Mach: the core of Apple's OS X

CS-384, Operating System Design
Winter 2005-2006

Submitted to:	Dr. Meier
Submitted by:	Erich Musick
Date Submitted:	February 23, 2006

Table of Contents

Introduction

Ever since Apple released its UNIX-based Mac OS X operating system in 2001, it has gained popularity as a viable alternative to Windows. Though many have yet to "make the switch" and begin using the operating system, Apple's recent addition of support for the Intel microprocessor architecture is yet another step that pushes people in that direction. Much of the stability and performance for which it is well known can be attributed to the careful selection of key aspects of the operating system's structure. Specifically, a low-level analysis of these core OS X components suggests that the selection of Mach for core kernel behavior plays a fundamental role in the efficiency of the entire operating system.

Overview of XNU

Apple's XNU ("**X** is **n**ot **U**nix") kernel is composed of three core components which together provide the foundational functionality for the Mac OS X operating system [Apple 2006]. The most well known component of the OS X kernel is based off of the Berkley Standard Distribution (BSD) of UNIX. The BSD functionality included in Apple's kernel is based primarily on FreeBSD 5.x and provides features such as permissions, an API to POSIX functions, BSD style system calls, and file system management. Though it is not as well known as a component of Mac OS X, a derivative of Carnegie Mellon University's Mach 3.0 microkernel provides much of the core kernel functionality that is necessary for the BSD-derived elements to work. It performs key traditional operating system responsibilities such as preemptive multitasking, memory management, interprocess communication, kernel debugging, and console input and output. The third component of the XNU kernel is the I/O kit, which provides object oriented device driver control through a language based on embedded C++ [Singh 2006].

Kernel Types

In the world of operating systems, there are two categories of kernels: monolithic kernels and microkernels. As their names suggest, their primary difference lies in their size and, consequently, the scope of operating system functionality contained within them [Galli 2000].

Monolithic Kernels

Described by one textbook as "The Big Mess," the structure of monolithic kernels can be summed up in the statement, "there is no structure" [Tanenbaum 2001]. A monolithic kernel contains most, if not all of the operating system and handles process, memory, file, name, and device management [Galli 2000]. It lacks protection boundaries, and contains no encapsulation of functionality [Tanenbaum 2001] – any function can reference any other function or any data. Because of their large code bases and lack of organization, monolithic kernels are generally difficult to debug, validate and modify. Their lack of modularization makes them poor candidates for distributed computing, because only a selection of all required tasks is done by each system yet each system contains the entire code base [Galli 2000].

Microkernels

Microkernels take a more modular, client/server approach to operating system design. The majority of the functionality is removed from the kernel, thus leaving only core processing to be done by the small kernel that remains. The removed functionality is instead implemented as user-level processes. The kernel mediates communication between these client processes and servers, such as those which facilitate file I/O or provide access to memory. The encapsulation provided by the kernel's restriction of access to servers helps to minimize the impact of bugs [Tanenbaum 2001]. Where an error caused by directly modifying a kernel-controlled variable (such as a process control block) might bring down an entire monolithic kernel-based system, the intermediary microkernel limits access to such sensitive information and forces all changes to go through the proper channels.

The modularity inherent in a microkernel-based operating system architecture results in more manageable pieces and lends itself well to distributed systems, in which a client running on one system requests data from a server on another. From the perspective of the client, "the same thing happens" whether the server is local or remote – "a request [is] sent and a reply [comes] back" [Tanenbaum 2001].

In many microkernels, the functionality specific to a particular computer architecture is kept together in a single module. Thus, the kernel can be ported as necessary to new architectures without being entirely rewritten. Rather, existing modules can be simply switched in or new ones can be written to meet the needs of the new architecture [Galli 2000].

The Mach Microkernel

Overview and History

Mach was initially developed by the Carnegie Mellon University's Computer Science department in the mid 1980s as a monolithic kernel [Singh 2006]. Intended originally to be used as the kernel for BSD 4.2, version 2.5 of Mach implemented most of the BSD functionality as a single kernel thread. Since it was not multithreaded, there were limits to its efficiency. As the code base grew substantially larger, the BSD functionality was extracted from the kernel and placed in external libraries, in true microkernel fashion. What remained in the kernel was used as the basis for Mach 3.0. Because the operating system-specific BSD code was moved out of the kernel and into user-level processes, the Mach microkernel simply acts as a foundational layer off of which different operating systems can be built [Silberschatz and Galvin 1993].

One of the primary goals of the Mach microkernel is "to be a distributed system capable of functioning on heterogeneous hardware." As such, it has provisions for supporting systems with anywhere between 1 and 1000 processors and was written to allow easy modification to run on a wide variety of computer architectures. Because of its roots in BSD, it also remains fully compatible in functionality with BSD 4.3 and is therefore a good choice for Apple's OS X kernel, in which BSD also plays a fundamental role [Silberschatz and Galvin 1993].

Philosophy

In addition to providing support for a variety of single and multiprocessor computer architectures built by various vendors, Mach developers aimed to limit the number of core abstractions. Using these few elements for nearly all operating system functionality avoids unnecessary repetition in code and maintains a reasonable size. It also supports the memory management and interprocess communication capabilities that are built in to the kernel and enable all other key operating system functionality to work on a broad range of architectures and over the computer networks they utilize. Additionally, much of the philosophy behind the development of UNIX, such as its easy-to-use API, its large library of utilities and applications, and its utilization of pipes has influenced the approach to Mach's architecture [Silberschatz and Galvin 1993].

Though it has its roots in and has been heavily influenced by BSD UNIX, the developers of Mach attempted to avoid some of the negative aspects of and mistakes made in UNIX. For example, there is significant redundancy present in UNIX and its support for multiprocessor systems is less than optimal. Additionally, Mach attempts to take the large number of abstractions present in UNIX and generalize them so as to create the minimal set while still providing all the necessary functionality of a kernel [Silberschatz and Galvin 1993].

Abstractions

The philosophy with which the Mach kernel was developed has resulted in a kernel that is "as efficient as other major versions of UNIX when performing similar tasks." This performance is achieved through the provision of the following six basic abstractions: [Silberschatz and Galvin 1993]

- **Tasks –** Commonly referred to in operating system theory as a process, a Mach task is the "basic organizational entity" [Kutrtzman and Dattatri 1995] and has its own memory space. A task, itself cannot be executed, but instead contains one or more threads with which it shares its resources [Silberschatz and Galvin 1993].
- **Thread –** The Mach concept of a thread differs little from its counterpart in other modern multi-threaded operating systems and can be viewed as the "executable entities in the system." Threads have their own registers and scheduling policies and can only exist in the context of a task [Kutrtzman and Dattatri 1995].

- **Port –** Mach utilizes ports as kernel-controlled bidirectional communication channels. In order to either send or receive data through these channels, a thread must have access rights, referred to as *port rights* [Silberschatz and Galvin 1993].
- **Port Set –** As its name implies, a port set is a collection of ports which utilize the same message queue. Each task is associated with a single a single port set. This helps to increase efficiency by allowing threads to send or receive messages through several ports simultaneously [Silberschatz and Galvin 1993].
- **Message –** Messages in Mach are the smallest unit of communication and contain the data being transferred between threads [Silberschatz and Galvin 1993].
- **Memory Object –** Memory objects consist of "byte-addressable data … managed by … an external memory manager" [Kutrtzman and Dattatri 1995]. Tasks create references to Mach's memory objects in order to access data from memory [Silberschatz and Galvin 1993].

Scheduling

In keeping with its philosophy, the Mach microkernel makes provisions for symmetric multiprocessing in its approach to scheduling. Thread execution is prioritized and is supported through a collection of run (ready) queues, composed of several local, per-processor queues and a global, system-wide queue. Underlying these run queues is a data structure which contains 32 doubly-linked prioritized queues [Black 1990]. It is interesting to note that as changes are made to the global ready queue, the kernel locks it to prevent conflicts caused by simultaneous modifications initiated by multiple processors [Silberschatz and Galvin 1993].

In earlier versions of Mach, ready queues managed threads with priorities from 0-127. Modifications have since been made to more closely correlate thread priority and the number of ready queues. Thus, priorities now range from 0-31 [Black 1990] and their assignment to threads is based on two factors: a “base” value and an offset that reflects historical CPU usage [Galli 2000]. In order to prevent starvation, Mach ages threads every two seconds [Black 1990].

In order to help eliminate scheduling inefficiencies, Mach allows threads to give two types of hints to the kernel. Each is designed with the specific goal of giving lower priority processes a greater opportunity to get scheduled onto the CPU. This is helpful when, for example, it is known that a lower priority process must execute some function before the higher priority process can continue to execute. Discouragement hints come in three strengths – mild, strong, and absolute – and reduce the likelihood that the process giving the hint will be switched onto the processor. Absolute discouragement hints prevent the thread issuing the hint from being put on the CPU for a specified amount of time. Strong hints decrement a thread’s own priority and mild hints simply “suggest” that the CPU avoid scheduling the thread. Handoff hints, on the other hand, explicitly tell the kernel to execute another thread in place of the one issuing the hint [Galli 2000].

Each processor has its own scheduler. When determining which thread to put on the processor next, the scheduler queries its own, local run queue first. If no processes are ready to be executed, it will select and execute the highest priority process from the global, system-wide run queue. Threads in local queues are bound to the processor in whose ready queue they reside. For example, the device driver for hardware connected to processor 1 will only ever be scheduled through processor 1’s run queue, as it is the only processor that can communicate directly with that hardware device [Silberschatz and Galvin 1993].

After the scheduler has selected the highest priority process from either the local or global run queue, it assigns it a time slice. Every 10 to 100 milliseconds, an interrupt occurs and the scheduler decrements the allotted time slice for the executing process. When the thread has no additional time remaining, the kernel context switches in either the next process ready to be executed or leaves the current process on the CPU, depending on which has a higher priority [Black 1990].

Interprocess Communication

In UNIX, communication between processes and systems is dependent on location. The sender must know the name of the recipient in order to address the message appropriately. Mach, on the other hand, takes a location-independent, “object-oriented” approach to interprocess communication [Silberschatz and Galvin 1993].

All communication – whether between two threads, a thread and memory, two distinct systems, or a thread and the kernel – can be seen as the transfer of information between two objects facilitated by a couple of the key Mach abstractions. Each object receives instructions in the form of messages on ports [Silberschatz and Galvin 1993]. In order to send or receive messages on a port, a thread’s task must have the proper rights. These port rights include send, send-once, receive, port set, and dead name. Just as a process that has ended enters the “terminated” state, so a destroyed port’s rights become “dead name.” The send-once right, as its name implies, allows the recipient of a message to send data back to the sender over the port a single time. A dead name right can be compared to a terminated process in traditional operating system theory [Kutrtzman and Dattatri 1995].

Messages sent by way of these kernel-controlled message queues consist of two main parts – a header and a data portion. The header designates the destination port – the port to which any necessary responses should be sent – and the length of the data. The data segment of a Mach message contains multiple data objects. In Mach 2.5, the combined size of these data objects was limited to 8 kilobytes, but the limit was removed in version 3.0. Data objects come in a variety of shapes and sizes, including port rights, raw data in the form of numbers, and pointers to data in memory. These pointers to data in memory "provide the means to transfer the entire address space of a task in one single message." When large quantities of data are being sent using these pointers, the kernel kicks in as necessary and aids in the transfer of the data. If the receiver and the sender of the data are on the same system, the kernel follows a copy-on-write technique, duplicating data for the recipient only once it writes to the data [Silberschatz and Galvin 1993].

Since ports are implemented as a part of the kernel, all utilization of this functionality must be mediated by the kernel, which enforces the port rights. Since the port functionality is securely separated from any user-level processes, the risk of tasks attempting to interfere inappropriately with communication is minimized. The kernel's oversight of the entire communication process prevents messages from being "counterfeited, misdirected or fabricated" [Kutrtzman and Dattatri 1995].

Mach's all-in-one approach to communication, utilizing one technique for all transmittal of messages, enhances support for multiprocessing and network communication. It further demonstrates the key Mach philosophy of providing the smallest, fastest kernel possible. Interprocess communication can often be slow, but implementing it at the core of the operating system, as Mach does, helps with the speed. Additionally, because only one method of communication is present, significant effort can be expended to optimize it, instead of developing several, less efficient methods [Silberschatz and Galvin 1993].

Memory Management

Memory objects, another key abstraction of the Mach microkernel, behave in a manner similar to other objects present in the system. They, too, utilize Mach's powerful interprocess communication and transfer data using pipes and messages. This method of implementation also provides allows developers to easily try out a wide variety of memory-manipulation algorithms [Silberschatz and Galvin 1993].

Mach's virtual address space allows for the addressing of more than 4 gigabytes of data. This address space is allocated to tasks not all in one single, contiguous memory region but in pages. Paging is the industry standard method of memory management and generally ensures optimal utilization of memory. As one with a basic understanding of operating system theory would expect, all threads in a task share the same memory. The bounds of this sharing are specified when a parent thread forks a child. The parent may grant the child either a copy of its memory space or permission to read and write directly to and from the parent's space original. Mach, though, does not provide any assistance in resolving conflicting memory writes between multiple threads in a single task. Rather, each task must implement its own safeguards to prevent or properly mitigate the occurrence of such conflicts [Silberschatz and Galvin 1993].

While memory management does not depend on the kernel - user processes are provided for the purpose of managing memory objects – Mach does provide a significant amount of support for such functionality. Because the Mach's kernel has its own memory needs, and not all user-based memory managers perform all the functionality the kernel needs to access memory, Mach provides a default memory manager. It utilizes a Pageout technique that takes advantage of the benefits of a FIFO with second chance implementation. In addition to supporting kernel access to memory, one key purpose of the default memory manager is to address simultaneous requests for modification of the same memory location [Silberschatz and Galvin 1993].

Concerns Regarding Mach's role in XNU

In his article entitled, "OS X is Holding back the Mac," Apple enthusiast and writer Dan Knight claims that Mach is the operating system's "Achilles' Heel." From this judgment, he suggests that Apple move away from using Mach in the OS X kernel in favor of a Linux-based monolithic kernel. Referencing a Wikipedia article about microkernels, he claims that "microkernels are inefficient because of all the communication taking place between different parts of the operating system" [2005]. Knight's suggestion, though is based on the false assumption that OS X has actually implemented Mach as a microkernel without any modifications. Apple, in its documentation on porting UNIX/Linux applications to OS X reminds developers that though "XNU is based on the Mach microkernel design," it also includes BSD functionality and is therefore "not technically a microkernel implementation." Apple even goes so far as to address the specific qualms Knight points out, acknowledging that communication between a microkernel and layers above it can result in a performance decrease. This issue is supposedly resolved, though, by the combination of "the strengths of Mach with the strengths of BSD" in the XNU kernel [2006].

Nonetheless, Knight's proposition is interesting. With Apple's recent move to the essentially industry-standard Intel architecture, it is not unreasonable to suggest that Knight is justified in thinking that there are other aspects of OS X that could benefit from changes. The challenge is finding where and how to make such enhancements. While the firm foundation provided by Mach is not likely to be replaced on a whim, only time will tell whether these improvements would be best made as optimizations of the existing XNU kernel or would be best gained by a kernel redesign.

Conclusion

The choice to place a variant of the Mach microkernel at the core of Mac OS X has provided a fundamental foundation for the operating system to become what many consider to be the most powerful consumer-oriented operating system in existence. Mach's emphasis on efficiency, accomplished by the utilization of few core abstractions and optimization of a single method of communication, has contributed to the success of Mac OS X. Additionally, Mach's architecture-independence that has contributed to its popularity and widespread use over the past two decades has also enabled Apple to make the recent switch to the Intel architecture. While the decision to use Mach in OS X's XNU kernel can be challenged, its long history and years of development make its removal as the core an unlikely prospect.

Works Cited

[Apple 2006] *Porting UNIX/Linux Applications to Mac OS X*, Apple Computer, Inc., http://developer.apple.com/documentation/Porting/Conceptual/PortingUnix/index.html (2006).

[Black 1990] D. L. Black, "Scheduling Support for Concurrency and Parallelism in the Mach Operating System," *Computer,* Volume 23, Number 5 (May 1990), pages 35-43.

[Galli 2000] D. L. Galli, *Distributed Operating Systems – Concepts & Practice*, Prentice Hall, Inc. (2000).

[Knight 2005] D. Knight, "OS X is Holding back the Mac," *Low End Mac*, http://lowendmac.com/musings/05/1214.html (2005).

[Kutrtzman and Dattatri 1995] S. Kurtzman and K. Dattatri, "Design Goals of Object-Oriented Wrappers for the Mach Microkernel," *Proceedings of the 40th IEEE Computer Society International Conference* (1995), pages 367-371.

[Silberschatz and Galvin 1993] A. Silberschatz, P. B. Galvin, "The Mach System," *Operating System Principles*, Fourth Edition, Addison-Wesley, Inc. (1993), pages 897-921.

[Singh 2006] A. Singh, "What is Mac OS X? – XNU: The Kernel," *kernelthread.com*, http://www.kernelthread.com/mac/osx/arch_xnu.html (2006).

[Tanenbaum 2001] A. S. Tanenbaum, *Modern Operating Systems*, Second Edition, Prentice Hall, Inc. (2001).