Appendix B

THE MACH SYSTEM

The Mach operating system is designed to incorporate the many recent innovations in operating-system research to produce a fully functional, technically advanced system. Unlike UNIX, which was developed without regard for multiprocessing, Mach incorporates multiprocessing support throughout. Its multiprocessing support is also exceedingly flexible, ranging from shared memory systems to systems with no memory shared between processors. Mach is designed to run on computer systems ranging from one to thousands of processors. In addition, Mach is easily ported to many varied computer architectures. A key goal of Mach is to be a distributed system capable of functioning on heterogeneous hardware.

Although many experimental operating systems are being designed, built, and used, Mach is better able to satisfy the needs of the masses than the others are because it offers full compatibility with UNIX 4.3BSD. As such, it provides a unique opportunity for us to compare two functionally similar, but internally dissimilar, operating systems. The order and contents of the presentation of Mach is different from that of UNIX to reflect the differing emphasis of the two systems. There is no section on the user interface, because that component is similar in 4.3BSD when running the BSD server. As we shall see, Mach provides the ability to layer emulation of other operating systems as well, and they can even run concurrently.

B.1 ■ History

Mach traces its ancestry to the Accent operating system developed at Carnegie Mellon University (CMU). Although Accent pioneered a number of novel operating system concepts, its utility was limited by its inability to execute UNIX applications and its strong ties to a single hardware architecture that made it difficult to port. Mach's communication system and philosophy are derived from Accent, but many other significant portions of the system (for example, the virtual memory system, task and thread management) were developed from scratch. An important goal of the Mach effort was support for multiprocessors.

Mach's development followed an evolutionary path from BSD UNIX systems. Mach code was initially developed inside the 4.2BSD kernel, with BSD kernel components being replaced by Mach components as the Mach components were completed. The BSD components were updated to 4.3BSD when that became available. By 1986, the virtual memory and communication subsystems were running on the DEC VAX computer family, including multiprocessor versions of the VAX. Versions for the IBM RT/PC and for SUN 3 workstations followed shortly. 1987 saw the completion of the Encore Multimax and Sequent Balance multiprocessor versions, including task and thread support, as well as the first official releases of the system, Release 0 and Release 1.

Through Release 2, Mach provides compatibility with the corresponding BSD systems by including much of BSD's code in the kernel. The new features and capabilities of Mach make the kernels in these releases larger than the corresponding BSD kernels. Mach 3 (Figure B.1) moves the BSD code outside of the kernel, leaving a much smaller *microkernel*. This system implements only basic Mach features in the kernel; all UNIX-specific code has been evicted to run in user-mode servers. Excluding UNIX-specific code from the kernel allows replacement of BSD with another operating system, or the simultaneous execution of multiple operating-system interfaces on top of the microkernel. In addition to BSD, user-mode implementations have been developed for DOS, the Macintosh operating system, and OSF/1. This approach has similarities to the *virtual-machine* concept, but the virtual machine is defined by software (the Mach kernel interface), rather than by hardware. As of Release 3.0, Mach

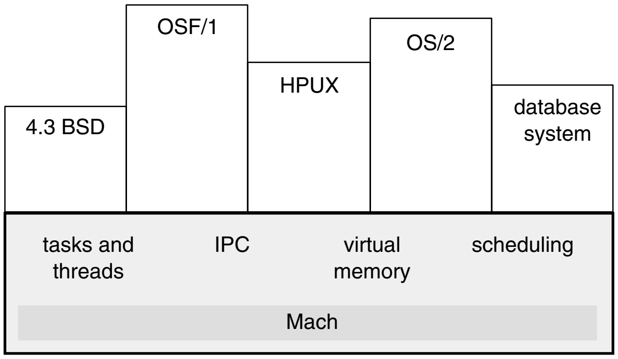

Figure B.1 Mach 3 structure.

became available on a wide variety of systems, including single-processor SUN, Intel, IBM, and DEC machines, and multiprocessor DEC, Sequent, and Encore systems.

Mach was propelled into the forefront of industry attention when the Open Software Foundation (OSF) announced in 1989 that it would use Mach 2.5 as the basis for its new operating system, OSF/1. The initial release of OSF/1 occurred a year later, and now competes with UNIX System V, Release 4, the operating system of choice among *UNIX International (UI)* members. OSF members include key technological companies such as IBM, DEC, and HP. Mach 2.5 is also the basis for the operating system on the *NeXT* workstation, the brainchild of Steve Jobs, of Apple Computer fame. OSF is evaluating Mach 3 as the basis for a future operating-system release, and research on Mach continues at CMU and OSF, and elsewhere.

B.2 ■ Design Principles

The Mach operating system was designed to provide basic mechanisms that most current operating systems lack. The goal is to design an operating system that is BSD compatible and, in addition, excels in the following areas.

- Support for diverse architectures, including multiprocessors with varying degrees of shared memory access: Uniform Memory Access (UMA), Non-Uniform Memory Access (NUMA), and No Remote Memory Access (NORMA)

- Ability to function with varying intercomputer network speeds, from wide-area networks to high-speed local-area networks and tightly coupled multiprocessors

- Simplified kernel structure, with a small number of abstractions; in turn these abstractions are sufficiently general to allow other operating systems to be implemented on top of Mach

- Distributed operation, providing network transparency to clients and an object-oriented organization both internally and externally

- Integrated memory management and interprocess communication, to provide both efficient communication of large numbers of data, as well as communication-based memory management

- Heterogeneous system support, to make Mach widely available and interoperable among computer systems from multiple vendors

The designers of Mach have been heavily influenced by BSD (and by UNIX in general), whose benefits include

- A simple programmer interface, with a good set of primitives and a consistent set of interfaces to system facilities
- Easy portability to a wide class of uniprocessors
- An extensive library of utilities and applications
- The ability to combine utilities easily via pipes

Of course, BSD was seen as having several drawbacks that need to be redressed:

- A kernel that has become the repository of many redundant features—and that consequently is difficult to manage and modify
- Original design goals that made it difficult to provide support for multiprocessors, distributed systems, and shared program libraries; for instance, because the kernel was designed for uniprocessors, it has no provisions for locking code or data that other processors might be using
- Too many fundamental abstractions, providing too many similar, competing means to accomplish the same task

It should be clear that the development of Mach continues to be a huge undertaking. The benefits of such a system are equally large, however. The operating system runs on many existing uni- and multiprocessor architectures, and can be easily ported to future ones. It makes research easier, because computer scientists can add features via user-level code, instead of having to write their own tailor-made operating system. Areas of experimentation include operating systems, databases, reliable distributed systems, multiprocessor languages, security, and distributed artificial intelligence. In its current instantiation, the Mach system is usually as efficient as are other major versions of UNIX when performing similar tasks.

B.3 ■ System Components

To achieve the design goals of Mach, the developers reduced the operating-system functionality to a small set of basic abstractions, out of which all other functionality can be derived. The Mach approach is to place as little as possible within the kernel, but to make what is there powerful enough that all other features can be implemented at user level.

Mach's design philosophy is to have a simple, extensible kernel, concentrating on communication facilities. For instance, all requests to the kernel, and all data movement among processes, are handled through one communication mechanism. By limiting all data operations to one mechanism, Mach is able to provide systemwide protection to its users by protecting the communications mechanism. Optimizing this one communications path can result in significant performance gains, and is simpler than trying to optimize several paths. Mach is extensible, because many traditionally kernel-based functions can be implemented as user-level servers. For instance, all pagers (including the default pager) can be implemented externally and called by the kernel for the user.

Mach is an example of an *object-oriented* system where the data and the operations that manipulate that data are encapsulated into an abstract object. Only the operations of the object are able to act on the entities defined in it. The details of how these operations are implemented are hidden, as are the internal data structures. Thus, a programmer can use an object only by invoking its defined, exported operations. A programmer can change the internal operations without changing the interface definition, so changes and optimizations do not affect other aspects of system operation. The object-oriented approach supported by Mach allows objects to reside anywhere in a network of Mach systems, transparent to the user. The *port* mechanism, discussed later in this section, makes all of this possible.

Mach's primitive abstractions are the heart of the system, and are as follows:

- A **task** is an execution environment that provides the basic unit of resource allocation. A task consists of a virtual address space and protected access to system resources via ports. A task may contain one or more threads.

- A **thread** is the basic unit of execution, and must run in the context of a task (which provides the address space). All threads within a task share the tasks' resources (ports, memory, and so on). There is no notion of a ″process″ in Mach. Rather, a traditional process would be implemented as a task with a single thread of control.

- A **port** is the basic object reference mechanism in Mach, and is implemented as a kernel-protected communication channel. Communication is accomplished by sending messages to ports; messages are queued at the destination port if no thread is immediately ready to receive them. Ports are protected by kernel-managed capabilities, or *port rights*; a task must have a port right to send a message to a port. The programmer invokes an operation on an object by *sending* a message to a port associated with that object. The object being represented by a port *receive*s the messages.

- A **port set** is a group of ports sharing a common message queue. A thread can receive messages for a port set, and thus service multiple ports. Each received message identifies the individual port (within the set) that it was received from; the receiver can use this to identify the object referred to by the message.

- A **message** is the basic method of communication between threads in Mach. It is a typed collection of data objects; for each object, it may contain the actual data or a pointer to out-of-line data. Port rights are passed in messages; passing port rights in messages is the only way to move them among tasks. (Passing a port right in shared memory does not work, because the Mach kernel will not permit the new task to use a right obtained in this manner.)

- A **memory object** is a source of memory; tasks may access it by mapping portions (or the entire object) into their address spaces. The object may be managed by a user-mode *external memory manager*. One example is a file managed by a file server; however, a memory object can be any object for which memory-mapped access makes sense. A mapped buffer implementation of a UNIX pipe is one example.

Figure B.2 illustrates these abstractions, which we shall elaborate in the remainder of this chapter.

An unusual feature of Mach, and a key to the system's efficiency, is the blending of memory and interprocess-communication features. Whereas some other distributed systems (such as Solaris, with its NFS features) have special-purpose extensions to the file system to extend it over a network, Mach provides a general-purpose, extensible merger of memory and messages at the heart of its kernel. This feature not only allows Mach to be used for distributed

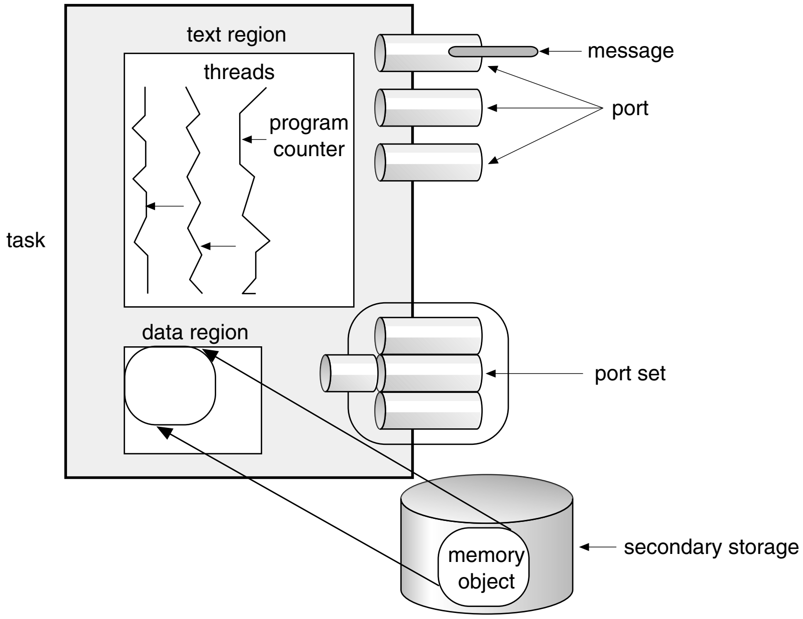

Figure B.2 Mach's basic abstractions.

and parallel programming, but also helps in the implementation of the kernel itself.

Mach connects memory management and communication (IPC) by allowing each to be used in the implementation of the other. Memory management is based on the use of *memory objects*. A memory object is represented by a port (or ports), and IPC messages are sent to this port to request operations (for example, *pagein, pageout*) on the object. Because IPC is used, memory objects may reside on remote systems and be accessed transparently. The kernel caches the contents of memory objects in local memory. Conversely, memory-management techniques are used in the implementation of message passing. Where possible, Mach passes messages by moving pointers to shared memory objects, rather than by copying the object itself.

IPC tends to involve considerable system overhead and is generally less efficient than is communication accomplished through shared memory, for intrasystem messages. Because Mach is a message-based kernel, it is important that message handling be carried out efficiently. Most of the inefficiency of message handling in traditional operating systems is due to either the copying of messages from one task to another (if the message is intracomputer) or low network transfer speed (for intercomputer messages). To solve these problems, Mach uses virtual-memory remapping to transfer the contents of large messages. In other words, the message transfer modifies the receiving task's address space to include a copy of the message contents. *Virtual copy*, or *copy-on-write,* techniques are used to avoid or delay the actual copying of the data. There are several advantages to this approach:

- Increased flexibility in memory management to user programs

- Greater generality, allowing the virtual copy approach to be used in tightly and loosely coupled computers

- Improved performance over UNIX message passing

- Easier task migration; because ports are location independent, a task and all its ports can be moved from one machine to another; all tasks that previously communicated with the moved task can continue to do so because they reference a task by only its ports and communicate via messages to these ports

We shall detail the operation of process management, IPC, and memory management. Then, we shall discuss Mach's chameleonlike ability to support multiple operating-system interfaces.

B.4 ■ Process Management

A *task* can be thought of as a traditional process that does not have an instruction pointer or a register set. A task contains a virtual address space, a set of port rights, and accounting information. A task is a passive entity that does nothing unless it has one or more *thread*s executing in it.

B.4.1 Basic Structure

A task containing one thread is similar to a UNIX process. Just as a **fork** system call produces a new UNIX process, Mach creates a new task to emulate this behavior. The new task's memory is a duplicate of the parent's address space, as dictated by the *inheritance attributes* of the parent's memory. The new task contains one thread, which is started at the same point as the creating **fork** call in the parent. Threads and tasks may also be suspended and resumed.

Threads are especially useful in server applications, which are common in UNIX, since one task can have multiple threads to service multiple requests to the task. They also allow efficient use of parallel computing resources. Rather than having one process on each processor (with the corresponding performance penalty and operating-system overhead), a task may have its threads spread among parallel processors. Threads also add efficiency to user-level programs. For instance, in UNIX, an entire process must wait when a page fault occurs, or when a system call is executed. In a task with multiple threads, only the thread that causes the page fault or executes a system call is delayed; all other threads continue executing. Because Mach has kernel-supported threads (see Chapter 5), the threads have some cost associated with them. They must have supporting data structures in the kernel, and more complex kernel-scheduling algorithms must be provided. These algorithms and thread states are discussed in Chapter 5.

At the user level, threads may be in one of two states.

- **Running:** The thread is either executing or waiting to be allocated a processor. A thread is considered to be running even if it is blocked within the kernel (waiting for a page fault to be satisfied, for instance).

- **Suspended:** The thread is neither executing on a processor nor waiting to be allocated a processor. A thread can resume its execution only if it is returned to the running state.

These two states are also associated with a task. An operation on a task affects all threads in a task, so suspending a task involves suspending all the threads in it. Task and thread suspensions are separate, independent mechanisms, however, so resuming a thread in a suspended task does not resume the task.

Mach provides primitives from which thread-synchronization tools can be built. This primitives provision is consistent with Mach's philosophy of providing minimum yet sufficient functionality in the kernel. The Mach IPC facility can be used for synchronization, with processes exchanging messages at rendezvous points. Thread-level synchronization is provided by calls to start and stop threads at appropriate times. A *suspend count* is kept for each thread. This count allows multiple suspend calls to be executed on a thread, and only when an equal number of resume calls occur is the thread resumed. Unfortunately, this feature has its own limitation. Because it is an error for a start call to be executed before a stop call (the suspend count would become negative), these routines cannot be used to synchronize shared data access. However, *wait* and *signal* operations associated with semaphores, and used for synchronization, can be implemented via the IPC calls. We discuss this method in Section B.5.

B.4.2 The C Threads Package

Mach provides low-level but flexible routines instead of polished, large, and more restrictive functions. Rather than making programmers work at this low level, Mach provides many higher-level interfaces for programming in C and other languages. For instance, the *C Threads* package provides multiple threads of control, shared variables, mutual exclusion for critical sections, and condition variables for synchronization. In fact, C Threads is one of the major influences of the POSIX P Threads standard, which many operating systems are being modified to support. As a result there are strong similarities between the C Threads and P Threads programming interfaces. The thread-control routines include calls to perform these tasks:

- Create a new thread within a task, given a function to execute and parameters as input. The thread then executes concurrently with the creating thread, which receives a thread identifier when the call returns.
- Destroy the calling thread, and return a value to the creating thread.
- Wait for a specific thread to terminate before allowing the calling thread to continue. This call is a synchronization tool, much like the UNIX **wait** system calls.
- Yield use of a processor, signaling that the scheduler may run another thread at this point. This call is also useful in the presence of a preemptive scheduler, as it can be used to relinquish the CPU voluntarily before the time quantum (scheduling interval) expires if a thread has no use for the CPU.

Mutual exclusion is achieved through the use of spinlocks, as were discussed in Chapter 7. The routines associated with mutual exclusion are these:

- The routine *mutex_alloc* dynamically creates a mutex variable.
- The routine *mutex_free* deallocates a dynamically created mutex variable.

- The routine *mutex_lock* locks a mutex variable. The executing thread loops in a spinlock until the lock is attained. A deadlock results if a thread with a lock tries to lock the same mutex variable. Bounded waiting is not guaranteed by the C Threads package. Rather, it is dependent on the hardware instructions used to implement the mutex routines.

- The routine *mutex_unlock* unlocks a mutex variable, much like the typical *signal* operation of a semaphore.

General synchronization without busy waiting can be achieved through the use of *condition variables*, which can be used to implement a *condition critical region* or a *monitor*, as was described in Chapter 7. A condition variable is associated with a mutex variable, and reflects a Boolean state of that variable. The routines associated with general synchronization are these:

- The routine *condition_alloc* dynamically allocates a condition variable.

- The routine *condition_free* deletes a dynamically created condition variable allocated as result of *condition_alloc*.

- The routine *condition_wait* unlocks the associated mutex variable, and blocks the thread until a *condition_signal* is executed on the condition variable, indicating that the event being waited for may have occurred. The mutex variable is then locked, and the thread continues. A *condition_signal* does not guarantee that the condition still holds when the unblocked thread finally returns from its *condition_wait* call, so the awakened thread must loop, executing the *condition_wait* routine until it is unblocked and the condition holds.

As an example of the C Threads routines, consider the bounded-buffer synchronization problem of Section 7.5.1. The producer and consumer are represented as threads that access the common bounded-buffer pool. We use a mutex variable to protect the buffer while it is being updated. Once we have exclusive access to the buffer, we use condition variables to block the producer thread if the buffer is full, and to block the consumer thread if the buffer is empty. Although this program normally would be written in the C language on a Mach system, we shall use the familiar Pascal-like syntax of previous chapters for clarity. As in Chapter 7, we assume that the buffer consists of *n* slots, each capable of holding one item. The *mutex* semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The *empty* and *full* semaphores count the number of empty and full buffers, respectively. The semaphore *empty* is initialized to the value *n*; the semaphore *full* is initialized to the value 0. The condition variable *nonempty* is true while the buffer has items in it, and *nonfull* is true if the buffer has an empty slot.

The first step includes the allocation of the mutex and condition variables:

```
mutex_alloc(mutex); condition_alloc(nonempty, nonfull);
```

```
repeat
           ...
   produce an item into nextp
           ...
   mutex_lock(mutex);
   while(full)
       condition_wait(nonfull, mutex);
           ...
   add nextp to buffer
           ...
   condition_signal(nonempty);
   mutex_unlock(mutex);
until false;
```

Figure B.3 The structure of the producer process.

The code for the producer thread is shown in Figure B.3; the code for the consumer thread is shown in Figure B.4. When the program terminates, the mutex and condition variables need to be deallocated:

mutex_free(*mutex*); **condition_free**(*nonempty, nonfull*);

B.4.3 The CPU Scheduler

The CPU scheduler for a thread-based multiprocessor operating system is more complex than are its process-based relatives. There are generally more threads in a multithreaded system than there are processes in a multitasking system. Keeping track of multiple processors is also difficult, and is a relatively new area

```
repeat
   mutex_lock(mutex);
   while(empty)
       condition_wait(nonempty, mutex);
           ...
   remove an item from the buffer to nextc
           ...
   condition_signal(nonfull);
   mutex_unlock(mutex);
           ...
   consume the item in nextc
           ...
until false;
```

Figure B.4 The structure of the consumer process.

of research. Mach uses a simple policy to keep the scheduler manageable. Only threads are scheduled, so no knowledge of tasks is needed in the scheduler. All threads compete equally for resources, including time quanta.

Each thread has an associated priority number ranging from 0 through 127, which is based on the exponential average of its usage of the CPU. That is, a thread that recently used the CPU for a large amount of time has the lowest priority. Mach uses the priority to place the thread in one of 32 global run queues. These queues are searched in priority order for waiting threads when a processor becomes idle. Mach also keeps per-processor, or local, run queues. A local run queue is used for threads that are bound to an individual processor. For instance, a device driver for a device connected to an individual CPU must run on only that CPU.

Instead of there being a central dispatcher that assigns threads to processors, each processor consults the local and global run queues to select the appropriate next thread to run. Threads in the local run queue have absolute priority over those in the global queues, because it is assumed that they are performing some chore for the kernel. The run queues (like most other objects in Mach) are locked when they are modified to avoid simultaneous changes by multiple processors. To speed dispatching of threads on the global run queue, Mach maintains a list of idle processors.

Additional scheduling difficulties arise from the multiprocessor nature of Mach. A fixed time quantum is not appropriate because there may be fewer runable threads than there are available processors, for instance. It would be wasteful to interrupt a thread with a context switch to the kernel when that thread's quantum runs out, only to have the thread be placed right back in the running state. Thus, instead of using a fixed-length quantum, Mach varies the size of the time quantum inversely with the total number of threads in the system. It keeps the time quantum over the entire system constant, however. For example, in a system with 10 processors, 11 threads, and a 100-millisecond quantum, a context switch needs to occur on each processor only once per second to maintain the desired quantum.

Of course, there are still complications to be considered. Even relinquishing the CPU while waiting for a resource is more difficult than it is on traditional operating systems. First, a call must be issued by a thread to alert the scheduler that the thread is about to block. This alert avoids race conditions and deadlocks, which could occur when the execution takes place in a multiprocessor environment. A second call actually causes the thread to be moved off the run queue until the appropriate event occurs. There are many other internal thread states that are used by the scheduler to control thread execution.

B.4.4 Exception Handling

Mach was designed to provide a single, simple, consistent exception-handling system, with support for standard as well as user-defined exceptions. To avoid redundancy in the kernel, Mach uses kernel primitives whenever possible. For instance, an exception handler is just another thread in the task in which the exception occurs. Remote procedure call (RPC) messages are used to synchronize the execution of the thread causing the exception (the "victim") and that of the handler, and to communicate information about the exception between the victim and handler. Mach exceptions are also used to emulate the BSD *signal* package, as described later in this section.

Disruptions to normal program execution come in two varieties: internally generated *exceptions* and external *interrupts*. Interrupts are asynchronously generated disruptions of a thread or task, whereas exceptions are caused by the occurrence of unusual conditions during a thread's execution. Mach's general-purpose exception facility is used for error detection and debugger support. This facility is also useful for other reasons, such as taking a core dump of a bad task, allowing tasks to handle their own errors (mostly arithmetic), and emulating instructions not implemented in hardware.

Mach supports two different granularities of exception handling. Error handling is supported by per-thread exception handling, whereas debuggers use per-task handling. It makes little sense to try to debug only one thread, or to have exceptions from multiple threads invoke multiple debuggers. Aside from this distinction, the only other difference between the two types of exceptions lies in their inheritance from a parent task. Taskwide exception-handling facilities are passed from the parent to child tasks, so debuggers are able to manipulate an entire tree of tasks. Error handlers are not inherited, and default to no handler at thread- and task-creation time. Finally, error handlers take precedence over debuggers if the exceptions occur simultaneously. The reason for this approach is that error handlers are normally part of the task, and therefore should execute normally even in the presence of a debugger.

Exception handling proceeds as follows:

- The victim thread causes notification of an exception's occurrence via a *raise* RPC message being sent to the handler.

- The victim then calls a routine to wait until the exception is handled.

- The handler receives notification of the exception, usually including information about the exception, the thread, and the task causing the exception.

- The handler performs its function according to the type of exception. The handler's action involves *clearing* the exception, causing the victim to resume, or *terminating* the victim thread.

To support the execution of BSD programs, Mach needs to support BSD-style signals. Signals provide software generated interrupts and exceptions. Unfortunately, signals are of limited functionality in multithreaded operating systems. The first problem is that, in UNIX, a signal's handler must be a routine in the process receiving the signal. If the signal is caused by a problem in the process itself (for example, a division by zero), the problem cannot be remedied, because a process has limited access to its own context. A second, more troublesome aspect of signals is that they were designed for only single-threaded programs. For instance, it makes no sense for all threads in a task to get a signal, but how can a signal be seen by only one thread?

Because the signal system must work correctly with multithreaded applications for Mach to run 4.3BSD programs, signals could not be abandoned. Producing a functionally correct signal package required several rewrites of the code, however! A final problem with UNIX signals is that they can be lost. This loss occurs when another signal of the same type occurs before the first is handled. Mach exceptions are queued as a result of their RPC implementation.

Externally generated signals, including those sent from one BSD process to another, are processed by the BSD server section of the Mach 2.5 kernel. Their behavior is therefore the same as it is under BSD. Hardware exceptions are a different matter, because BSD programs expect to receive hardware exceptions as signals. Therefore, a hardware exception caused by a thread must arrive at the thread as a signal. So that this result is produced, hardware exceptions are converted to exception RPCs. For tasks and threads that do not make explicit use of the Mach exception-handling facility, the destination of this RPC defaults to an in-kernel task. This task has only one purpose: Its thread runs in a continuous loop, receiving these exception RPCs. For each RPC, it converts the exception into the appropriate signal, which is sent to the thread that caused the hardware exception. It then completes the RPC, clearing the original exception condition. With the completion of the RPC, the initiating thread reenters the run state. It immediately sees the signal and executes its signal-handling code. In this manner, all hardware exceptions begin in a uniform way—as exceptions RPCs. Threads not designed to handle such exceptions, however, receive the exceptions as they would on a standard BSD system—as signals. In Mach 3.0, the signal-handling code is moved entirely into a server, but the overall structure and flow of control is similar to those of Mach 2.5.

B.5 ■ Interprocess Communication

Most commercial operating systems, such as UNIX, provide communication between processes, and between hosts with fixed, global names (internet addresses). There is no location independence of facilities, because any remote system needing to use a facility must know the name of the system providing that facility. Usually, data in the messages are untyped streams of bytes. Mach simplifies this picture by sending messages between location-independent ports. The messages contain typed data for ease of interpretation. All BSD communication methods can be implemented with this simplified system.

The two components of Mach IPC are *ports* and *messages*. Almost everything in Mach is an object, and all objects are addressed via their communications ports. Messages are sent to these ports to initiate operations on the objects by the routines that implement the objects. By depending on only ports and messages for all communication, Mach delivers location independence of objects and security of communication. Data independence is provided by the NetMsgServer task, as discussed later. Mach ensures security by requiring that message senders and receivers have *rights*. A right consists of a port name and a capability (send or receive) on that port, and is much like a capability in object-oriented systems. There can be only one task with receive rights to any given port, but many tasks may have send rights. When an object is created, its creator also allocates a port to represent the object, and obtains the access rights to that port. Rights can be given out by the creator of the object (including the kernel), and are passed in messages. If the holder of a receive right sends that right in a message, the receiver of the message gains the right and the sender loses it. A task may allocate ports to allow access to any objects it owns, or for communication. The destruction of either a port or the holder of the receive right causes the revocation of all rights to that port, and the tasks holding send rights can be notified if desired.

B.5.1 Ports

A port is implemented as a protected, bounded queue within the kernel of the system on which the object resides. If a queue is full, a sender may abort the send, wait for a slot to become available in the queue, or have the kernel deliver the message for it.

There are several system calls to provide the port functionality:

- Allocate a new port in a specified task and give the caller's task all access rights to the new port. The port name is returned.
- Deallocate a task's access rights to a port. If the task holds the receive right, the port is destroyed and all other tasks with send rights are, potentially, notified.
- Get the current status of a task's port.
- Create a backup port, which is given the receive right for a port if the task containing the receive right requests its deallocation (or terminates).

When a task is created, the kernel creates several ports for it. The function *task_self* returns the name of the port that represents the task in calls to the kernel. For instance, for a task to allocate a new port, it would call *port_allocate* with *task_self* as the name of the task that will own the port. Thread creation results in a similar *thread_self* thread kernel port. This scheme is similar to the standard process-id concept found in UNIX. Another port created for a task is returned by *task_notify*, and is the name of the port to which the kernel will send event-notification messages (such as notifications of port terminations).

Ports can also be collected into *port set*s. This facility is useful if one thread is to service requests coming in on multiple ports (for example, for multiple objects). A port may be a member of at most one port set at a time, and, if a port is in a set, it may not be used directly to receive messages. Instead, the message will be routed to the port set's queue. A port set may not be passed in messages, unlike a port. Port sets are objects that serve a purpose similar to the 4.3BSD **select** system call, but they are more efficient.

B.5.2 Messages

A message consists of a fixed-length header and a variable number of typed data objects. The header contains the destination's port name, the name of the reply port to which return messages should be sent, and the length of the message (see Figure B.5). The data in the message (*in-line* data) were limited to less than 8K in Mach 2.5 systems, but Mach 3.0 has no limit. Any data exceeding that limit must be sent in multiple messages, or more likely via reference by a pointer in a message (*out-of-line* data, as we shall describe shortly). Each data section may be a simple type (numbers or characters), port rights, or pointers to out-of-line data. Each section has an associated type, so that the receiver

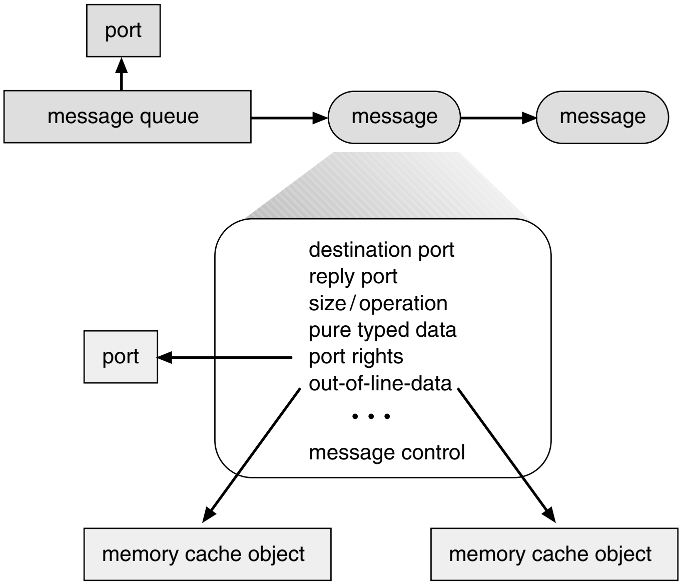

Figure B.5 Mach messages.

can unpack the data correctly even if it uses a byte ordering different from that used by the sender. The kernel also inspects the message for certain types of data. For instance, the kernel must process port information within a message, either by translating the port name into an internal port data structure address, or by forwarding it for processing to the NetMsgServer, as we shall explain.

The use of pointers in a message provides the means to transfer the entire address space of a task in one single message. The kernel also must process pointers to out-of-line data, as a pointer to data in the sender's address space would be invalid in the receiver's—especially if the sender and receiver reside on different systems! Generally, systems send messages by copying the data from the sender to the receiver. Because this technique can be inefficient, especially in the case of large messages, Mach optimizes this procedure. The data referenced by a pointer in a message being sent to a port on the same system are not copied between the sender and the receiver. Instead, the address map of the receiving task is modified to include a copy-on-write copy of the pages of the message. This operation is *much* faster than a data copy, and makes message passing efficient. In essence, message passing is implemented via virtual-memory management.

In Version 2.5, this operation was implemented in two phases. A pointer to a region of memory caused the kernel to map that region of memory into its own space temporarily, setting the sender's memory map to copy-on-write mode to ensure that any modifications did not affect the original version of the data. When a message was received at its destination, the kernel moved its mapping to the receiver's address space, using a newly allocated region of virtual memory within that task.

In Version 3, this process was simplified. The kernel creates a data structure that would be a copy of the region if it were part of an address map. On receipt, this data structure is added to the receiver's map and becomes a copy accessible to the receiver.

The newly allocated regions in a task do not need to be contiguous with previous allocations, so Mach virtual memory is said to be *sparse*, consisting of regions of data separated by unallocated addresses. A full message transfer is shown in Figure B.6.

B.5.3 The NetMsgServer

For a message to be sent between computers, the destination of a message must be located, and the message must be transmitted to the destination. UNIX traditionally leaves these mechanisms to the low-level network protocols, which require the use of statically assigned communication endpoints (for example, the port number for services based on TCP or UDP). One of Mach's tenets is that all objects within the system are location independent, and that the location is transparent to the user. This tenet requires Mach to provide location-

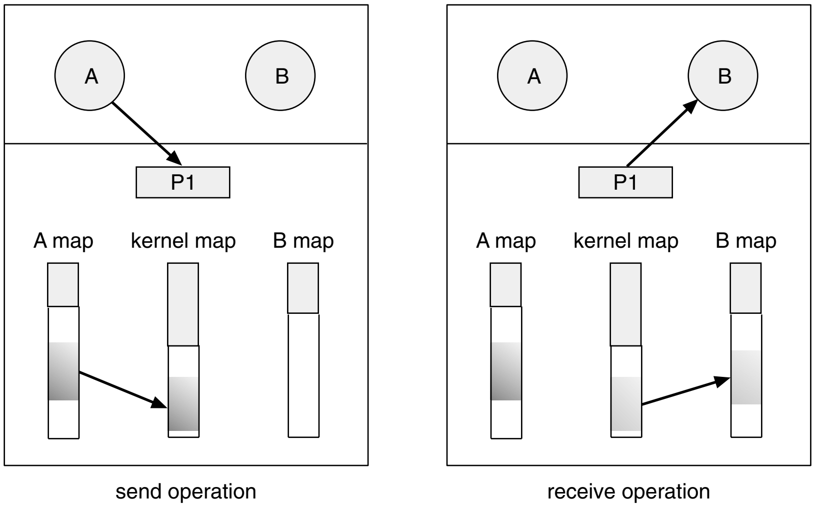

Figure B.6 Mach message transfer.

transparent naming and transport to extend IPC across multiple computers. This naming and transport are performed by the *Network Message Server* or NetMsgServer, a user-level capability-based networking daemon that forwards messages between hosts. It also provides a primitive networkwide name service that allows tasks to register ports for lookup by tasks on any other computer in the network. Mach ports can be transferred only in messages, and messages must be sent to ports; the primitive name service solves the problem of transferring the first port that allows tasks on different computers to exchange messages. Subsequent IPC interactions are fully transparent; the NetMsgServer tracks all rights and out-of-line memory passed in intercomputer messages, and arranges for the appropriate transfers. The NetMsgServers maintain among themselves a distributed database of port rights that have been transferred between computers and of the ports to which these rights correspond.

The kernel uses the NetMsgServer when a message needs to be sent to a port that is not on the kernel's computer. Mach's kernel IPC is used to transfer the message to the local NetMsgServer. The NetMsgServer then uses whatever network protocols are appropriate to transfer the message to its peer on the other computer; the notion of a NetMsgServer is protocol-independent, and NetMsgServers have been built that use various protocols. Of course, the NetMsgServers involved in a transfer must agree on the protocol used. Finally, the NetMsgServer on the destination computer uses that kernel's IPC to send the message to the correct destination task. The ability to extend local IPC transparently across nodes is supported by the use of proxy ports. When a send right is transferred from one computer to another, the NetMsgServer on the destination computer creates a new port, or proxy, to represent the original port at the destination. Messages sent to this proxy are received by the NetMsgServer and are forwarded transparently to the original port; this procedure is one example of how the NetMsgServers cooperate to make a proxy indistinguishable from the original port.

Because Mach is designed to function in a network of heterogeneous systems, it must provide a way to send between systems data that are formatted in a way that is understandable by both the sender and receiver. Unfortunately, computers vary the format in which they store types of data. For instance, an integer on one system might take 2 bytes to store, and the most significant byte might be stored before the least significant one. Another system might reverse this ordering. The NetMsgServer therefore uses the type information stored in a message to translate the data from the sender's to the receiver's format. In this way, all data are represented correctly when they reach their destination.

The NetMsgServer on a given computer accepts RPCs that add, look up, and remove network ports from the NetMsgServer's name service. As a security precaution, a port value provided in an add request must match that in the remove request for a thread to ask for a port name to be removed from the database.

As an example of the NetMsgServer's operation, consider a thread on node A sending a message to a port that happens to be in a task on node B. The program simply sends a message to a port to which it has a send right. The message is first passed to the kernel, which delivers it to its first recipient, the NetMsgServer on node A. The NetMsgServer then contacts (through its database information) the NetMsgServer on node B and sends the message. The NetMsgServer on node B then presents the message to the kernel with the appropriate local port for node B. The kernel finally provides the message to the receiving task when a thread in that task executes a *msg_receive* call. This sequence of events is shown in Figure B.7.

Mach 3.0 provides an alternative to the NetMsgServer as part of its improved support for NORMA multiprocessors. The NORMA IPC subsystem of Mach 3.0 implements functionality similar to the NetMsgServer directly in the Mach kernel, providing much more efficient internode IPC for multicomputers with fast interconnection hardware. For example, the time-consuming copying of messages between the NetMsgServer and the kernel is eliminated. Use of NORMA IPC does not exclude use of the NetMsgServer; the NetMsgServer can still be used to provide MACH IPC service over networks that link a NORMA multiprocessor to other computers. In addition to NORMA IPC, Mach 3.0 also provides support for memory management across a NORMA system, and the ability for a task in such a system to create child tasks on nodes other than its own. These features support the implementation of a single-system-image operating system on a NORMA multiprocessor; the multiprocessor behaves like one large system, rather than like an assemblage of smaller systems (for both users and applications).

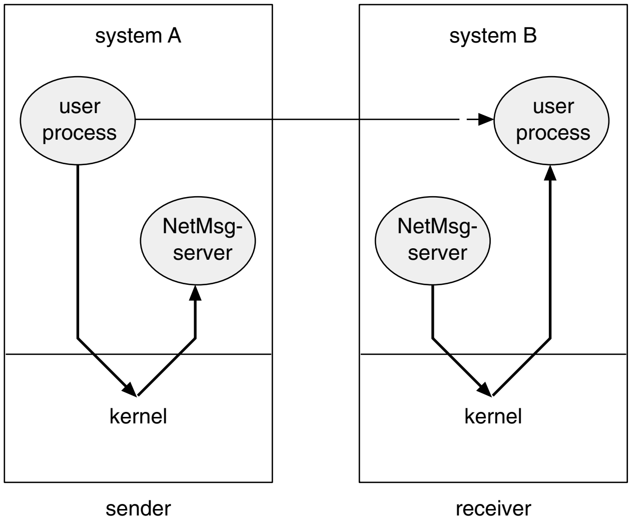

Figure B.7 Network IPC forwarding by NetMsgServer.

B.5.4 Synchronization Through IPC

The IPC mechanism is extremely flexible, and is used throughout Mach. For example, it may be used for thread synchronization. A port may be used as a synchronization variable, and may have *n* messages sent to it for *n* resources. Any thread wishing to use a resource executes a receive call on that port. The thread will receive a message if the resource is available; otherwise, it will wait on the port until a message is available there. To return a resource after use, the thread can send a message to the port. In this regard, receiving is equivalent to the semaphore operation *wait*, and sending is equivalent to *signal*. This method can be used for synchronizing semaphore operations among threads in the same task, but cannot be used for synchronization among tasks, because only one task may have receive rights to a port. For more general-purpose semaphores, a simple daemon may be written that implements the same method.

B.6 ■ Memory Management

Given the object-oriented nature of Mach, it is not surprising that a principle abstraction in Mach is the *memory object*. Memory objects are used to manage secondary storage, and generally represent files, pipes, or other data that are mapped into virtual memory for reading and writing (Figure B.8). Memory objects may be backed by user-level memory managers, which take the place of the more traditional kernel-incorporated virtual-memory pager found in

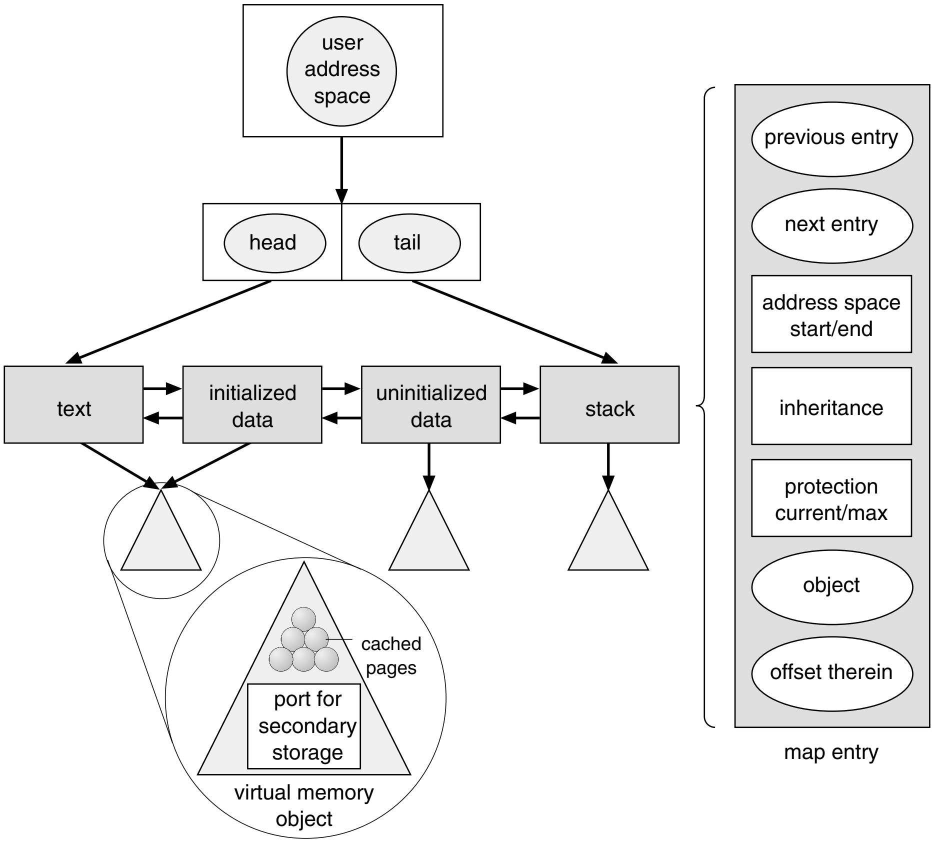

Figure B.8 Mach virtual memory task address map.

other operating systems. In contrast to the traditional approach of having the kernel provide management of secondary storage, Mach treats secondary-storage objects (usually files) as it does all other objects in the system. Each object has a port associated with it, and may be manipulated by messages being sent to its port. Memory objects—unlike the memory-management routines in monolithic, traditional kernels—allow easy experimentation with new memory-manipulation algorithms.

B.6.1 Basic Structure

The virtual address space of a task is generally *sparse*, consisting of many holes of unallocated space. For instance, a memory-mapped file is placed in some set of addresses. Large messages are also transferred as shared memory segments. For each of these segments, a section of virtual-memory address is used to provide the threads with access to the message. As new items are mapped or removed from the address space, holes of unallocated memory appear in the address space.

Mach makes no attempt to compress the address space, although a task may fail (crash) if it has no room for a requested region in its address space. Given that address spaces are 4 gigabytes or more, this limitation is not currently a problem. However, maintaining a regular page table for a 4 gigabyte address space for each task, especially one with holes in it, would use excessive amounts of memory (1 megabyte or more). The key to sparse address spaces is that page-table space is used for only currently allocated regions. When a page fault occurs, the kernel must check to see whether the page is in a valid region, rather than simply indexing into the page table and checking the entry. Although the resulting lookup is more complex, the benefits of reduced memory-storage requirements and simpler address-space maintenance make the approach worthwhile.

Mach also has system calls to support standard virtual-memory functionality, including the allocation, deallocation, and copying of virtual memory. When allocating a new virtual-memory object, the thread may provide an address for the object or may let the kernel choose the address. Physical memory is not allocated until pages in this object are accessed. The object's backing store is managed by the default pager (discussed in Section B.6.2). Virtual-memory objects are also allocated automatically when a task receives a message containing out-of-line data.

Associated system calls return information about a memory object in a task's address space, change the access protection of the object, and specify how an object is to be passed to child tasks at the time of their creation (shared, copy-on-write, or not present).

B.6.2 User-Level Memory Managers

A secondary-storage object is usually mapped into the virtual address space of a task. Mach maintains a cache of memory-resident pages of all mapped objects, as in other virtual-memory implementations. However, a page fault occurring when a thread accesses a nonresident page is executed as a message to the object's port. The concept of a memory object being created and serviced by nonkernel tasks (unlike threads, for instance, which are created and maintained by only the kernel) is important. The end result is that, in the traditional sense, memory can be paged by user-written memory managers. When the object is destroyed, it is up to the memory manager to write back any changed pages to secondary storage. No assumptions are made by Mach about the content or importance of memory objects, so the memory objects are independent of the kernel.

There are several circumstances in which user-level memory managers are insufficient. For instance, a task allocating a new region of virtual memory might not have a memory manager assigned to that region, since it does not represent a secondary-storage object (but must be paged), or a memory manager could fail to perform pageout. Mach itself also needs a memory manager to take care of its memory needs. For these cases, Mach provides a *default memory manager*. The Mach 2.5 default memory manager uses the standard file system to store data that must be written to disk, rather than requiring a separate swap space, as in 4.3BSD. In Mach 3.0 (and OSF/1), the default memory manager is capable of using either files in a standard filesystem or dedicated disk partitions. The default memory manager has an interface similar to that of the user-level ones, but with some extensions to support its role as the memory manager that can be relied on to perform pageout when user-level managers fail to do so.

Pageout policy is implemented by an internal kernel thread, the *pageout daemon*. A paging algorithm based on FIFO with second chance (Section 10.4.5) is used to select pages for replacement. The selected pages are sent to the appropriate manager (either user level or default) for actual pageout. A user-level manager may be more intelligent than the default manager, and may implement a different paging algorithm suitable to the object it is backing (that is, by selecting some other page and forcibly paging it out). If a user-level manager fails to reduce the resident set of pages when asked to do so by the kernel, the default memory manager is invoked and it pages out the user-level manager to reduce the user-level manager's resident set size. Should the user-level manager recover from the problem that prevented it from performing its own pageouts, it will touch these pages (causing the kernel to page them in again), and can then page them out as it sees fit.

If a thread needs access to data in a memory object (for instance, a file), it invokes the *vm_map* system call. Included in this system call is a port which identifies the object, and the memory manager which is responsible for the region. The kernel executes calls on this port when data are to be read or written in that region. An added complexity is that the kernel makes these calls asynchronously, since it would not be reasonable for the kernel to be waiting on a user-level thread. Unlike the situation with pageout, the kernel has no recourse if its request is not satisfied by the external memory manager. The kernel has no knowledge of the contents of an object or of how that object must be manipulated.

Memory managers are responsible for the consistency of the contents of a memory object mapped by tasks on different machines (tasks on a single machine share a single copy of a mapped memory object). Consider a situation in which tasks on two different machines attempt to modify the same page of an object concurrently. It is up to the manager to decide whether these modifications must be serialized. A conservative manager implementing strict memory consistency would force the modifications to be serialized by granting write access to only one kernel at a time. A more sophisticated manager could allow both accesses to proceed concurrently (for example, if the manager knew that the two tasks were modifying distinct areas within the page, and that it could merge the modifications successfully at some future time). Note that most external memory managers written for Mach (for example, those implementing mapped files) do not implement logic for dealing with multiple kernels, due to the complexity of such logic.

When the first *vm_map* call is made on a memory object, the kernel sends a message to the memory manager port passed in the call, invoking the *memory_manager_init* routine, which the memory manager must provide as part of its support of a memory object. The two ports passed to the memory manager are a *control port* and a *name port*. The control port is used by the memory manager to provide data to the kernel (for example, pages to be made resident). Name ports are used throughout Mach. They do not receive messages, but rather are used simply as a point of reference and comparison. Finally, the memory object must respond to a *memory_manager_init* call with a *memory_object_set_attributes* call to indicate that it is ready to accept requests. When all tasks with send rights to a memory object relinquish those rights, the kernel deallocates the object's ports, thus freeing the memory manager and memory object for destruction.

There are several kernel calls that are needed to support external memory managers. The *vm_map* call has already been discussed in the paragraph above. There are also commands to get and set attributes and to provide page-level locking when it is required (for instance, after a page fault has occurred but before the memory manager has returned the appropriate data). Another call is used by the memory manager to pass a page (or multiple pages, if read-ahead is being used) to the kernel in response to a page fault. This call is necessary since the kernel invokes the memory manager asynchronously. There are also several calls to allow the memory manager to report errors to the kernel.

The memory manager itself must provide support for several calls so that it can support an object. We have already discussed *memory_object_init* and others. When a thread causes a page fault on a memory object's page, the kernel sends a *memory_object_data_request* to the memory object's port on behalf of the faulting thread. The thread is placed in wait state until the memory manager either returns the page in a *memory_object_data_provided* call, or returns an appropriate error to the kernel. Any of the pages that have been modified, or any *precious pages* that the kernel needs to remove from resident memory (due to page aging, for instance), are sent to the memory object via *memory_object_data_write*. Precious pages are pages that may not have been modified, but that cannot be discarded as they otherwise would, because the memory manager no longer retains a copy. The memory manager declares these pages to be precious and expects the kernel to return them when they are removed from memory. Precious pages save unnecessary duplication and copying of memory.

Again, there are several other calls for locking, protection information and modification, and the other details with which all virtual memory systems must deal.

In the current version, Mach does not allow external memory managers to affect the page-replacement algorithm directly. Mach does not export the memory-access information that would be needed for an external task to select the least recently used page, for instance. Methods of providing such information are currently under investigation. An external memory manager is still useful for a variety of reasons, however:

- It may reject the kernel's replacement victim if it knows of a better candidate (for instance, MRU page replacement).
- It may monitor the memory object it is backing, and request pages to be paged out before the memory usage invokes Mach's pageout daemon.
- It is especially important in maintaining consistency of secondary storage for threads on multiple processors, as we shall show in Section B.6.3.
- It can control the order of operations on secondary storage, to enforce consistency constraints demanded by database management systems. For example, in transaction logging, transactions must be written to a log file on disk before they modify the database data.
- It can control mapped file access.

B.6.3 Shared Memory

Mach uses shared memory to reduce the complexity of various system facilities, as well as to provide these features in an efficient manner. Shared memory generally provides extremely fast interprocess communication, reduces overhead in file management, and helps to support multiprocessing and database management. Mach does not use shared memory for all these traditional shared-memory roles, however. For instance, all threads in a task share that task's memory, so no formal shared-memory facility is needed within a task. However, Mach must still provide traditional shared memory to support other operating-system constructs, such as the UNIX **fork** system call.

It is obviously difficult for tasks on multiple machines to share memory, and to maintain data consistency. Mach does not try to solve this problem directly; rather, it provides facilities to allow the problem to be solved. Mach supports consistent shared memory only when the memory is shared by tasks running on processors that share memory. A parent task is able to declare which regions of memory are to be *inherited* by its children, and which are to be readable–writable. This scheme is different from copy-on-write inheritance, in which each task maintains its own copy of any changed pages. A writable object is addressed from each task's address map, and all changes are made to the same copy. The threads within the tasks are responsible for coordinating changes to memory so that they do not interfere with one another (by writing to the same location concurrently). This coordination may be done through normal synchronization methods: critical sections or mutual-exclusion locks.

For the case of memory shared among separate machines, Mach allows the use of external memory managers. If a set of unrelated tasks wishes to share a section of memory, the tasks may use the same external memory manager and access the same secondary-storage areas through it. The implementor of this system would need to write the tasks and the external pager. This pager could be as simple or as complicated as needed. A simple implementation would allow no readers while a page was being written to. Any write attempt would cause the pager to invalidate the page in all tasks currently accessing it. The pager would then allow the write and would revalidate the readers with the new version of the page. The readers would simply wait on a page fault until the page again became available. Mach provides such a memory manager: the Network Memory Server (NetMemServer). For multicomputers, the NORMA configuration of Mach 3.0 provides similar support as a standard part of the kernel. This XMM subsystem allows multicomputer systems to use external memory managers that do not incorporate logic for dealing with multiple kernels; the XMM subsystem is responsible for maintaining data consistency among multiple kernels that share memory, and makes these kernels appear to be a single kernel to the memory manager. The XMM subsystem also implements virtual copy logic for the mapped objects that it manages. This virtual copy logic includes both copy-on-reference among multicomputer kernels, and sophisticated copy-on-write optimizations.

B.7 ■ Programmer Interface

There are several levels at which a programmer may work within Mach. There is, of course, the system-call level, which, in Mach 2.5, is equivalent to the 4.3BSD system-call interface. Version 2.5 includes most of 4.3BSD as one thread in the kernel. A BSD system call traps to the kernel and is serviced by this thread on behalf of caller, much as standard BSD would handle it. The emulation is not multithreaded, so it has limited efficiency.

Mach 3.0 has moved from the single-server model to support of multiple servers. It has therefore become a true microkernel without the full features normally found in a kernel. Rather, full functionality can be provided via emulation libraries, servers, or a combination of the two. In keeping with the definition of a microkernel, the emulation libraries and servers run outside the kernel at user level. In this way, multiple operating systems can run concurrently on one Mach 3.0 kernel.

An emulation library is a set of routines that lives in a read-only part of a program's address space. Any operating-system calls the program makes are translated into subroutine calls to the library. Single-user operating systems, such as MS-DOS and the Macintosh operating system, have been implemented solely as emulation libraries. For efficiency reasons, the emulation library lives in the address space of the program needing its functionality, but in theory could be a separate task.

More complex operating systems are emulated through the use of libraries and one or more servers. System calls that cannot be implemented in the library are redirected to the appropriate server. Servers can be multithreaded for improved efficiency. BSD and OSF/1 are implemented as single multithreaded servers. Systems can be decomposed into multiple servers for greater modularity.

Functionally, a system call starts in a task, and passes through the kernel before being redirected, if appropriate, to the library in the task's address space or to a server. Although this extra transfer of control will decrease the efficiency of Mach, this decrease is somewhat ameliorated by the ability for multiple threads to be executing BSD-like code concurrently.

At the next higher programming level is the *C Threads* package. This package is a run-time library that provides a C language interface to the basic Mach threads primitives. It provides convenient access to these primitives, including routines for the forking and joining of threads, mutual exclusion through mutex variables (Section B.4.2), and synchronization through use of condition variables. Unfortunately, it is not appropriate for the C Threads package to be used between systems that share no memory (NORMA systems), since it depends on shared memory to implement its constructs. There is currently no equivalent of C Threads for NORMA systems. Other run-time libraries have been written for Mach, including threads support for other languages.

Although the use of primitives makes Mach flexible, it also makes many programming tasks repetitive. For instance, significant amounts of code are associated with sending and receiving messages in each task that uses messages (which, in Mach, is most tasks). The designers of Mach therefore provide an interface generator (or stub generator) called *MIG*. MIG is essentially a compiler that takes as input a definition of the interface to be used (declarations of variables, types and procedures), and generates the RPC interface code needed to send and receive the messages fitting this definition and to connect the messages to the sending and receiving threads.

B.8 ■ Summary

The Mach operating system is designed to incorporate the many recent innovations in operating-system research to produce a fully functional, technically advanced operating system.

The Mach operating system was designed with the following three critical goals in mind:

- Emulate 4.3BSD UNIX so that the executable files from a UNIX system can run correctly under Mach.

- Have a modern operating system that supports many memory models, and parallel and distributed computing.
- Design a kernel that is simpler and easier to modify than is 4.3BSD.

As we have shown in this chapter, Mach is well on its way to achieving these goals.

Mach 2.5 includes 4.3BSD in its kernel, which provides the emulation needed but enlarges the kernel. This 4.3BSD code has been rewritten to provide the same 4.3 functionality, but to use the Mach primitives. This change allows the 4.3BSD support code to run in user space on a Mach 3.0 system.

Mach uses lightweight processes, in the form of multiple threads of execution within one task (or address space), to support multiprocessing and parallel computation. Its extensive use of messages as the only communications method ensures that protection mechanisms are complete and efficient. By integrating messages with the virtual-memory system, Mach also ensures that messages can be handled efficiently. Finally, by having the virtual-memory system use messages to communicate with the daemons managing the backing store, Mach provides great flexibility in the design and implementation of these memory-object-managing tasks.

By providing low-level, or primitive, system calls from which more complex functions may be built, Mach reduces the size of the kernel while permitting operating-system emulation at the user level, much like IBM's virtual-machine systems.

■ Exercises

B.1 What three features of Mach make it appropriate for distributed processing?

B.2 Name two ways that port sets are useful in implementing parallel programs.

B.3 Consider an application that maintains a database of information, and provides facilities for other tasks to add, delete, and query the database. Give three configurations of ports, threads, and message types that could be used to implement this system. Which is the best? Explain your answer.

B.4 Give the outline of a task that would migrate subtasks (tasks it creates) to other systems. Include information about how it would decide when to migrate tasks, which tasks to migrate, and how the migration would take place.

B.5 Name two types of applications for which you would use the MIG package.

B.6 Why would someone use the low-level system calls, instead of the C Threads package?

B.7 Why are external memory managers not able to replace the internal page-replacement algorithms? What information would need to be made available to the external managers for them to make page-replacement decisions? Why would providing this information violate the principle behind the external managers?

B.8 Why is it difficult to implement mutual exclusion and condition variables in an environment where like-CPUs do not share any memory? What approach and mechanism could be used to make such features available on a NORMA system?

B.9 What are the advantages to rewriting the 4.3BSD code as an external, user-level library, rather than leaving it as part of the Mach kernel? Are there any disadvantages? Explain your answer.

Bibliographical Notes

The Accent operating system was described by Rashid and Robertson [1981]. An historical overview of the progression from an even earlier system, RIG, through Accent to Mach was given by Rashid [1986]. General discussions concerning the Mach model are offered by Tevanian and Smith [1989].

Accetta et al. [1986] presented an overview of the original design of Mach. The Mach scheduler was described in detail by Tevanian et al. [1987a] and Black [1990]. An early version of the Mach shared memory and memory-mapping system was presented by Tevanian et al. [1987b].

The most current description of the C Threads package appears in Cooper and Draves [1987]; MIG was described by Draves et al. [1989]. An overview of these packages' functionality and a general introduction to programming in Mach was presented by Walmer and Thompson [1989] and Boykin et al. [1993].

Black et al. [1988] discussed the Mach exception-handling facility. A multithreaded debugger based on this mechanism was described in Caswell and Black [1989].

A series of talks about Mach sponsored by the OSF UNIX consortium is available on videotape from OSF. Topics include an overview, threads, networking, memory management, many internal details, and some example implementations of Mach. The slides from these talks were given in [OSF 1989].

On systems where USENET News is available (most educational institutions in the United States, and some overseas), the news group *comp.os.mach* is used to exchange information on the Mach project and its components.

An overview of the microkernel structure of Mach 3.0, complete with performance analysis of Mach 2.5 and 3.0 compared to other systems, was given in Black et al. [1992]. Details of the kernel internals and interfaces of Mach 3.0 were provided in Loepere [1992]. Tanenbaum [1992] presented a comparison of Mach and Amoeba. Discussions concerning parallelization in Mach and 4.3BSD are offered by Boykin and Langerman [1990].

Ongoing research was presented at USENIX Mach and Micro-kernel Symposia [USENIX 1990, USENIX 1991, and USENIX 1992b]. Active research areas include virtual memory, real time, and security [McNamee and Armstrong 1990].

Credits

Figs. A.1, A.6, and A.8 reproduced with permission from Open Software Foundation, Inc. Excerpted from Mach Lecture Series, OSF, October 1989, Cambridge, Massachusetts.

Figs. A.1 and A.8 presented by R. Rashid of Carnegie Mellon University and Fig. 20.7 presented by D. Julin of Carnegie Mellon University.

Figs. A.6 from Accetta/Baron/Bolosky/Golub/Rashid/Tevanian/Young, “Mach: a new kernel foundation for UNIX development,” Proceedings of Summer USENIX, June 1986, Atlanta, Georgia. Reprinted with permission of the authors.