MACH: THE MODEL FOR FUTURE UNIX

Will a new, object-oriented kernel change the face of Unix?

Avadis Tevanian Jr. and Ben Smith

Unix is over 20 years old. While the computer hardware for Unix has radically changed since Unix was first designed, the basic concepts of the operating-system kernel have remained the same.

The Mach kernel is designed to take advantage of new computer architectures and provide for the needs of modern programs. It is also a return to the original Unix concept of having only the most essential functions in the kernel—a concept that has been lost in the versions of Unix from the big-iron computer manufacturers, whose kernels can exceed 2 megabytes.

Great Ideas from a Small Team

A small group of researchers at Carnegie Mellon University started the design of Mach in 1984. They wrote the first lines of code in 1985. Originally, Mach was intended to support large-scale parallel computation. However, early on in the design phase, the team decided that designing only for large-scale parallel computation would be of limited life and utility. So they changed the design to make it independent of the hardware architecture. Mach was initially implemented on the VAX-11/780 and now runs on a wide range of hardware, including almost all VAX processors, the IBM RT PC, Sun-3 workstations, the Encore Multimax, the Sequent Balance 21000, various 80386 machines, and the NeXT Computer.

Mach is designed to handle problems associated with both parallel programming on multiprocessor machines and distributed programming, where the work is done by several separate computers communicating over a network. The concept of multiple threads of control executing in parallel within a single task facilitates parallel computing. A capability-based interprocess communication mechanism eases distributed programming. Finally, an extremely powerful virtual memory system allows applications of all sizes to efficiently share the memory resources of these complex architectural designs. With Mach, these very same concepts work equally well on inexpensive, single-processor machines.

To derive Mach, the Carnegie Mellon team extended the model of Unix computing by adding five abstractions: the *task*, the *thread*, the *port*, the *message*, and the *memory object*. Obviously, the language and concepts of object-oriented programming permeate the design of Mach. Many people call Mach an "object-oriented operating system."

The Mach kernel maintains only the most basic services: processor scheduling, interprocess communication, and management of virtual memory. All other services are *service tasks*, independent user-level programs.

Tasks and Threads

Mach splits the traditional Unix abstraction of a process into a task and threads. A task is the environment in which threads run. It includes protected access and control of all system resources, including the CPUs, the physical I/O ports, and memory (virtual and real). The structures associated with files are in the domain of the task. A task address space uses a structured map of memory objects (see below).

A thread is an entity (an object) capable of performing computation, and for low overhead, it contains only the minimal state necessary. Another term for a thread is a *lightweight process*. A thread contains the processor state, the contents of a machine's registers. All threads within a task share the virtual memory address space and communications privileges associated with their task. The Unix abstraction of a process is simulated in Mach by combining a task and a single thread. However, Mach goes beyond this abstraction by allowing multiple threads to execute simultaneously within a task. On a multiprocessor, multiple threads can, in fact, execute in parallel on separate processors, whereas on a uniprocessor they only conceptually execute in parallel.

Ports, Port Sets, and Messages

A port is a communications channel, a sort of object reference for tasks, threads, and other objects. Application programs

continued

Mach on the NeXT Cube

While most current users of Mach are content to rely on Unix compatibility, NeXT has found the basic functionality of a Unix system to be insufficient to produce high-quality end-user applications software. NeXT utilizes the Mach functionality for communication between applications and window servers, sound playback and recording servers, and other applications.

Applications on the NeXT Computer convey information to a user according to the NextStep User Interface, which comprises several components: The Window Server manages all the windows on a display; the Application Kit is an implementation of the classes that define the user interface; the Interface Builder is a tool that allows the user interface for NextStep-conforming applications to be built with little or no programming; finally, the Workspace Manager provides a graphical user interface to a user's files and applications.

Two major types of communication occur between NextStep applications. First, applications communicate with the Window Server in order to implement a graphical user interface according to the client/server model. Second, applications communicate with each other using the Workspace Manager as a rendezvous point. Both types of communication are performed using Mach's intertask message-passing primitives.

Sound playback and recording make extensive use of Mach features. On a NeXT machine, compact-disk-quality sound can be synthesized in a digital signal processor. The device driver responsible for controlling the DSP and the sound direct-memory-access channels is accessed using Mach's message-passing primitives. This allows great flexibility in how the hardware can be accessed and provides network-transparent access to the driver. Threads are also used by high-level sound software to control sound I/O for an application that needs to perform normal processing while playing or recording sounds.

communicate with objects managed by the kernel and server tasks through the objects' ports. This is the software counterpart to the communications ports on the hardware. An object is said to have "access rights" to a port if it has dealings with that port. A port can move around from object to object, like moving a board and the cables connected to it from one machine to another.

The object that has the port screwed into it is said to have *receive access rights* to the port. Receive access rights imply send access rights as well. More than one thread may concurrently attempt to receive messages from a given port, but all the threads must be within the same task. In other words, only one task can have receive access rights to the port.

The object intending to pipe messages to the port has *send access rights*. More than one thread and more than one task can hold send access rights to any port. Messages travel from the object with send access rights to the port on the object with receive access rights.

For the time being, there is also a third port access right, *ownership*, which determines which object gains receive rights when these rights are relinquished. The Mach documentation implies that ownership rights will probably not be implemented in future releases—a definite discouragement for using this privilege.

Both tasks and threads have a special *kernel port* by which the kernel recognizes them.

Some special types of ports are associated only with tasks: the *notify port*, through which the task receives messages from the kernel about its port access rights and the status of messages it has sent; the *exception port*, through which the task receives messages from the kernel when an exception occurs (see "Exception Handling," below); and the *bootstrap port*, with which new tasks attach to any services that they need.

Threads also have some special types of ports: the *thread reply port*, for early messages from a young thread's parent and early remote procedure calls (RPCs); and the *thread exception port*, similar to the task exception port. Ports can be strung together into *port sets*, through which several objects can grab any messages from a single message queue.

A message is a string of data prefixed by a header. The header describes the message and its destination. The body of the message may be as large as the entire address space of a task. There are *simple messages*, which don't contain any references to other ports; and *non-simple messages*, which can make reference to other ports—conceptually similar to indirect addressing.

Messages are the primary way that tasks communicate with each other and the kernel. They can even be sent between tasks on different computers.

The Memory Object

Each Mach task can use up to 4 gigabytes of virtual memory for the execution of its threads. This space is used for the memory objects but also for messages and memory-mapped files.

When a task allocates regions of virtual memory, the regions must be aligned on page boundaries. The task can create memory objects for use by its threads; these can actually be mapped onto the space of another task. Spawning new tasks is more efficient because memory does not need to be copied to the child. The child needs only to touch the necessary portions of its parent's address space. When spawning a child task, it is possible to mark the pages to be copied or protected (the child is prohibited access).

Since messages are actually mapped into the virtual memory resources of tasks, intertask (interprocess) communication is far more efficient than old-time Unix implementations where the messages are copied from one task to the limited memory space of the kernel and then to the task receiving the message. In Mach, the message actually resides in the memory space shared by the communicating tasks.

Memory-mapped files facilitate program development by simplifying memory and file operations to a single set of operations for both. However, Mach still supports the standard Unix file read, write, and seek system calls.

Virtual Memory

The Mach virtual memory system provides the programmer with a clean interface, which allows virtual memory to be allocated and deallocated at arbitrary addresses and sizes, restricted only by the page size of the underlying hardware. Applications can, on a page-by-page basis, specify access modes such as read-only, read/write, or shared. Finally, also on a page-by-page basis, virtual memory can be shared between

continued

tasks in a controlled fashion that is based on inheritance.

The virtual memory system achieves portability by splitting its implementation into two parts. The first part, the architecture-independent part, is common to all implementations of Mach. The second part, the architecture-dependent part, is specific to each hardware architecture that Mach runs on. This split makes it possible for Mach to provide a consistent, high level of functionality on all hardware architectures with only a minimal porting effort.

Open Memory Management

Instead of limiting virtual memory semantics to those defined by the kernel, Mach provides an interface that allows user-level programs (external memory managers) to define the exact semantics of virtual memory that can be mapped into any task's virtual address space. Such programs are responsible for handling operations such as "page in" (when a page of memory is referenced) and "page out" (when a page of memory is moved out of the normal working set). In addition, external memory managers can instruct the Mach kernel to take special action on memory, such as restrict access to data in order to provide data consistency and security.

The Mach kernel guarantees that only authorized senders can send messages on a particular port.

External memory management allows Mach to be extended in powerful ways without changing the base Mach kernel. For example, network-consistent shared memory can be implemented by an external memory manager. The shared memory manager can use the external memory interface to control which pages of memory can be accessed by which machines at various times in order to guarantee control. Not only does this remove that complexity from the kernel, but it allows the shared memory manager to choose which algorithms it uses to enforce consistency and security.

The Mach Kernel and IPC

The kernel functions can be classified into the following five groups:

- Task and thread management
- Port management
- Message queuing and support
- Virtual memory management
- Paging management

The kernel is responsible for the creation and management of all tasks and threads, the structure of ports associated with the tasks and threads, the messages between objects (through the object ports), and the allocation of physical and virtual memory. It manages what and how port capabilities are used. The kernel guarantees that only authorized senders or receivers can send or receive messages on each particular port. Thus, the Mach kernel guarantees secure interprocess communication (IPC) within a host.

The Mach kernel automatically queues messages for tasks executing on its machine. However, transmission of messages between separate Mach hosts is performed transparently by an intermediate server task.

The intermachine-process-server task is the *network message server*, and it maintains the mapping of local "proxy" ports to global "network" ports. It forwards messages using network protocols of its choice. In addition, it is free to make decisions related to security, or lack of it, depending on the environment in which it is run.

Exception Handling vs. Signal Handling

In traditional Unix, *signals* are used for notifying programs of events external to the program. The handling of signals is done within the program, but the semantics vary from one kind of signal to another. Signals come from only a portion of the events that affect a program from the outside. Bus errors, segmentation and protection violations, arithmetic processor errors (e.g., underflow, overflow, and divide by zero), and events associated with debugging also need to be able to communicate with programs. External events that affect the execution of a program are called *exceptions*. The traditional ways of handling exceptions (through application program signal handling and kernel handling of hardware errors) separate the application program or service program from the hardware that caused the exception and assume that there is only one processor (no longer a valid assumption).

Mach has taken a generalized view of all exceptions. An exception requires suspension of the "victim" thread that caused the event and the notification of an exception handler. The handler performs some operations as a result of the exception, and then the victim is either revived or terminated. Because the handler is never within the victim thread, all the exception handling involves a form of RPCs. Mach ports and messages are the elements through which all this happens. The handler's port for communicating with the task is the thread (or task) exception port.

This design provides a single facility with a consistent method of handling all exceptions, a simple interface, full support for debuggers and error handlers, and no duplication of functionality within the kernel. In addition, and of great interest to developers and researchers, this design allows for user-defined exceptions.

System Layers

The Mach kernel provides only the basic primitives needed for building distributed and parallel applications. Although Unix is an operating system, it is also a complete environment suitable for use by developers and end users. Mach is just a kernel. The operating-system environment is built on top of it. But, since Mach makes few traditional operating-system decisions within the kernel itself, it is possible to build a completely different operating-system environment on top of it.

Currently, the Mach kernel is the basis for a BSD 4.3 Unix-compatible system. In this system, the Mach kernel implements the features particular to an operating-system kernel and the features provided by the Mach kernel interface. Unix compatibility is provided by the original BSD 4.3 implementation, modified for use with Mach. In effect, many of the internals of BSD 4.3 have been replaced with Mach equivalents. This technique yields a highly compatible system with performance often exceeding that of the original BSD 4.3 system.

continued

- **Remains In Place** while you use your computer.
- **Avoids Costly Repairs.** Protects delicate electronics from dust, spills, smoke, ashes, staples.
- **Soft, Flexible,** retains normal keyboard feel.
- **Washable, Durable High-Tech Polymer** lasts years.
- **Hundreds of Models.** SafeSkin is available for most PCs, laptops, workstations and clone keyboards.
- Office • Home • Factory • Classroom • Laboratory

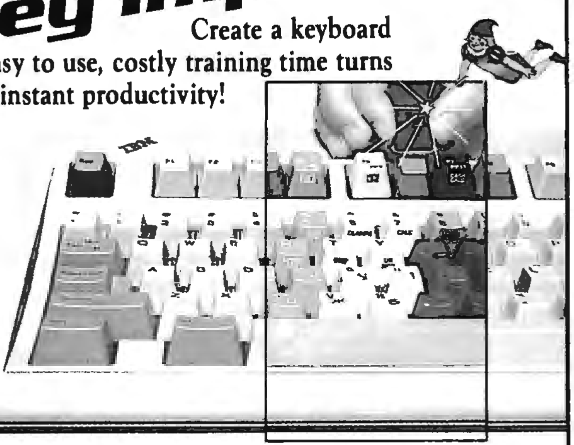
List Price $29.95. Please call or write for free color brochure. Dealer inquiries encouraged.

SafeSkin™ KEYBOARD PROTECTOR

Merritt Computer Products, Inc. 5565 Red Bird Center Drive Suite 150, Dallas, Texas 75234/(214) 339-0753 • FAX (214) 339-1313 In Canada call 1-800-663-1061

Custom Key Imprinting

Create a keyboard so easy to use, costly training time turns into instant productivity!



Custom Keys and Snap-On IBM KeyCaps: Available in a wide variety of colors and imprinted in your choice of colors and fonts. Keytop and Keyfront **Labels:** Supporting emulation, word processing or custom made to your specs. Won't wear out or come off keys until intentionally removed. FlexShield Keyboard Protectors: Extend keyboard life. Protect from dirt, liquid and damaging environments without restricting keyboard operation. Call for your **FREE CATALOG** of *Custom Keyboard Enhancements.*

Hooleon CORPORATION

For orders or custom info, call: 602 634-7515
FAX: 602 634-4620

P.O. Box 201, Dept. BYTE, Cornville, AZ 86325

SEE US AT FALL COMDEX IN THE RIVIERA – BOOTH R8534

OS/2 and Mach

Like Mach, OS/2 supports threads, virtual memory, and message-passing mechanisms. Although Mach and OS/2 provide similar types of functionality at the lowest levels, they differ in important ways.

OS/2 threads, for example, have some unusual semantics. Instead of the Mach model of all threads being equal, OS/2 treats some threads (e.g., the first thread in a process) specially. It manages virtual memory in segments, rather than pages—a finer grain and more flexible control than segments. Also, OS/2 imposes some other restrictions, such as the use of different memory allocators for different-size memory regions. Rather than provide one coherent mechanism for interprocess communication, OS/2 provides many different mechanisms (e.g., semaphores, pipes, queues, and signals). OS/2 lacks multiuser operations. Finally, OS/2 was designed to run on Intel 80286/80386 processors and is not portable to other processor families. This is not to say that OS/2 doesn't do well in its own niche, but it is not as complex or universal as Mach.

The Future of Mach

Unix compatibility makes Mach attractive to a wide audience by allowing it to transcend its role as a research project and emerge as a viable commercial operating system. The NeXT Computer already provides an excellent example of how to tie visual displays to audio input and output. The primitives of Mach were essential for NeXT to implement this functionality efficiently in a true multitasking environment. (See the text box "Mach on the NeXT Cube" on page 412.)

Mach is also influencing how other systems evolve. In the future, more and more systems are likely to support Mach features. Mach has become the platform for experimental Unix operating-system work. For instance, Trusted Information Systems, under a Defense Advanced Research Projects Agency contract, evaluated Mach as a possible base for a verifiably secure operating system, a "trusted system" meeting the B3 level of security as specified in the National Computer Security Center's "Orange Book." (See "Safe and Secure?" in the May BYTE.) Researchers at Trusted Information Systems ascertained that Mach's design made implementation of classification labels and access control lists much easier than in traditional Unix. The design separation of the kernel and services made modification of the operating system much more straightforward and easier to verify as being a trusted system. They have gone on to build a proof-of-concept prototype trusted system. But until Mach is free of BSD code, a truly trusted Mach operating system will not be possible. Meanwhile, they are working with the Mach team at Carnegie Mellon to ensure that facilities for trusted systems be properly implemented in future releases.

Work on Mach continues at Carnegie Mellon and organizatons such as NeXT. Eventually Mach will stand on its own and be completely free of BSD code. It will have been shaped by the tortuous tests of many other institutions, including industry and government. Thanks to the availability of Mach on the NeXT Computer, the ideas of thousands of researchers and students will add to its clever design and continue to shape it for modern computer design and software. It's a great example for all developers of applications and operating systems. But as operating systems go, Mach is very young, and few people understand all the possibilities it really provides. ■

Avadis Tevanian Jr. is the chief operating-system scientist at NeXT, Inc. He can be reached on BIX c/o "editors." Ben Smith is a BYTE technical editor and can be reached on BIX as "bensmith."