Meet Mach

By James Scott
December 17, 1997

OPENSTEP's (and now, Rhapsody's) beauty is more than skin deep. The flexibile, object-oriented user and developer environments are built upon a flexible, object-oriented operating system kernel called Mach. Mach was designed from the ground up as a multitasking, multithreaded operating system with very powerful communications facilities. It excels in multiprocessor and distributed environments. Moreover, it's relatively easy to port to a new hardware architecture. These features have made it the foundation of several cutting-edge operating systems.

Return to Main Page

In this article we'll review the history of Mach and outline the services it provides. This is a conceptual look, suitable for those who aren't familiar with the Mach architecture. Excellent technical documentation is available elsewhere; some pointers are provided below.

History

Mach traces its ancestry to the Rochester Intelligent Gateway (RIG), an operating system developed in the mid-1970s at the Rochester Institute of Technology. RIG was designed as a modular operating system for the Data General Eclipse minicomputer. Its main goal was demonstrating the utility and flexibility of an operating system based on a collection of processes communicating through a message-passing protocol. RIG achieved its goal, even demonstrating that its messaging facility could function across a network. Richard Rashid, one of RIG's designers, continued researching modular operating systems when he moved to Carnegie-Mellon University in 1979. The second generation modular OS ran on an early engineering workstation called the PERQ. This new operating system, called Accent, expanded RIG's capabilities. Accent added an innovative virtual memory system, transparent network messaging, and other features to RIG's modular base.

The Mach project started in 1984 and grew out of Rashid's desire to improve on Accent and provide a foundation upon which other operating systems could be built. Mach was designed around a *microkernel* - a small, efficient kernel providing basic services such as process control and memory management. Operating system emulators running as user processes (or *tasks* in the Mach parlance) used these services as the basis for higher-level functions such as file system and

network support. In this way, Mach could masquerade as another operating system, even running several OS emulators simultaneously. Other features included support for parallel multiprocessing, multi-threaded tasks, and refinements to Accent's excellent VM system. The designers targeted Berkeley Software Distribution (BSD) UNIX as the first emulated OS, due to the large amount of software available for it. Before the first release of Mach, the BSD emulation was moved into the Mach kernel. This violated the microkernel concept but did ensure complete compatibility with BSD UNIX - a concern of DARPA, one of the sponsors of the research. The first release of Mach appeared in 1986, running on the multiprocessor VAX 11/784.

Many companies have used various versions of the Mach kernel in commercial products. The first release of NeXTSTEP utilized the Mach 2.0 kernel, with some features added by NeXT. Later releases of NEXTSTEP and OPENSTEP added more features to the kernel, some adapted from Mach 2.5. The developer release of Apple's Rhapsody utilizes much of the OPENSTEP architecture and is based on Mach 2.5. The production version may be based on Mach 3.0, with BSD 4.4 and MacOS emulators. Mach 3.0 has all OS emulation code running in user space, leaving a microkernel of pure Mach. Current releases of OSF/1 and MkLinux from the Open Group are also based on the Mach 3.0 microkernel. Research on Mach at Carnegie-Mellon ended in 1994, but the Open Software Foundation's Cambridge and Grenoble Research Institutes continue to refine Mach's structure.

Architecture

Mach provides a small number of fundamental services - memory management, process management, communications, and I/O. These services do not in themselves provide all the features of a "real" operating system. Emulators running as user tasks build upon the microkernel to provide higher-level services, such as a file system and GUI. This approach provides several advantages. First, the kernel is smaller so it's easier to maintain. Second, moving the OS emulators into user space reduces their dependancy on the underlying hardware, making them much easier to port to a new architecture. Also, since they run as user tasks, developers can use standard tools (such as gdb) to debug them. Finally, more than one OS emulator can run simultaneously, allowing (for instance) Yellow Box and Blue Box applications to run side-by-side.

The Mach microkernel manages five fundamental abstractions - tasks, threads, ports, messages and memory objects. Tasks and threads provide the environment and means of executing a program. Ports and messages allow tasks and threads to interact with each other and with the

kernel. Memory objects provide a very powerful means of allocating and managing virtual memory. All of these concepts are discussed in more detail below. For clarity we will treat the Mach kernel and UNIX emulator as separate entities, even though this is not technically the case in Mach releases prior to 3.0.

Tasks and Threads

Tasks provide an environment for execution and are the basic unit of resource allocation in Mach. They consist mainly of a virtual, paged address space and at least four ports (ports will be described in more detail later). The *task port* (also known as a task's *kernel port*) is used to communicate with the kernel. The task port identifies the process that is requesting a service and the task which will be affected. The kernel uses a task's *notify port* to advise it of changes in port access rights and the status of messages it has sent. A task receives notification of exceptions (such as illegal memory access attempts or protection violations) through its *exception port*. The *bootstrap port* provides initialization information to a task, including how to access other system services.

A task also contains scheduling information that tells the kernel how to run its threads, a pointer to its operating system emulator, and statistics such as how much CPU time and memory the task has used. A Mach task does not contain a current working directory, file descriptor array, or user ID - all of which are associated with UNIX processes. That kind of information is handled by the OS emulator and is hidden from the kernel.

Threads are the executable entities in Mach. Tasks are passive; a task cannot perform any operations unless it contains at least one thread. Threads execute instructions and manipulate the contents of an address space. A thread is associated with exactly one task, but a task can contain multiple running threads. These threads share the resources provided by the task and are not protected from each other. They coexist in the task's virtual memory space.

In addition to its thread port (which is analogous to the task port above), a thread has a *reply port* and an *exception port*. The reply port is used in Mach remote procedurce calls (function calls over a network) to identify the calling task to the replying task. By default, a thread's exception port is set to NULL and all exceptions are sent to the thread's task. Ports may be associated with a thread but they are actually task-level resources. This means that a thread can have access to the ports of other threads running in the same task.

Threads are useful when several operations can execute concurrently on the same data. For instance, a file management program could use several threads at once. The

main thread could provide a display of the current directory and react to user commands. A user request to format a floppy disk would be executed in its own thread, allowing the main thread to remain responsive while the disk is formatting. On a multiprocessor host, each thread can be scheduled to execute on a separate processor, greatly improving performance.

Communications

In Mach, communication among operating system entities is performed by passing *messages* between *ports*. Ports form the basic object reference in Mach. Operations on objects (such as threads or tasks) are performed by sending messages to and receiving messages from the ports associated with those objects. Messages are of variable length, and may be of any size up to the size of the virtual address space. On 32-bit processors, that's 2^32=4GB! Message passing is not limited to objects on a single node; Mach provides facilities for passing messages across a network.

Ports can be thought of as mailboxes to which messages are sent and queued until another object reads them. They are implemented as message queues inside the kernel. Each port is protected; the kernel grants send or receive access to threads allowing them to send messages to or read messages from a port. Like UNIX pipes, ports support one-way communication. Only one object can have receive rights on a port at any given moment, but several objects can hold send rights for that port. Unlike pipes, ports support message streams (as opposed to byte streams). Three messages of 100 bytes each are guaranteed to arrive as three messages, not one message of 300 bytes. The kernel also guarantees that all messages will be delivered, and that messages sent by a particular object will arrive in the order in which they were sent. Ports can be grouped into read-only *port sets*, which allow an object to read from several ports at once. This allows, for instance, a server to listen for input from several ports without dedicating a thread to each one.

It should be noted that Mach objects and messages do not correspond to Objective-C objects and messages. The Objective-C runtime environment uses a different mechanism for passing messages among its objects.

Virtual Memory

Mach's modularity and flexibility extend to its virtual memory system. Mach divides memory management functions into three parts. The first part, known as the pmap *module*, runs in the kernel. It manages the hardware MMU and is therefore machine-specific. The second part of the virtual memory manager consists of machine-independent kernel code responsible for processing page faults and

managing task address maps. The third part, known as the *memory manager* or *external pager*, handles logical memory management duties, such as tracking memory pages stored in the swapfile on disk. There is a default memory manager that runs as part of the kernel, but users can supply their own memory managers (running outside the kernel) for special situations.

As mentioned above, each task receives a 4GB (on 32-bit machines) virtual memory space. Mach supports sparse memory allocation through the use of *memory regions*. A memory allocation call can specify a base address in addition to the size of the region to be allocated, allowing the programmer to control where in the virtual address space the memory is allocated. This new memory region does not have to be continuous with a previously allocated region. The memory manager keeps a list of allocated regions, eliminating the need for an inefficient linear table. Mach also refrains from mapping virtual pages to physical pages until the thread attempts to write data to a virtual page, keeping memory usage to a minimum.

Memory regions allocated by a thread are one example of Mach *memory objects*. A file on disk can also be mapped into a previously unused portion of the virtual address space, forming a memory object. Receiving a message also causes the creation of a memory object as the message is mapped into the receiving thread's address space. Think of a computer's physical memory as a cache for memory objects. The kernel manages the pages of the memory objects, keeping track of which ones are in physical memory and swapping them to disk as needed. The kernel also allows tasks mapped to those objects to use the physical memory pages that they occupy.

Mach allows the task to specify protection values for its virtual memory pages. Protection values can be any combination of read, write, and execute. These attributes can be used to control access to memory shared between tasks. Mach also provides very fine-grained control over memory inheritance. A task can assign an inheritance attribute for each allocated page in its address space. Pages marked *copy* are mapped into the child task's address space. However, they are not physically copied to another location until the child task attempts to write to that address. This is initially much faster and uses less memory than physically copying all of the parent task's allocated memory into the child task's address space. The *share* attribute allows true sharing between parent and child tasks - each can read and write the page, but the programmer is responsible for avoiding race conditions. If the page is marked *absent*, the child does not inherit that page from the parent. That address is left unallocated in the child's address space.

This elaborate virtual memory system provides features not found in other operating systems. Its design isolates machine-dependent code into a few modules, making it very portable. The memory allocation and protection functions give the programmer a great deal of control but make very efficient use of system resources, allocating physical memory only in response to write requests. Also unique is the way the memory system supports the messaging system. When an object receives a message, the kernel simply maps the message from the sender's memory space into the receiver's memory space. This is much less taxing that the typical UNIX implementation, which involves physically copying the message from the sender, into the kernel, and from there to the receiver.

Mach manages only the most fundamental system services, but it offers solid building blocks for more complex structures. The task and thread management functions capitalize on whatever resources the system has available - single processor or multiprocessor. The message passing facilities allow objects to send complex data structures to each other and provide for transparent communication over a network. Mach's virtual memory management is a model of flexibility and efficiency, maximizing programmer control while minimizing demands on the physical memory and swapfile.

There are several sources for more detailed information on Mach. The NEXTSTEP/OPENSTEP documentation contains technical documentation on Mach function calls. It also documents NeXT-specific features such as loadable kernel servers. General information on the Mach project and a wealth of published and unpublished material can be found at the CMU CS Project Mach Homepage. Several textbooks, including Tannenbaum's *Modern Operating Systems* (Prentice Hall 1992) and *Distributed Operating Systems* (Prentice Hall 1997) review Mach and compare it to other operating systems.

James Scott has been a NEXTSTEP developer since 1991. He and his trusty NeXTstation, Grendel, live in St. Louis, Missouri.