port. */
r = kern_loader_look_up(&kl_port);
if (r != KERN_SUCCESS) {
    mach_error("Can't find kernel loader", r);
    exit(1);
}

/* Get the kernel's task port. */
r = task_by_unix_pid(task_self(), 0, &kern_port);
if (r != KERN_SUCCESS) {
    mach_error("Error looking up kernel's port", r);
    exit(1);
}

/* Get the server's com port. */
r = kern_loader_server_com_port(kl_port, kern_port, MYDRIVER_NAME,
    &server_com_port);
if (r != KERN_SUCCESS) {
    kern_loader_error("Error looking up server com port", r);
    exit(1);
}
```

RETURN

KERN_SUCCESS: The call succeeded.

KERN_LOADER_NO_PERMISSION: *task_port* wasn't the server's task port.

KERN_LOADER_UNKNOWN_SERVER: *server_name* wasn't recognized.

SEE ALSO

kern_loader_get_log(), **kern_loader_log_level()**, **kern_loader_look_up()**

kern_loader_server_info()

SUMMARY Get information about a kernel server

SYNOPSIS

#include <mach.h>
#include <kernserv/kern_loader_reply.h>

kern_return_t **kern_loader_server_info**(port_t *loader_port*, port_t *task_port*,
server_name_t *server_name*, server_state_t **server_state*,
vm_address_t **load_address*, vm_size_t **load_size*, server_reloc_t *relocatable*,
server_reloc_t *loadable*, port_name_array_t **port_list*,
unsigned int **port_list_count*, port_name_string_array_t **port_names*,
unsigned int **port_names_count*, boolean_array_t **advertised*,
unsigned int **advertised_count*)

ARGUMENTS

loader_port: **kern_loader**'s port, obtained from calling **kern_loader_look_up**().

task_port: The kernel's task port, obtained using **task_by_unix_pid**(). Specify PORT_NULL if you don't want to have data returned in *port_list*.

server_name: The string associated with the server. For example, the MIDI driver's name is "midi".

server_state: Returns the state of the loadable kernel server. The value is one of the following: Zombie, Allocating, Allocated, Loading, Loaded, Unloading, Deallocated (as defined in the header file **kernserv/kern_loader_types.h**).

load_address: Returns the address in the kernel address space that the kernel server starts at.

load_size: Returns the number of bytes used by the kernel server. *load_address* + *load_size* – 1 is the last address in the kernel map that's used by the kernel server's text and data.

relocatable: Returns the location of the relocatable object file for this server.

loadable: Returns the location of the loadable object file (if any) for this server. This is a file created by **kern_loader** from *relocatable* and then loaded against **/mach**.

port_list: Returns the ports that the kernel server has made available to **kern_loader**, using the HMAP or SMAP load command. If you don't pass in the correct *task_port*, this list will consist of null ports.

port_list_count: Returns the number of ports that the kernel server has made available to **kern_loader**. Even if *task_port* isn't valid, and so nothing is returned in *port_list*, this argument holds the number of ports that would have been returned.

port_names: Returns the strings associated with the ports in *port_list*.

port_names_count: Returns the number of names in *port_names*. This number is the same as *port_list_count*.

advertised: For each entry in *port_list*, returns true if the port is advertised with the Network Name Server.

advertised_count: Returns the number of entries in *advertised*. This number is the same as *port_list_count*.

DESCRIPTION

kern_loader_server_info() returns information about a particular loadable kernel server.

EXAMPLE

```
/* Get kern_loader's port. */
error=kern_loader_look_up(&loader_port);
if (error != KERN_SUCCESS) {
    kern_loader_error("Couldn't find kern_loader's port", error);
    exit(1);
}

/* Get the information. */
error=kern_loader_server_info(loader_port, PORT_NULL, "midi",
    &server_state, &load_addr, &load_size, relocatable, loadable,
    (port_name_array_t *)&scratch, &count, &port_names, &count,
    &advertised, &count);
if (error != KERN_SUCCESS)
    kern_loader_error("Couldn't get info on midi", error);
else
    printf("The relocatable object file is located at:  %s\n",
    relocatable);
```

RETURN

KERN_SUCCESS: The call succeeded.

KERN_LOADER_UNKNOWN_SERVER: *server_name* wasn't recognized.

SEE ALSO

kern_loader_look_up(), **kern_loader_server_list**()

kern_loader_server_list()

SUMMARY Get the names of all known servers

SYNOPSIS

#include <mach.h>
#include <kernserv/kern_loader.h>

kern_return_t **kern_loader_server_list**(port_t *loader_port*,
server_name_array_t **server_names*, unsigned int **server_names_count*)

ARGUMENTS

loader_port: **kern_loader**'s port, obtained from calling **kern_loader_look_up**().

server_names: Returns an array whose entries are the strings associated with all known servers.

server_names_count: Returns the number of entries in *server_names*.

DESCRIPTION

Use this function to get the string associated with each loadable kernel server that **kern_loader** is keeping track of.

EXAMPLE

```
r = kern_loader_look_up(&loader_port);
if (r != KERN_SUCCESS) {
    kern_loader_error("Couldn't get loader_port", r);
    exit(1);
}

r = kern_loader_server_list(loader_port, &server_names, &count);
if (r != KERN_SUCCESS)
    kern_loader_error("Couldn't get the list", r);
else
    for (i=0; i<count; i++)
        printf("Server %d:  %s\n", i, server_names[i]);
```

RETURN

KERN_SUCCESS: The call succeeded.

SEE ALSO

kern_loader_look_up(), **kern_loader_server_info()**

kern_loader_server_task_port()

SUMMARY Get the task port of a loadable kernel server

SYNOPSIS

#include <mach.h>
#include <kernserv/kern_loader.h>

kern_return_t **kern_loader_server_task_port**(port_t *loader_port*, port_t *kernel_port*, server_name_t *server_name*, port_t **server_task_port*)

ARGUMENTS

loader_port: **kern_loader**'s port, obtained from calling **kern_loader_look_up()**.

kernel_port: The kernel's task port.

server_name: The string associated with a loaded server. For example, the MIDI driver's name is "midi".

server_task_port: Returns the kernel port for the task in which the loadable kernel server is executing.

DESCRIPTION

This function returns the task port of the server. Each loadable kernel server currently executes in its own task, but uses the kernel address space. The port returned by **kern_loader_server_task_port()** isn't necessary for any other kernel-server loader functions, but might be useful for gathering debugging information.

EXAMPLE

```
port_t              loader_port, kernel_task, server_port;
kern_return_t       r;
port_name_array_t   names;
unsigned int        i, names_count, types_count;
port_type_array_t   types;

r = kern_loader_look_up(&loader_port);
if (r != KERN_SUCCESS) {
    kern_loader_error("Couldn't get loader_port", r);
    exit(1);
}

r = task_by_unix_pid(task_self(), 0, &kernel_task);
if (r != KERN_SUCCESS) {
    kern_loader_error("Couldn't get kernel_task", r);
    exit(2);
}

r = kern_loader_server_task_port(loader_port, kernel_task, "midi",
    &server_port);
if (r != KERN_SUCCESS)
    kern_loader_error("Couldn't get the server port", r);
else
    printf("Midi's task port is %d\n", server_port);

r = port_names((task_t)server_port, &names, &names_count, &types,
    &types_count);
if (r != KERN_SUCCESS)
    mach_error("Error calling port_names()", r);
else
    for (i=0; i<names_count; i++)
        printf("Port %d has type %d\n", names[i], types[i]);
```

RETURN

KERN_SUCCESS: The call succeeded.

KERN_LOADER_NO_PERMISSION: *task_port* wasn't the server's task port.

KERN_LOADER_UNKNOWN_SERVER: *server_name* wasn't recognized.

SEE ALSO

kern_loader_look_up(), **kern_loader_server_com_port()**

kern_loader_status_port()

SUMMARY Specify a port for **kern_loader** to send status to

SYNOPSIS

#include <mach.h>
#include <kernserv/kern_loader.h>

kern_return_t **kern_loader_status_port**(port_t *loader_port*, port_t *listen_port*)

ARGUMENTS

loader_port: **kern_loader**'s port, obtained from calling **kern_loader_look_up**().

listen_port: The port we want to receive the status messages on.

DESCRIPTION

Use this function to get general status from **kern_loader**. You can receive many reply messages as the result of just one call to **kern_loader_status_port**().

You must define the function that handles status reply messages, as described in Chapter 3. This function receives the status string along with its priority, using the priorities defined in the header file **sys/syslog.h** (LOG_EMERG, LOG_ALERT, and so on).

EXAMPLE

```
r = kern_loader_look_up(&kl_port);
if (r != KERN_SUCCESS) {
    mach_error("kl_util: can't find kernel loader", r);
    exit(1);
}

r = port_allocate(task_self(), &status_port);
if (r != KERN_SUCCESS) {
    mach_error("kl_util: can't allocate reply port", r);
    exit(1);
}

/* Get generic status messages on this port. */
r = kern_loader_status_port(kl_port, status_port);
if (r != KERN_SUCCESS) {
    kern_loader_error("Couldn't specify status port", r);
        exit(1);
}
```

```
    /* Create a thread to listen on status_port. */
    cthread_detach(cthread_fork((cthread_fn_t)receive_thread,
        (any_t)status_port));

    /*
     * Sleep for a while so we can enter kl_util commands at a shell
     * window.  The output of all commands (except status lines from
     * kl_util -s) will show up in both the window that's running this
     * program and in the window that's running kl_util.  (kl_util
     * also has a status port registered.)
     */
    sleep(30);

    exit(0);
}

kern_loader_reply_t kern_loader_reply = {
    0,                /* argument to pass to function */
    0,                /* timeout for rpc return msg_send */
    print_string,     /* string function */
    0,                /* reply_ping function */
    0                 /* log_data function */
};

void receive_thread(port_name_t port)
{
    char          msg_buf[KERN_LOADER_REPLY_INMSG_SIZE];
    msg_header_t *msg = (msg_header_t *)msg_buf;
    kern_return_t r;

    /* message handling loop */
    while (TRUE) {
        /* Receive the next message in the queue. */
        msg->msg_size = KERN_LOADER_REPLY_INMSG_SIZE;
        msg->msg_local_port = port;
        r = msg_receive(msg, MSG_OPTION_NONE, 0);
        if (r != KERN_SUCCESS)
            break;

        /* Handle the message we just received. */
        kern_loader_reply_handler(msg, &kern_loader_reply);
    }

    /* We get here only if msg_receive returned an error. */
    mach_error("receive_thread receive", r);
    exit(1);
}
```

```
/* Called every time kern_loader has status to report. */
kern_return_t print_string(void *arg, printf_data_t string,
    u_int string_count, int level)
{
    /* If the string is empty, return. */
    if (string_count == 0 || !string)
        return KERN_SUCCESS;

    /* Print the string we were passed, with our prefix. */
    printf("print_string: %s", string);

    return KERN_SUCCESS;
}
```

RETURN

KERN_SUCCESS: The call succeeded.

SEND_INVALID_PORT: *listen_port* isn't a valid port.

SEE ALSO

kern_loader_look_up(), **kern_loader_reply_handler()**

kern_loader_unload_server()

SUMMARY Unload a server

SYNOPSIS

#include <mach.h>
#include <kernserv/kern_loader.h>

kern_return_t **kern_loader_unload_server**(port_t *loader_port*, port_t *task_port*, server_name_t *server_name*)

ARGUMENTS

loader_port: **kern_loader**'s port, obtained from calling **kern_loader_look_up()**.

task_port: The kernel's task port, obtained using **task_by_unix_pid()**.

server_name: The string associated with the server. For example, the MIDI driver's name is "midi".

DESCRIPTION

Use this function to unload a running loadable kernel server, leaving it allocated.

EXAMPLE

```
r = kern_loader_look_up(&loader_port);
if (r != KERN_SUCCESS) {
    kern_loader_error("Couldn't get loader_port", r);
    exit(1);
}

r = task_by_unix_pid(task_self(), 0, &kernel_task);
if (r != KERN_SUCCESS) {
    kern_loader_error("Couldn't get kernel_task", r);
    exit(2);
}

r = kern_loader_unload_server(loader_port, kernel_task,
    "NextDimension");
if (r != KERN_SUCCESS)
    kern_loader_error("Couldn't unload the server", r);
```

RETURN

KERN_SUCCESS: The server was successfully unloaded.

KERN_LOADER_SERVER_UNLOADED: The server was already unloaded.

KERN_LOADER_NO_PERMISSION: *task_port* wasn't the kernel's task port. (Make sure **task_by_unix_pid**() is called by a process with superuser permission.)

KERN_LOADER_UNKNOWN_SERVER: *server_name* wasn't recognized.

SEE ALSO

kern_loader_load_server(), **kern_loader_look_up**()

Index

NeXT Computer, Inc.
900 Chesapeake Drive
Redwood City, CA 94063

Printed in U.S.A.
2913.00
12/90

Text printed on
recycled paper