Mach micro kernel – A case study

Viswanath Veerappan
University of Texas
Arlington

This paper documents the architecture of the MACH kernel. It is assumed that the reader is familiar with basic ideas of an operating system and the functions of a kernel. Emphasis is placed on the features that make MACH unique from other kernels. This paper satisfies partially the course requirements of cse6306 (University ofTexas,Arlington).

Abstract
Shared memory multiprocessors are becoming increasingly available, and with them a faster way to program applications and system services via the use of shared memory. The major limitation in using shared memory is that it is not extensible network-wise, and therefore is not suited for building distributed applications. The MACH operating system kernel was designed in order to overcome this problem. MACH supports distributed and parallel computation with environments consisting of multiprocessors and network of uniprocessrs. The MACH operating system can be used as a system software kernel, which can support a variety of operating system environments.

1. Introduction

Operating systems have become increasingly large complex and expensive to maintain over the years. The software problems faced by manufacturers are magnified by a need for compatibility with the old and new memory architectures, CPU architectures and I/O organizations. MACH is an operating system kernel developed at Carnegie-Mellon University to support distributed and parallel computation [2]. MACH incorporates in one system a number of key facilities, which distinguish it from earlier virtual machine systems. It is a communication-oriented operating system kernel providing

a) multiple tasks, each with a large, paged virtual memory space,
b) multiple threads of execution within each task, with a flexible
c) scheduling facility,
d) flexible sharing of memory between tasks,
e) message-based inter-process communication,
f) transparent network extensibility, and
g) a flexible capability-based approach to security and protection.

Work on the MACH project has been implemented at several different institutions and as a result there are different versions of MACH available. This

paper will place emphasis on the architecture of the original MACH OS kernel developed at Carnegie Mellon University, with comparisons made to the other significant versions that are available.

2. Basic Architecture

As a working environment for developing application programs, MACH can be viewed as a small, extensible system kernel which provides scheduling, virtual memory and interprocess communications [14] and several, possibly parallel, operating system support environments which provide the following two items:

1) distributed file access and remote execution.
2) emulation for established operating system environments such as UNIX.

The MACH kernel is designed such that new operating system functions can be added to the existing kernel without the need to modify the underlying kernel base. A completely transparent network extensibility of all kernel functions is provided.

The basic MACH kernel functions are as follows:

- port and port set management facilities,
- task and thread creation and management facilities,
- virtual memory management functions [2],
- operations on memory objects

2.1 MACH threads and tasks

A thread [10] in a MACH is the basic unit of scheduling. Threads contain the minimal processing state associated with a computation, e.g. a program counter, a stack pointer, and a set of registers. A thread exists within exactly one task; however, one task may contain many threads. A thread may be in a suspended state (prevented from running), or in a runnable state (may be running or be scheduled to run). There is a non-negative suspend count associated with each thread. The suspend count is zero for runnable threads and positive for suspended threads. On a multiprocessor host, multiple threads from one task may be executing simultaneously (within the task's one address space).
Tasks are the basic unit of protection. Tasks contain the capabilities, namely the port rights, resource limits, and address space of a running entity. Tasks perform no computation; they are a framework for running threads. All threads within a task have access to all of that task's capabilities, and are thus not protected from each other. Tasks may be suspended or resumed as a whole. A thread may only execute when both it and its task are runnable. Resuming a task does not cause all component threads to begin executing, but only those threads that are not suspended. A task

includes a paged virtual address space (potentially sparse) and protected access to system resources (such as processors, port capabilities and virtual memory). Factors such as the decomposition of parallel programs affect their performance. Measurements of parallel program performance are improved if supported by information such as how programs are scheduled. This can be implemented by the MKM, the Mach (context-switch) kernel monitor [7].

2.2 Inter-process Communication

The conventional notion of a process is, in MACH, represented by a task with a single thread of control [1]. The MACH inter-process communication facility is defined in terms of ports and messages. Inter-process communication provides location independence, security and data type tagging. Message passing is the primary means of communication both among tasks, and between tasks and the operating system kernel itself [9]. System traps implement functions that are directly concerned with message communication. All the other functions are implemented by messages to a task's port.

MACH message primitives manipulate three distinct objects:

1. Ports - protected kernel objects to which messages may be sent and logically queued until reception. It is a simplex communication channel -- implemented as a message queue managed and protected by the kernel. A port is also the basic object reference mechanism in MACH. Ports are used to refer to objects; operations on objects are requested by sending messages to the ports that represent them.

2. Port sets - A port set is a group of ports, implemented as a queue combining the message queues of the constituent ports. A thread may use a port set to receive a message sent to any of several ports. They are protected kernel objects that combine multiple port queues and from which messages may be dequeued.

3. Messages - Messages are ordered collections of typed data consisting of a fixed size message header and a variable size message body. Messages may be of any size and may contain inline data, pointers to data, and capabilities for ports. A message may consist of some or all of the following:
   a. Pure data
   b. Copies of memory ranges
   c. Port rights

A task communicates with another task by building a data structure that contains a set of data elements, and then performing a message-send operation on a port for which it holds send rights [3]. At some later time, the task with receive rights to that port will perform a message-receive operation. The message transfer is an

asynchronous operation. The message is logically copied into the receiving task, possibly with copy-on-write optimizations. Multiple threads within the receiving task can be attempting to receive messages from a given port, but only one thread will receive any given message.

Inter-process communication in Mach is implemented using the message queue. Only one task can hold the receive right for a port denoting a message queue. This one task is allowed to receive (read) messages from the port queue. Multiple tasks can hold rights to the port that allow them to send (write) messages into the queue. MACH and other server interfaces are defined in a high-level remote procedure call language called MIG [17]; from that definition, interfaces for C are generated.

2.2.1 MACH objects

MACH has a C-based object-oriented programming package that is integrated with the inter-process communication facility [3]. This provides the means to dynamically specify classes and objects [6]. Multiple inheritance, automatic object locking, garbage collection of objects, class/superclass hierarchy are all enabled due to this package.

2.3 Memory management

In MACH, physical memory is used as a cache of the contents of secondary storage objects called memory objects [1]. Memory objects are commonly files managed by a file server, but in the MACH kernel, a memory object may be implemented by any object (i.e. port) that can handle requests to read and write data. For each memory object the kernel keeps track of those pages that are currently in the physical memory cache and allows tasks mapped to that memory to use those physical pages [4]. A task uses the ‘vm_allocate’ call to the kernel, which allocates a memory object that provides zero-filled memory on reference; this memory object is managed by a default memory manager [5]. The memory manager defines a protocol for giving out memory object ports. This could take the form of the memory manager registering a general service port somewhere that clients could find and exporting an object create or object lookup call that will return a memory object port. This is the port that is passed to the kernel in the vm_map call.

The virtual address space of a task is represented as a series of mappings from contiguous virtual address ranges to such memory objects. A memory object is a secondary storage object that is mapped into a task's virtual memory. When a virtual memory request occurs that cannot be resolved through the use of a previously cached physical page, the kernel must make a request of the memory object for the required data. As the physical page cache becomes full, the kernel must replace

pages from the cache, writing the contents of modified pages back to the corresponding memory objects. Currently, the Mach virtual memory system allows the user to create memory objects that are managed by user-defined processes. An external pager is a process that provides data in response to page faults (pagein) and backing storage for page cleaning (page-out) requests. MACH supports multiprocessor scheduling and is currently in use on both general-purpose multiprocessor and uniprocessor systems. Normally, when a memory object is no longer referenced by any virtual address space, the MACH kernel will deallocate its port rights to that memory object after sending all port rights for the control and name ports in a memory_object_terminate call. To enhance performance, a memory manager may allow a MACH kernel to maintain its memory cache for a memory object after all virtual address space references to it are gone, by asserting the caching parameter to the memory_object_set_attributes call. However, allowing caching does not prevent the kernel from terminating an object.

In the event that a memory manager destroys a memory object port that is currently mapped into one or more virtual address spaces, future page faults on addresses mapped to this object (for which data is not available in the cache) will result in a memory exception.

In Mach the size of a virtual page can be changed and set even on a per-machine basis. Transforming a single page size scheduler into a multiple page size scheduler is not immediate. The multiple page size scheduler uses an arbitrary page size (scheduler page size) and solves the problem by two means:

- for requests smaller than the scheduler page size, the request is rounded up to the scheduler page size, and

- for requests larger than the scheduler page size, the request is fulfilled by multiple scheduler pages, after appropriate synchronization [1].

2.3.1 Synchronization

Synchronization is accomplished via a queuing mechanism. In MACH, importance is given to avoid false contention and descheduling of kernels until absolutely necessary. It is also necessary satisfy requests as quickly as possible while maintaining fairness. When the scheduler receives a request from a kernel, it may take one of the following actions:

1. Satisfy the request immediately.
2. Deschedule some writers and enqueue the request.
3. Simply enqueue the request.

The first is the case when there are no writers on any of the data that the kernel requests. For a read request, the scheduler can simply add the kernel to the set of readers of each scheduler-page; if the request is a write request, then the scheduler deschedules all readers of any scheduler-page in the writer's request range before scheduling the writer.

In the second case, the scheduler finds that there are writers on some of the requested data, but none of them have yet been descheduled. The scheduler deschedules the writers, and the request is queued.

In the third case, the scheduler finds descheduled writers on some of the requested data, indicating that other requests are already waiting for those scheduler-pages. In this case, the scheduler does not deschedule the rest of the writers because the requesting kernel is not yet ready to use their pages; the request is simply enqueued. When a descheduled writer sends a confirmation (a memory object_lock_completed () message), the scheduler finds the request that was awaiting it. If the confirmation was the last one that the request was waiting for, then the scheduler satisfies the request (as in case 1 above) and checks to see if there are any more requests that might be satisfied as well.

When many kernels share many memory objects serviced by the same pager the availability of each object decreases, because the pager becomes the bottleneck where all requests pile up. Even when few kernels are involved, the location of the server is important because local and remote messages might have very different costs. A distributed solution that can allocate any number of servers on any number of machines is more usable. In this way the sharing of memory between tasks located on the same (multi)processor is decoupled from unrelated events on other machines.

2.4 Transparent Shared libraries

Mach provides a facility known as the transparent shared library [6], which is a code library that is loaded in the address space of a program without its knowledge. This code library can now intercept system calls made by that program. These libraries are loaded by a parent process and inherited by the child processes using the virtual memory management facilities. The main purpose of this shared library is to provide binary compatibility with non-MACH OS environments. Other uses of this facility are:

a. Support for multiple OD environments like Unix 4.3, Unix V.4, etc.
b. Debugging and monitoring
c. Network re-direction of OS traps.

2.4 Fault Tolerance

The schedulers provide a mechanism for surviving kernel crashes whereby memory availability is preserved despite a failure of the current owner of a page [3]. This avoids the alternative of making the whole object permanently unavailable. In the event of a crash, or any other reason due to which communication with the server is severed, the server reverts to the latest copy it has of the page, which is the one that was sent to the writer when it asked for write permission. Failure of a kernel only needs to be detected when the server needs a page back from it.

2.5 Processor Allocation

The CPU-Server is a user-mode server that performs processor allocation [12] for the Mach operating system he server performs processor allocation by assigning processors to processor sets provided by its clients. This allows the clients to use and manage the processors without giving clients complete control over them; only the server has the port capabilities required to reassign processors. Processor sets are entities exported by the Mach kernel; threads assigned to a processor set run exclusively on processors assigned to that set and vice versa (with the tm exception of some Unix system calls).

The server interface is designed around a class of objects called requests [12]. A request consists of the following components:

- A run duration,
- A sequence of <processor set, number of processor> pairs.

A request is satisfied by assigning each processor set its corresponding number of processors for the run duration specified. The server enforces internal limits on the number of processors and the maximum run time. Current limits are 15 minutes and 75% or less of the processors on the system.

3. Other versions of MACH

Considerable work has been done on MACH by different organizations. The Open Software Foundation, University of Utah, Tenon Systems, and the Free Software Foundation are some of the organizations that have developed different versions of MACH by implementing different features to the existing version. Some of the other versions of MACH are:

1) MACH-US [18]
2) MACH 4
3) xMACH
4) Real Time MACH (MACH-RT) [8]
5) GNUMACH

3.1 MACH-US

Mach-US system is an OS developed as part of the CMU MACH project. It is comprised of a set of servers, each of which supports orthogonal system services. Instead of one server supplying all of the system services as under the Mach BSD4.3 single server (UX), the Mach Multiserver (Mach-US) has several servers: a task server, a file server, a tty server, an authentication server, a network server, etc. It also has and emulation library that is mapped dynamically into each user process, and uses the system servers to support the application programmers interface (API) of the U. Object-Oriented Generic OS Interfaces. Several sets of C++ based object-oriented interfaces (virtual classes/methods for multiple inheritance) define the semantics supplied by the system servers.

3.2 Mach4

The goal of the Mach4 project was to investigate some new research ideas, fix the major problems of Mach 3, and provide the base needed by the Flux project at the University of Utah, resulting in a fast, flexible, functional kernel.

3.3 xMach

xMach work began in November of 1999 by J. Mallett at the University of Utah. The goal was to create a free 4.4BSD style Operating System, based on a microkernel design. The central goals of this project were security, portability, with a minimum amount of codebase. Currently xMach is maintained, developed, and documented by a much larger group of people when it started. The most recent development version of xMach is: 0.1-CURRENT

3.4 Real-Time MACH

The Real-Time Mach microkernel-based operating system focuses on the subject of distributed real-time OS mechanisms and services for developing and supporting real-time and multimedia applications in current and future OSs. The philosophy behind RT-Mach is firmly based on real-time scheduling theory and in particular on priority-driven preemptive scheduling. RT-Mach extends this philosophy by adding a fundamental OS notion of temporal protection that enables the timing behavior of applications to be isolated from one another. The principles

behind this resource management philosophy have many implications to OS subsystems including scheduling policies, synchronization primitives, inter-process communication, file systems, windowing systems, device drivers, network protocols and communication protocol processing. The capabilities of Real-Time Mach span many of these subsystems while current and future planned work will address the other subsystems.

3.5 GNUMACH

GNU Mach is the microkernel of the GNU system. A microkernel provides only a limited functionality, just enough abstraction on top of the hardware to run the rest of the operating system in user space. In the GNU system, Mach is the base of a functional multi-server operating system, the Hurd. The GNU Hurd servers and the GNU C library implement the POSIX compatible base of the GNU system on top of the microkernel architecture provided by Mach.

4. Discussions

Systems like MACH have brought very important contributions to the research in microkernel model. The benefits in flexibility, extensibility, and in the ease of code writing provided by the microkernel approach are embedded in all serious operating systems currently being developed. Alternative ways of solving most of the performance problems brought by the microkernel architecture are being found.

A MACH user can run binaries developed for the different OS's at the same machine at the same time. This is a very useful feature but the performance of these emulations is now the major concern. When a system call to an emulated OS occurs, MACH transparently redirects it to an emulation library, which calls a user-level OS server. Exceptions are also redirected to this server.

The performance concern is particularly important for distributed applications or for local applications, which access remote servers because, network operations are based on user-level processes. Much of the performance degradation is due to the cost of maintaining separate protection domains, traversing software layers, and using a semantically rich inter-process communication mechanism

Communications between components of the extended ``OS'' requires that formalized message-passing mechanisms be used. As a result, code must be written to use the formal mechanisms, rather than processes being able to informally use system memory. This may reduce performance. Analysis of the HP PA-RISC port of Linux atop MACH indicates that the microkernelled version of Linux runs

approximately 10% slower than HP/UX [10]. New kinds of deadlocks and other error conditions are possible between system components that would not be possible with a monolithic kernel.

Implementing communication services using user-level network servers gave a great level of flexibility to MACH users and system administrators. It becomes possible to choose among several different network services, protocols, and interfaces to use.

Since the MACH microkernel is OS neutral, different OS systems can be hosted on it. The most significant advantage is the portability feature and the decoupling feature from the hosted OS.

4.1 Current Status

At this time the project on the original MACH has come to a close at CMU. Parts of the Mach Operating system have been incorporated in a number of commercial operating systems including:

- Encore's Multimax
- NeXT OS
- MachTen for the Macintoshes
- Omron's Luna
- DEC's OSF/1 for the DEC Alpha
- IBM's OS/2 for the RS6000 based machines.

There is still some work being done at CMU on the Mach multi-server system (Mach_US) and real-time Mach. Development work on Mach is continuing at the Open Software foundation, University of Utah's Flexmach project, Helsinki University of Technology's LITES system and the Free Software Foundation's HURD system

5. References

[1] Robert V. Baron, David Black, William Bolosky, Jonathan Chew, Richard P. Draves, David B. Golub, Richard F. Rashid, Avadis Tevanian, Jr., and Michael Wayne Young. **Mach Kernel Interface Manual** [abstract] [postscript] [doc] August, 1990. Unpublished manuscript from the School of Computer Science, Carnegie Mellon University.

[2] Alessandro Forin, Joseph Barrera, Michael Young, Richard Rashid. **Design, Implementation and Performance Evaluation of a Distributed Shared Memory Server for Mach.** [abstract] [postscript] [doc] A shorter version of this paper appears in the Proceedings of the Winter USENIX Conference, January 1989.

[3] Richard Rashid, Daniel Julin, Douglas Orr, Richard Sanzi, Robert Baron, Alessandro Forin, David Golub, Michael Jones. **Mach: A System Software kernel.** [abstract], [postscript] Proceedings of the 34th Computer Society International Conference COMPCON 89, February 1989.

[4] Linda R. Walmer and Mary R. Thompson. **A Programmer's Guide to the Mach User Environment.** [abstract] [postscript] [doc] February 1988. School of Computer Science, Carnegie Mellon University.

[5] Linda R. Walmer and Mary R. Thompson. **A Programmer's Guide to the Mach System Calls.** [abstract] [postscript] [doc] February 1988. School of Computer Science, Carnegie Mellon University.

[6] Richard Rashid, Robert Baron, Alessandro Forin, David Golub, Michael Jones, Daniel Julin, Douglas Orr, Richard Sanzi. **Mach: A Foundation for Open Systems.** [abstract], [postscript] Proceedings of the Second Workshop on Workstation Operating Systems(WWOS2), September 1989.

[7] Ted Lehr, David L. Black. **Mach Kernel Monitor(with applications using the PIE environment.** [abstract] [postscript] [doc] February, 1990.

[8] Stefan Savage, Hideyuki Tokuda. **Real Time Mach: Exporting Time to the User.** [abstract], [postscript] This paper is scheduled to appear in the Proceedings of the Third Usenix Mach Symposium(Machnix), April 19-21, 1993

[9] Silberschatz, J.L Peterson, P.B. Galvin **Operating Systems Concepts.** Addison-Wesley Publishing Company, 3rd Edition 1991Chapter 16, pages 597-628

[10] Bryan Ford, Mike Hibler, Jay Lepreau **Notes on Thread Models in Mach 3.0**. UUCS-93-012, April 1993. (Abstract)

[11] Jay Lepreau, Mike Hibler, Bryan Ford, Jeff Law **In-Kernel Servers on Mach 3.0: Implementation and Performance** *Proc. of the Third Usenix Mach Symposium*, Santa Fe, NM, April 1993.

[12] David L. Black. **The Mach cpu_server: An Implementation of Processor Allocation** [abstract] [postscript] [doc] August 1989. Unpublished manuscript from the School of Computer Science, Carnegie Mellon University

[13] David. L. Black. **Scheduling Support for Concurrency and Parallelism in the Mach Operating System.** [Abstract] [Postscript] CMU Technical Report CMU-CS-90-125, April 1990.

[14] Joseph S. Barrera III. **A Fast Mach Network IPC Implementation** [abstract], [postscript] Proceedings of the Usenix Mach Symposium, November 1991.

[15] Keith Loepere. **Mach 3 Kernel Principles.** [Postscript] Open Software Foundation document

[16] Keith Loepere. **Mach 3 Kernel Interface.** [part 1 postscript], [part 2 postscript], [part 3 postscript] Open Software Foundation document

[17] Richard P. Draves, Michael B. Jones, Mary R. Thompson MIG - The Mach Interface Generator". School of Computer Science, Carnegie Mellon University, November 1989.

[18] J. Mark Stevenson, Daniel P. Julin Mach-US: UNIX On Generic OS Object Servers", *Proceedings of the 1995 USENIX Conference*, January 1995.