Systems Programming and Operating Systems	Induprakas Kodukula
CS414, Summer 1996	Class#19, 8/6/96

1 Class #19

This section discusses the *Mach* operating system. The main contributions of the *Mach* operating system were the following:

Tried to demonstrate that OS could be structured using message passing.

Tried to integrate virtual memory and interprocess communication.

The basis behind Mach is the so-called *μ-kernel* principle, which states that perfection is achieved, not when there is nothing left to add, but when there is nothing to take away. For an operating system design, this means to have a really minimal kernel which provides all the necessary mechanisms for various functionality required, but leaves as much of the policy decisions to user processes as possible.

The fundamental abstractions behind *Mach* are the following:

Processes: Also called a *task* in Mach terminology. A process provides an environment for execution and consists of threads, an addr space and a number of ports.

Threads: A thread is a single executable entity and consists of a single stack, a program counter and a set of registers.

Memory Objects: are data structures that can be mapped into an address space

Ports: are used to send and receive messages.

Messages: are the primary means of communication and *Mach* had several innovations in how copying is avoided.

1.1 Processes

A process in *Mach* is an address space, a collection of threads and a sequence of port numbers. Typical ports include:

Process port: Used to communicate with the kernel

Bootstrap port: Used for initialization purposes when a process starts up: For example, this is how a process finds out the set of kernel ports that can provide key services to a process

Exception port: Used to report exceptional situations to a process

Registered port: Used to communicate with standard system services.

A process can be thought of as consisting of a large number of threads. A process is runnable if at least one of the threads it contains is runnable. If no threads of a process are runnable, the process is said to be blocked. A process can be created as an image of a *prototype* process and this process is similar to **fork**.

1.2 Threads

Threads (also mistakenly called light-weight processes) execute instructions in an address space. All threads in a single address space share all the resources in that address space. Threads may also have *thread-specific* private resources. Every thread has a *thread port* associated with it which can be used to access thread specific data. In addition, a thread can access the thread ports for other threads, and hence can control each other.

The following operations are defined on threads:

fork
exit
join
detach
yield (necessary if scheduling is non-preemptive)
self
signal
wait
broadcast
cond_wait

1.3 Memory Management

The goal of memory management in *Mach* is to be flexible and at the separate machine dependent code from machine independent code. This is done in 3 stages:

1. *pmap module*: This runs in the kernel mode and is concerned with managing the memory management unit (MMU). This unit is responsible for setting up MMU, hardware page tables and traps all page faults.
2. *machine independent kernel code*: Responsible for processing page faults, managing address maps and replacing pages as necessary.
3. *external pager*: Responsible for backing store: this keeps track of which virtual pages are currently in use, which virtual pages are currently in memory etc.

However, *Mach* supports the notion of *sparsely populated* regions of memory by a process. For example, in a 32-bit processor, the top half of the memory is used by the operating systems and the bottom half $(0 \ldots 2^{31} - 1))$ is used as virtual addresses for processes. *Mach* allows a process to allocate *regions* of memory at regions that are separated by large spaces. If sparse regions were not supported explicitly, a large number of page table entries would be required to cover the entire virtual memory space and a large amount of these entries would be pointing to regions of space that no process would be using.

Next, *Mach* provides the notion of a *Memory Object*. A memory object can be a page, a set of pages or an entire data structure that can be mapped into a region. For example, mapping a file in this manner allows memory-style random access to a file.

Finally, *Mach* provides *external pagers* for each object that is mapped into memory to handle a page fault. A three way communication mechanism ensues when a page fault happens:

> When a thread attempts a page reference to a non-existent page, the memory manager is informed via an **object port**.

The memory manager takes necessary action and informs the kernel at a **control port**.

Finally, every external pager has a **name port** for each region in memory, so that a thread needing to do an operation on a specific region can contact the memory manager at a specific port.

1.4 Communication

The final aspect of *Mach* is communication. The first such abstraction is a **port**. A **port** is a kernel data structure that provides a reliable seqeunced stream of messages. A port has protection implicitly associated with it, so only an authorized process is allowed to read from or write to a port. *Capability lists* are used for this purpose.

Mach also provides the notion of operations on **port sets**. Operations on a **port set** succeed if it succeeds on even one of the ports in the set. This is similar to the functionality of the **select** operation in UNIX.

Ports can have rights. These rights can be summarized as follows:

Receive: At any time, only one process can receive on a port.

Send: Usual meaning.

Send-Once: Destroyed after a port has been used once, this is used in a *request-reply* kind of protocol.

Finally, messages in *Mach* are of two types:

Simple Messages: Simply a sequence of bytes.

Complex Messages: Sequence of $< descriptor, data >$ fields, where the descriptor includes capability information that allows for safe retrieval of messages.

Mach attempts to make message passing more efficient by not doing needless copying of data if message passing can be safely achieved by changing the memory mapping of the processes involved in message passing.