A Comparison of Three Microkernels

Andrew S. Tanenbaum

Dept. of Mathematics and Computer Science
Vrije Universiteit
Amsterdam, The Netherlands

Abstract

The future of supercomputing lies in massively parallel computers. The nodes of these machines will need a different kind of operating system than current computers have. Many researchers in the field believe that microkernels provide the kind of functionality and performance required. In this paper we discuss three popular microkernels, Amoeba, Mach, and Chorus, to show what they can do. They are compared and contrasted in the areas of process management, memory management, and communication.

Keywords: Microkernel, Amoeba, Mach, Chorus, operating system

1. INTRODUCTION

Future advances in supercomputing will require the use of massively parallel computers, containing thousands of powerful CPUs. To perform well, these parallel supercomputers will require operating systems radically different from current ones. Most researchers in the operating systems field believe that these new operating systems will have to be much smaller than traditional ones to achieve the efficiency and flexibility needed.

In this paper we first summarize the state-of-the-art in operating systems for parallel supercomputers, and point out their shortcomings. Then we discuss a new trend—microkernels—and point out why there is interest in them. The rest of the paper deals with three modern microkernels: Amoeba, Mach, and Chorus. After a brief introduction to each one, we look at the three main areas that any operating system must handle: processes, memory management and communication, and describe how each of these example microkernels deals with that area.

2. OPERATING SYSTEMS FOR PARALLEL SUPERCOMPUTERS

Except for the simplest embedded applications, every CPU needs an operating system to manage its resources and hide the details of the hardware from the application programs. Typical functions of an operating system are to allow processes to create, destroy, and manage subprocesses, allocate and return memory units, making sure that each process can access only the memory it is entitled to access (by setting up the MMU properly), handle communication between processes, both locally and on other machines, and perform input/output.

Parallel supercomputers are no exception: they also need operating systems. First, the individual nodes have resources that have to be managed: the processor must be scheduled, memory must be allocated, etc. Second, the system as a whole has global resources that must be handled: processes have to be assigned to processors, I/O has to be performed and so on. Since parallel computing is in its infancy, no consensus has yet formed on the best way to tackle the problem, but it is becoming clear that simplicity, flexibility, and high performance are crucial here. At present, two approaches to parallel operating systems are widely used. Below we discuss both of them and show why they are converging on a third approach.

The first approach is the hosted system in which the parallel supercomputer consists of a large number of (identical) processing nodes plus a *host* computer. The host computer, which

can range from an ordinary workstation to a Cray Y-MP, runs a standard operating system, normally some version of UNIX®. The other nodes do not run any operating system at all. Instead, the application program is compiled with a special runtime library containing procedures for doing limited process and memory management, plus procedures for communicating with other nodes and with the host. The application program runs in kernel mode (if there is one) and takes over the whole machine. Most system calls are executed by sending messages to the host computer, which then executes them and sends back the results. Parallel supercomputers using this approach include the Cray T3D, Fujitsu AP1000, Meiko CS-1, Thinking Machines CM-2, and Intel iPSC/*x*.

The other model, the *symmetric* system, has a complete operating system running on each node. Again, some version of UNIX is the most popular choice. There is no need for a host since each node runs a complete operating system (although a host can be used for program development). Parallel supercomputers using this approach include the Meiko CS-2, Thinking Machines CM-5, and IBM SP1.

Each of these models has its problems. The hosted model is very primitive, and harks backs almost 50 years to the very first computers, which also used runtime libraries instead of operating systems. It also means that it is impossible for multiple users to run jobs on the same parallel supercomputer at the same time in a protected way. For high-performance applications that are highly I/O bound, such as imaging or scientific visualization, when a node is idle waiting for data, it is not possible to just run another process, because there are no other processes. Although theoretically an application using this model could use virtual memory by programming the MMU itself, in practice the lack of an operating system means that virtual memory is not available and programs must fit into real memory. Finally, having a single host handle nearly all the system calls means that the host is potentially a bottleneck and is certainly a single point of failure that will bring the whole system down if it crashes. As a consequence of all these problems, the trend is definitely away from having the computing engines run only a small runtime library.

Putting a full version of UNIX on every node solves most of these problems, but unfortunately it creates new ones. The basic problem is that UNIX is too large and inefficient for many parallel applications. For example, it was designed on the assumption that every node must control disks, terminals, and other I/O devices that are not present on all the nodes of a parallel supercomputer. If a system has 1024 processors, only a small number of which have a disk, it is wasteful to load each of the processors down with an operating system that contains a large, sophisticated file system.

Another basic problem is that with many operating systems, communication is a major issue, if not *the* major issue. In most existing operating systems, including UNIX, communication was grafted on late in life, does not fit in well, and is inefficient.

3. MICROKERNELS

The solution appears to be to have a new kind of operating system that is effectively a compromise between having no operating system at all and having a large monolithic operating system that does many things that are not needed. At the heart of this approach is a small piece of code, called a *microkernel*, that is present on all machines. It runs in protected (i.e., kernel) mode, and provides basic process management, memory management, communication, and I/O services to the rest of the system. All of the other traditional operating system services, such as the file system, are supplied by server processes, generally running in user mode and only on those machines that need them. This division of labor allows the microkernel to be small and fast, and does not burden each CPU with facilities (such as a complete file system) that it does not need.

Three strategies for achieving this goal are possible. In the first one, the runtime system is

gradually augmented with additional features until it does everything it has to do. In the second one, the full operating is stripped down until only a minimal subset is left. In the third one, a new microkernel is written from scratch, specifically designed to systems with many processors. In the short run, the first two (evolutionary) approaches are easier, but in the long run, code resulting from patch after patch tends to be complex and incomprehensible. A far better attack is to start over and write the necessary microkernels from scratch. For parallel computing, this approach is feasible because there is no huge backlog of parallel codes going back decades that must continue to run unchanged, and porting old sequential programs to parallel systems requires so much tuning and other work, that a little bit more will not make much difference.

Microkernel-based systems are starting to become available on parallel supercomputers. The Intel Paragon, Convex Exemplar, Unisys SPP, ICL Goldrush, Archipel Volvox, and BBN TC2000, all run microkernel-based operating systems, for example. In this paper, three microkernels will be described and compared to give the reader a feel for what a microkernel is and what the design issues are.

3.1. Amoeba

Our first example is Amoeba, designed and largely implemented at the Vrije Universiteit in Amsterdam starting in 1981. It was envisioned from the beginning as a minimal kernel to run on a system consisting of a large number of CPUs, called the processor pool. The complete system consists of the microkernel and various servers (file, directory, etc.), which together provide UNIX emulation as well as the native Amoeba application interface.

Like most operating systems, Amoeba has processes. Processes can have multiple threads of control within a single process, all of which share the process address space and resources. A thread is the active entity within a process. Each thread has a program counter and executes sequentially.

Unlike most comparable operating systems, Amoeba is based on the idea of objects, each of which is named and protected by a 128-bit *capability*. When a process creates an object (e.g., a file), the server managing the object returns a capability for that object. The capability contains bits telling which of the operations on the object the holder of the capability may perform. Since capabilities are managed by user processes themselves, and can be given away by their owners, the rights are protected from tampering by encryption. As a consequence, different users may have capabilities for the same object, but with different rights.

Amoeba is used by a number of universities and companies around the world as a base for research in parallel systems, languages, and applications, and is the operating system on an experimental multicomputer at the Vrije Universiteit consisting of 80 SPARC processors and 2.4 GB of memory. For more information about Amoeba, see the published papers [Mullender et al., 1990; Tanenbaum et al., 1990; and Tanenbaum et al., 1991].

3.2. Mach

Mach began its life at the University of Rochester as RIG, an operating system for the Data General Eclipse minicomputer [Ball et. al, 1976]. When one of its designers, Richard Rashid moved to Carnegie-Mellon University he designed a new operating system (Accent) based on RIG for the PERQ computer [Rashid, 1986b]. As experience was gained, this system evolved into Mach and was retargeted for multiprocessors.

Initially Mach was not a microkernel system, but was a monolithic system containing ideas from Accent, new ideas (such as multiprocessor support), and a large amount of code taken directly from Berkeley UNIX (because UNIX compatibility was a major design requirement). The last version in this line was Mach 2.5, which formed the basis of the Open Software Foundation's OSF/1 system. Subsequent work at CMU led to version 3.0, from which the Berkeley code had been removed from the kernel and put into user space, leading to a much

smaller kernel.

In addition to the work at CMU, OSF's research lab is developing its own version of the Mach microkernel (OSF/1 AD) [Zacjew et al., 1993]. The OSF version differs in some ways from the CMU version, including the network communication package, message formats, memory management, and support for real-time work. This paper will focus on the CMU version 3.0, but nearly all of what is said here applies to both the CMU and OSF versions.

Mach is based on five major concepts: processes, threads, ports, messages, and memory objects. A process, as in other systems, is a container for holding threads and other resources that are managed together. A *thread* is a lightweight process-within-a-process, as in Amoeba. A *port* is a mailbox that is used for communication. A *message* is a typed data structure that one thread can send to another thread's port so the receiving thread can read it. Finally, a memory object is a coherent region of memory, all of whose words have certain shared properties and which can be manipulated as a whole. In addition, there are a number of subsidiary concepts that will not be covered here.

The Mach microkernel runs on a variety of parallel supercomputers, including the Intel Paragon and Convex Exemplar. Additional information about Mach is given in the literature [Accetta et al., 1986; Baron et al., 1985; Black et al., 1992; Boykin et al., 1993; Draves, 1990; and Rashid, 1986a].

3.3. Chorus

Chorus began in 1980 as a research project at INRIA, near Paris, devoted to experimenting with distributed systems technology. The first version was written in interpreted UCSD Pascal for a collection of 8086s connected by a ring network. This version had a monolithic kernel, but as time went on, new versions were written in C and later C++, functionality was moved from the kernel to user space, and UNIX emulation was added.

Like Mach, Chorus has processes, threads, ports, messages, and memory objects. Although the names used by Chorus are different from those used by Mach, the functionality is roughly the same. Like Amoeba, Chorus also has capabilities for managing both system objects and user objects.

Chorus runs on the Cray T3D, Unisys SPP, ICL Goldrush, and Archipel Volvox among other parallel supercomputers. Additional information about Chorus has been published in various papers [Gien and Grob, 1992; Lea et al., 1993; Rozier, 1992].

4. PROCESS MANAGEMENT

In parallel supercomputers, processes play an important role, since there are many of them and they must be allocated to processors, often dynamically. In many applications, processes must also synchronize (e.g., no process may begin phase 2 until all processes have completed phase 1), so process synchronization is a key issue. Finally, for fine-grained computation, it is frequently useful to have multiple threads of control in a single process. In this section we describe how these issues are dealt with in Amoeba, Mach, and Chorus.

One note before starting the technical discussion concerns terminology. Each system invented its own names for many concepts. For the most part we avoid these and use the standard terminology widely used in the field.

4.1. Process Management in Amoeba

When a process is created in Amoeba, it is basically like a process in UNIX, namely an address space, a single thread of control and some resources. The process can then create additional threads of control that share the original address space. All threads of an Amoeba process run on the same processor, so threads are only quasiparallel (i.e., timeshared). For true parallelism, multiple processes are needed, one (or more) per processor.

The Amoeba kernel is itself structured as a small nucleus at the bottom and a collection of kernel threads that offer services to user processes. One of these services allows user processes to allocate a contiguous range of memory addresses called a *segment.* Segments are Amoeba objects, and as such are named, controlled, and protected by capabilities [Tanenbaum et al., 1986]. Using the segment capabilities, the process can write data into the segments. The user can then build a process descriptor containing the segment capabilities and other information, and give it to another kernel thread with a request to construct a process. The segments must all be on the same machine, but this need not be the machine on which the caller is running. The kernel then returns a capability for the new process, which is also an Amoeba object. The process capability contains rights to manage the process, for example to stop it, restart it, signal it or destroy it.

The parent process, such as a shell, can pass the capability for its new child process to another process, such as a debugger. Using the process capability, the debugger has fine-grained control over the process. For example, it can stun (freeze) the process, inspect or change its memory (i.e., its segments), and then restart it.

Although all threads in a process share the same program text and global data, each thread has its own stack, its own stack pointer, and its own copy of the machine registers. In addition, if a thread wants to create and use variables that are global to all its procedures but invisible to other threads, library procedures are provided for that purpose.

Three methods are provided for threads to synchronize: signals, mutexes, and semaphores. Signals are asynchronous interrupts sent from one thread to another thread in the same process. They are conceptually similar to UNIX signals, except that they are between threads rather than between processes. Signals can be raised, caught, or ignored. Asynchronous interrupts between processes use the stun mechanism.

The second form of interthread synchronization is the mutex. A *mutex* is like a binary semaphore. It can be in one of two states, locked or unlocked. Trying to lock an unlocked mutex causes the mutex to become locked. The calling thread continues. Trying to lock a mutex that is already locked causes the calling thread to block until another thread unlocks the mutex.

The third way threads can synchronize is by counting semaphores. These are slower than mutexes, but there are times when they are needed.

Amoeba threads are created and managed by the kernel so when a thread blocks (for example when waiting for an incoming message), the kernel can schedule another thread. In some systems (although in none of our three examples), threads are managed entirely in user space, without the kernel's knowledge. The disadvantage of this scheme is that when one thread blocks for any reason (I/O, page fault, etc.), the kernel has to block the entire process because it is not even aware that other (runnable) threads exist. The advantage of user threads is that context switching between them is faster than with kernel threads, since no trap to the kernel is required.

Amoeba threads can be scheduled either as run-to-completion or preemptively. In the former model, once a thread starts, it is allowed to run until it finishes or blocks. In the latter, threads are time-sliced, each one getting a fixed quantum. If a thread runs longer than its quantum, it is suspended and another thread is selected.

4.2. Process Management in Mach

Mach has processes and threads too. Like the Amoeba threads, Mach's threads are managed by the kernel, but a user-level threads package, called CThreads, can be run within a user process. CThreads allows a group of *m* user threads to be multiplexed onto a group of *n* kernel threads in arbitrary ways.

Synchronization is done using mutexes and condition variables. The mutex primitives are

lock, *trylock*, and *unlock*. Primitives are also provided to allocate and free mutexes. Mutexes work like binary semaphores, providing mutual exclusion, but not conveying information.

The operations on condition variables are *signal*, *wait*, and *broadcast*, which are used to allow threads to block on a condition and later be awakened when another thread has caused that condition to occur. *Signal* wakes up one waiting process; *broadcast* wakes them all up.

Mach provides extensive support for (shared-memory) multiprocessors. The CPUs in a multiprocessor can be assigned to *processor sets* by software. Each CPU belongs to exactly one processor set. Threads can also be assigned to processor sets by software. Thus each processor set has a collection of CPUs at its disposal and a collection of threads that need computing power. The job of the scheduling algorithm is to assign threads to CPUs in a fair and efficient way. For purposes of scheduling, each processor set is a closed world, with its own processing resources and its own customers, independent of all the other processor sets.

This mechanism gives processes a large amount of control over their threads. A process can give an important thread exclusive use of a processor set containing exactly one CPU, thus insuring that the thread runs all the time. It can also dynamically reassign threads to processor sets as the work proceeds, keeping the load balanced.

The Mach thread scheduling algorithm is based on priorities and scheduling queues. A scheduling queue is an array of lists of threads indexed by thread priority. Some threads (such as I/O drivers) are tied to one specific CPU, so each CPU has its own local queue of runnable threads, as well as access to the shared global queue. When a thread exits, blocks on I/O, or uses up its time slice, its CPU first checks its local queue to see if any local threads are runnable. If so, the first one on the highest priority queue is run for one quantum, at which time the algorithm is repeated. If the local queue is empty, the global one is checked. Processes may change their threads' priorities within limits, to allow control over the scheduling algorithm.

Threads may be preempted. On multiprocessors, the length of the quantum is variable, depending on the number of processors and runnable threads. The more runnable threads and the fewer CPUs there are, the shorter the quantum. This algorithm gives good response time to short requests, even on heavily loaded systems, but provides high efficiency (i.e., long quanta) on lightly loaded systems. A minimum quantum guarantees that context switching overhead will not come to dominate the performance.

On every clock tick, the CPU increments the priority counter of the currently running thread by a small amount. As the value goes up, the priority goes down and the thread will eventually move to a higher-numbered (i.e., lower-priority) queue. The priority counters are lowered by the passage of time.

4.3. Process Management in Chorus

Chorus has three kinds of processes. They differ in the amount of trust and privilege they have. Trusted process may call the kernel directly; untrusted ones may not. Privileged processes may do I/O directly; unprivileged ones may not.

Kernel processes are the most powerful of the three. They all run in kernel mode and share the same address space with the microkernel itself, and with each other. They can also be loaded into memory and removed from memory during execution, as needed.

Each *system process* runs in its own private address space (unlike each kernel process). They are trusted to make kernel calls, but they do not have the privilege to execute the hardware I/O instructions. System processes run in user mode.

User processes are neither privileged nor trusted. They cannot do I/O directly and can only get kernel services indirectly, by asking a system process.

System processes can be grouped together to form protected subsystems. Each subsystem has user processes that depend on it for providing service. A subsystem forms a kind of virtual kernel for its user processes. This mechanism has been used, for example, to provide a

subsystem that offers UNIX emulation. It has also been used to provide a base for object-oriented programming.

Associated with each process and each port is a *protection identifier*. Protection identifiers are inherited when a process forks. Protection identifiers do not have any semantics. They are just labels that identify a process and all its descendants. The UNIX subsystem uses them to implement UIDs (user identifiers).

Chorus processes each have at least one thread. Threads are known to, and scheduled by, the kernel. Each process has a priority and each thread has a relative priority. For scheduling purposes, the sum of the process' priority and thread priority is what counts. Threads in different processes compete for the CPU. On a multiprocessor with *k* CPUs, the *k* top-priority threads are run, although it is also possible to bind a particular thread to a particular CPU. Threads synchronize using mutexes and semaphores.

4.4. Comparison

All three systems support processes with multiple threads per process. In all three cases, the threads are managed and scheduled by the kernel, although user-level threads packages can be built on top of them. Amoeba gives processes the choice of run-to-completion vs. preemptive scheduling for its threads. Mach and Chorus allow processes to determine the priorities and scheduling policies of their threads in software.

Amoeba provides UNIX emulation via the library. Mach provides binary emulation of UNIX, MS-DOS, and other operating systems by catching system call traps and reflecting them to user-space emulators. Chorus supports system processes as well as kernel and user processes, and allows these to be set up to provide binary emulation of UNIX (and conceivably other operating systems). These can be loaded dynamically on demand.

Amoeba is based on the object model, and has capabilities for processes, segments, and other kernel and user objects, providing an integrated naming and protection scheme for all objects in the entire system. Chorus also has capabilities, but to a lesser extent, but they do not include rights bits for restricting which operations can be performed on which objects. Mach only has capabilities for ports.

Thread synchronization is done by mutexes and semaphores in Amoeba and Chorus. In Mach it is done by mutexes and condition variables.

All three kernels run on multiprocessors, but they differ in how they use the CPUs. Amoeba does not distribute the threads of a single process over the CPUs. They all run on the same processor. Instead, it is processes, not threads, that are spread over the CPUs. Mach, in contrast, allows fine-grained control of which threads are assigned to which CPUs using the processor set concept. This mechanism allows true parallelism among the threads of a single process. Chorus also supports having different threads running on different CPUs but with less user control over processor assignment than Mach.

5. MEMORY MANAGEMENT

All operating systems, including those for parallel supercomputers, must manage memory. Depending on the CPU chip used, virtual memory may also be available, in which case it, too, must be managed. Memory management on parallel computers is not appreciably different from memory management on single--processor computers, but it is still important. In this section we examine memory management in Amoeba, Mach, and Chorus.

5.1. Memory Management in Amoeba

Amoeba has an extremely simple memory model. A process can have any number of segments it wants to, and they can be located wherever it wants in the process' virtual address space. Segments are not swapped or paged, so a process must be entirely memory resident to run. Furthermore, although the hardware MMU is used, each segment is stored contiguously in memory.

There are three arguments for this unusual design: performance, simplicity, and economics. Having a process entirely in memory all the time makes communication go faster since no checks have to be made when an outgoing message spans virtual page boundaries. Similarly, on input, the buffer is always in memory, so large blocks of data can be placed there simply and without page faults. Not having paging or swapping also makes the system simpler and makes the kernel smaller and more manageable. Finally, as memory is getting so cheap, the primary reason for virtual memory, namely fitting large programs in small memories is decreasing. Lack of demand paging is not unusual in the supercomputer world, of course.

A segment may be mapped into the address space of two or more processes at the same time. This allows processes to operate on shared memory, for example, in a multiprocessor.

5.2. Memory Management in Mach

The conceptual model Mach provides to its user processes is a linear address space from 0 to some maximum address. Within this address space, processes can define *regions*, which are ranges of addresses, and can map memory objects onto regions. Memory objects are demand paged. A memory object can be one or more pages, but it can also be an entire file. When a file is mapped onto a region, the file can be read or written by just addressing it. No I/O calls are then needed. It is possible for two or more processes to share the same memory object at the same time. If a file is too large to be mapped in its entirety, a portion of it can be mapped.

When a process creates a child, three options are provided for handling the sharing of each memory object. The object can be absent in the child, shared between parent and child, or copied into the child, so parent and child each have their own private copies.

When the third option is chosen, Mach does not really copy the pages to the child but uses the *copy-on-write* mechanism instead. It places all the necessary pages in the child's virtual memory map, but marks them all read-only in both the child's and the parent's address spaces. As long as both make only read references to these pages, everything works fine. However, if either attempts to write on any page, a protection fault occurs. The operating system then makes a copy of the page, and maps the copy into the faulting process' address space as read-write, replacing the read-only page that was there. This scheme reduces copying overhead by eliminating the need to copy pages that are not actually written, even if they are potentially writable.

Another interesting concept present in Mach is that of the *external memory manager*. When a thread touches a page (i.e., a part of a memory object) that is not presently in memory, a page fault occurs and is caught by the kernel. However, unlike in traditional systems, the fault is not handled by the kernel. Instead it is passed to a user-level process for processing. This process is called an *external pager*, and different memory objects can have different external pagers.

Normally, the external pager will read in the missing page from a file or the disk's swap area and then pass it to kernel. The kernel can then install it, change the memory map, and restart the faulting process.

Of course if this were the entire story, memory would eventually fill up with pages. To prevent this from occurring, a kernel thread operates as a paging daemon, checking to see if memory is perhaps getting a bit full, and if so, sending a message to an external pager asking it to please remove some pages from memory. The advantage of having external pagers is that it removes a large amount of code from the kernel and puts it in user space. In particular, it

removes much of the policy to user space while leaving the mechanism in the kernel. The disadvantage is additional complexity, and the problems associated with balky or buggy external pagers that exhibit socially undesirable behavior (like not removing pages when asked to).

5.3. Memory Management in Chorus

Memory management in Chorus has many similarities to memory management in Mach, but it is not identical. The implementation, in particular, differs in a variety of ways.

Memory management in Chorus is based on regions and segments, a region being a range of contiguous addresses, and a segment being a collection of bytes named by a capability. Segments can be read using I/O primitives. Files and swap areas are segments. Segments can be mapped onto regions, but the fit need not be exact. Under normal conditions, mapped segments are demand paged, although real-time programs can disable this feature.

On machines with memory-mapped device registers, the kernel provides special I/O segments. These segments allow kernel threads to perform I/O by just reading and writing memory.

Mapped segments are backed by external pagers, as in Mach. In Mach, the initiative for traffic between the kernel and the external pagers generally comes from the kernel. In Chorus, a pager can take the initiative and ask the kernel to return dirty pages to it. When a mapper gets pages back from the kernel, it can keep them in its own memory or write them to disk, as it wishes. A complex protocol between the kernel and the mapper allows the mapper to specify which pages it wants back. If a mapper wants to keep track of how many kernel page frames it is occupying it can, but it does not have to since the kernel can discard clean pages at will.

5.4. Comparison

In Amoeba, a process can have any number of variable-length segments mapped into its virtual address space wherever it wants to. Once in, a segment can later be mapped out. Segments are controlled by capabilities, and can be read and written by any process holding the capabilities, including remote processes, for example, for debugging. Amoeba does not support demand paging, so all of a process' segments must be in memory when it is running.

In Mach, memory consists of memory objects. These, in turn, are built out of pages that can be moved in and out of memory as space requires. A memory object need not be fully in memory to be used. When an absent page is touched, its external pager is told to find it and bring it in. This mechanism supports full demand paging.

Multiple processes can share pages in a variety of ways. For example, a child process and its parent can share pages using the copy-on-write mechanism. However, this scheme does not work on a massively parallel system, where child and parent will normally be on different CPUs. This is one of several examples of where Mach's history as a kernel for a single machine shows through.

The memory model used by Chorus is largely the same as in Mach. It has segments controlled by capabilities that can be mapped in. These are demand paged under the control of an external pager, as in Mach.

All three systems support distributed shared memory but they do it in different ways [Bennett et al., 1990; and Li and Hudak 1989]. Amoeba supports an object-based system that allows variable-sized software objects to be shared using the kernel's reliable broadcast mechanism. Mach has a network message server that supports a page-based system. Chorus supports a page based system, but also has a special subsystem, COOL, for supporting software objects [Lea et al., 1993].

6. COMMUNICATION

Communication is probably the true test of a parallel operating system. In a single-processor operating system, interprocess communication is internal, and is thus managed entirely locally. In a parallel supercomputer, interprocess communication means sending messages between machines over some kind of network. These messages must be sent reliably and with a minimum of overhead. Most of the communication protocols developed for conventional networking, such as TCP, OSI, and X.25 are too heavyweight for use in a parallel system. In particular, the critical parameter here is not the bandwidth achievable for long transmissions, but the delay required for short messages.

In 1984, Birrell and Nelson published a paper describing how to make remote message passing look like a local procedure call [Birrell and Nelson, 1984]. This scheme, called *remote procedure call* (RPC), potentially provides a simple, high performance interprocess communication mechanism that retains most of the semantics of local procedure call. Very briefly, the calling process calls a library procedure, the *stub*, that has the same interface and parameters as a procedure on the remote machine. The library procedure packs the parameters into a message and sends them to a complementary procedure on the remote machine, which unpacks them and makes the actual call to the remote procedure. The reply goes the other way and unblocks the caller when it gets back. In this way, the neither the calling nor called procedure is aware that they are on different machines. This technique is widespread in distributed and parallel systems, including all three of our examples.

6.1. Communication in Amoeba

Amoeba supports three forms of communication: remote procedure call (RPC) using point-to-point message passing, group communication, and raw message passing. The latter is only used in special circumstances, so we will not consider it further.

The normal point-to-point communication paradigm in Amoeba is remote procedure call. This paradigm forces a certain amount of structure on the programmer in a way that just sending and receiving messages does not do.

In order for a client thread to do an RPC with a server thread, the client must know the server's address. Addressing is done by allowing any thread to choose a random 48-bit number, which we here will call a service address (*port* in the Amoeba literature, but this conflicts with the term in Mach and Chorus). Different threads in a process may use different service addresses if they so desire. All messages are addressed from a sender to a destination service address. A service address is nothing more than a kind of logical thread address. There is no mailbox and no storage associated it. It is similar to an IP address or an Ethernet address in that respect, except that it is not tied to any particular physical location. Cryptographic techniques are used to prevent one process from masquerading as another.

Amoeba RPC supports at-most-once semantics. In other words, the system guarantees that an RPC will never be carried out more than one time, even in the face of server crashes and rapid reboots.

Amoeba also supports reliable, totally-ordered group communication in the kernel. A *group* in Amoeba consists of one or more processes that are cooperating to carry out some task or provide some service. Processes can be members of several groups at the same time.

The implementation works best on LANs that support broadcasting or multicasting, but these features are not required. Reliable communication is also not required, as the protocol can recover from lost messages.

In short, the reliable boradcasting algorithm works as follows. One machine is elected to perform the *sequencer* function. To do a reliable broadcast, any process uses the kernel *send-to-group* primitive that sends a message to the sequencer. The sequencer then adds a sequence number and broadcasts the message. It also saves it in a buffer in case it needs the message

later.

When another kernel receives a broadcast message, it checks to see if the message bears the next sequence number in numerical order. If so, it passes it up to the waiting user process or buffers it temporarily. If the kernel sees that it has missed a broadcast, it asks the sequencer to give it the message it missed. The protocol is described in more detail in [Kaashoek et al., 1993].

6.2. Communication in Mach

All communication in Mach is based on reliable, one-way message passing. Remote procedure call, reliable byte streams, and other models can be built on top of this message passing. The central concept in the message passing system is the port, which keeps track of incoming messages. A thread can create as many ports as it needs. Messages go from a thread to a port, so for bidirectional communication, two ports are needed.

When a thread creates a port, a kind of capability is created for the port and put on a capability list stored in the kernel. Each process (not each thread) has a capability list for its ports. A small integer is returned to the caller giving the position of the capability in the list, analogous to a file descriptor in UNIX. This integer is used for subsequent calls using the port.

Capabilities contain rights telling what the holder can do with the corresponding port. The rights are RECEIVE, SEND, and SEND-ONCE. Each port has an owner, and only the owner has the RECEIVE right and can receive from the port. Initially the process creating the port is the owner, but the capability with the RECEIVE right can be passed to another process in a message. Only one process at a time may hold the capability with RECEIVE right for a port.

The SEND and SEND-ONCE rights allow the holder to send to the port, the difference between the two is that the latter is good for sending a single message, after which the capability is automatically deleted by the kernel. Capabilities with the SEND-ONCE right are used to build RPC communication, in which the client gives the server a capability for exactly one reply message.

Ports can be grouped in sets. A port set also has a capability, but only with RECEIVE rights. It is not possible to send to a port set. When a thread reads from a port set, it gets the first message from one of the ports in the set, but no guarantee is given about which port will be chosen.

The entity sent from a thread to a port is a *message*. A message contains a short header and an arbitrarily long sequence of fields, each of which has a well-defined type. The kernel guarantees reliable delivery of messages, so users do not have to be concerned with lost messages or acknowledgements. Messages from a given port are delivered in the order they are received. They are never combined, so even if a port contains many small messages, a thread reading from the port will only get one message. Specially marked messages can carry capabilities from one process to another. They are inserted by the sending kernel and removed by the receiving kernel.

To optimize message transport, when a process receives a message, the message is not actually copied. Instead it is mapped in using the copy-on-write mechanism described above. Only when a page is written to, does a fault occur and an actual copy made.

The description of ports and messages given above is more-or-less directly descended from RIG, and applies only to messages sent to a receiver on the same machine as the sender. To extend communication beyond one machine, Mach invented the concept of a *network message server*, a process in user space that can act as a proxy for remote processes. For example, a server on machine *B* can register with the network message server on machine *A*, which can then create a local port for the server on *A* and add it to the port set for all its remote customers. Clients on *A* can send requests to this port. When the network message server gets the message, it forwards it to the destination machine using whatever network protocol is appropriate.

This design puts the network code in user space, analogous to the external pager. However, several years of experience with this scheme demonstrated that the performance was unsatisfactory, so the design was changed, and in releases after 3.0 the networking code will be in the kernel. It is already in the OSF microkernel.

6.3. Communication in Chorus

In broad outline, communication in Chorus is similar to communication in Mach, but with numerous differences in the technical details. Like Mach, Chorus has ports and messages, and communication occurs when a thread sends a message to a port.

Ports are named and protected by Chorus capabilities, which are managed in user space. Ports can also be moved from one machine to another, for example, to allow a new server to take over the work load when the machine the old one is on must go down for maintenance. When a port is moved, optionally all its messages move with it or are deleted.

Ports can be grouped into port sets. It is not only possible to send a message to a port, but to a port set as well. The caller can specify whether the message is to go to one port in the set or to all the ports in the set. In the former case, there are various ways of indicating which port to use.

In addition, a thread can ask to receive a message from any of the ports its process owns, without specifying the specific port. Ports can be enabled or disabled. Ports can have priorities, in which case the highest priority enabled nonempty port is selected. Timers can be set on reads, so that if no message appears within a certain time interval, the call returns an error code.

Chorus supports two kinds of communication: one-way message passing and RPC. In the former, no guarantees are made about delivery and messages can be lost without the sender getting notification. Although message passing on a single machine will always be reliable, over a network it may not be. The advantage of this method is its simplicity and performance.

The second form of communication is RPC. Here there is always a request from the client and a reply from the sender. In the absence of crashes, RPC is always reliable. Even if the face of crashes, Chorus guarantees at-most-once semantics.

It is possible to associate a handler with a port, so when a message is received on the port, a thread is automatically created to run the handler and service the incoming message.

Messages consist of two parts, a 64-byte fixed part, intended for message headers, and a variable length body. Both are untyped. When a message is successfully read from a port, the fixed part, if present, is always copied into the caller's address space. A parameter in the call specifies whether the body is to be copied, mapped (with copy-on-write semantics), or temporarily left in the kernel. In the latter case, the calling thread can first inspect the header and then decide what to do with the body.

6.4. Comparison

Amoeba supports three forms of communication: unreliable one-way message passing, reliable RPC, and reliable, totally-ordered group communication. Mach supports one form: reliable one-way message passing. Chorus supports two forms: unreliable one-way message passing and reliable RPC, although the ability to send to a port set provides a primitive, but unreliable, broadcast facility too.

In Amoeba, messages are addressed to service addresses. The receiving thread must do a RECEIVE call that provides the buffer directly in user space. In Mach and Chorus, messages are sent to ports. These are data structures managed by the kernel that provide message storage. Both Mach and Chorus support port sets, although in Mach they are only for receiving, not sending. Alone among the three, Chorus allows a thread to listen to all of its (enabled) ports at once. Chorus' ports are named by capabilities managed in user space. Mach's ports are named by capabilities managed by the kernel and referred to by their indices in the kernel's capability

list.

Only Amoeba allows multiple replicated servers to listen to the same service address to stochastically distribute the requests over the servers. Only Mach has SEND-ONCE capabilities. Only Chorus supports automatic thread creation for handling incoming messages.

Messages in Amoeba and Chorus have a fixed part and a variable part and are untyped. Messages in Mach have only a variable part and are typed.

Mach and Chorus allow messages to be transmitted from one process to another using the copy-on-write mechanism. Amoeba does not have this. However, when sending messages between machines it is of no use; copying is always needed.

7. CONCLUSION

Parallel operating systems at present tend to be either simple runtime systems, complete UNIX systems, or microkernels. The latter combine the best features of the other two. We expect microkernels to become increasingly widely used as time goes on. In this paper we have surveyed three different microkernels and described how they deal with processes, memory, and communication.

It is surprising how similar the three microkernels are in many ways, especially considering the fact that they were independently designed and implemented by three different groups in three different countries. Perhaps the explanation is that all three groups were highly experienced in the field, and all three went through about four or five major iterations, changing their respective systems considerably based on experience gained from the earlier versions. This convergence suggests that perhaps a consensus is forming about how to design a microkernel-based operating system for future distributed and parallel computers.

ACKNOWLEDGEMENTS

I would like to thank Dick Grune, Philip Homburg, Marc Maathuis, Greg Sharp, and Greg Wilson for their helpful comments.

REFERENCES

Accetta, M., Baron, R., Golub, D., Rashid, R., Tevanian, A., And Young, M. 1986. Mach: A New Kernel Foundation for UNIX Development. *Proc. Summer 1986 USENIX Conf.*, pp. 93-112.

Ball, J.E., Feldman, J.A., Low, J.R., Rashid, R.F., and Rovner, P.D. 1976. RIG, Rochester's Intelligent Gateway System Overview. *IEEE Trans. on Software Engineering*, SE-2, 4 (Oct.), pp. 321-328.

Baron, R.; Rashid, R.; Siegel, E.; Tevanian, A. And Young, M. 1985. Mach-1: An Operating Environment for Large-Scale Multiprocessor Applications. *IEEE Software*, 2, 4 (July), pp. 65-67.

Bennett, J.K., Carter, J.K., and Zwaenepoel, W. 1990. Munin Distributed Shared Memory Based on Type-Specific Memory Coherence. In *Proc. Second ACM Symp. on Prin. and Practice of Parallel Programming* (Seattle, March), ACM, pp. 168-176.

Birrell, A.D., and Nelson, B.J. 1984. Implementing Remote Procedure Calls. *ACM Trans. Comput. Systems* 2, 1 (Feb.), pp. 39-59.

Black, D.L., Golub, D.B., Julin, D.P., Rashid, R.F., Draves, R.P., Dean, R.W., Forin, A.,

Barrera, J., Tokuda, H., Malan, G., and Bohman, D. 1992. Microkernel Operating System Architecture and Mach. In *Proc. USENIX Workshop on Microkernels and Other Kernel Architectures*, USENIX Association, pp. 11-30.

Boykin, J., Kirschen, D., Langerman, A., and LoVerso, S. 1993. *Programming under Mach*, Reading, MA, Addison-Wesley.

Draves, R.P. 1990. The Revised IPC Interface. In *Proc. First USENIX Conf. on Mach*, USENIX Association, pp. 101-121.

Gien, M., and Grob, L. 1992. Microkernel Based Operating Systems Moving UNIX onto Modern System Architectures. In *Proc. UniForum'92 Conference*, USENIX Association.

Kaashoek, M.F., Tanenbaum, A.S., and Verstoep, K. 1993. Group Communication in Amoeba and its Applications. *Distributed Systems Engineering J.* 1, (July), pp. 48-58.

Lea, R., Jacquemot, C., and Pillevesse, E. 1993. COOL System Support for Distributed Programming. *Commun. of the ACM*, 36, (Sept.), pp. 37-46.

Li, K., and Hudak, P. 1989. Memory Coherence in Shared Virtual Memory Systems. *ACM Trans. on Computer Systems*, 7, (Nov.), pp. 321-359.

Mullender, S.J., Rossum, G. van, Tanenbaum, A.S., Renesse, R. van, and Staveren, H. van 1990. Amoeba—A Distributed Operating System for the 1990s. *IEEE Computer Magazine*, 23, 5 (May), pp. 44-53.

Rashid, R.F. 1986a. Threads of a New System. *Unix Review*, 4, 8 (Aug.), pp. 37-49.

Rashid, R.F. 1986b. From RIG to Accent to Mach The Evolution of a Network Operating System. In *Fall Joint Computer Conference*, AFIPS, pp. 1128-1137.

Rozier, M. 1992. Chorus. In *Proc. USENIX Workshop on Microkernels and Other Kernel Architectures*, USENIX Association.

Tanenbaum, A.S., Kaashoek, M.F., Renesse, R. van, and Bal, H. 1991. The Amoeba Distributed Operating System - A Status Report. *Computer Communications*, 14, 4 (July-Aug.), pp. 324-335.

Tanenbaum, A.S., Mullender, S.J., and van Renesse, R. 1986. Using Sparse Capabilities in a Distributed Operating System. In *Proc. Sixth International Conf. on Distr. Computer Systems* (Cambridge, May 19-13), IEEE Computer Society Press, pp. 558-563.

Tanenbaum, A.S., Renesse, R. van, Staveren, H. van., Sharp, G.J., 1990. Mullender, S.J., Jansen, J., and Rossum, G. van Experiences with the Amoeba Distributed Operating System. *Commun. of the ACM* 33, 12 (Dec.), pp. 46-63.

Zajcew, et. al. 1993. An OSF/1 Unix for Massively Parallel Multicomputers. In *Proceedings of the Winter 1993 USENIX Technical Conference*, (San Diego, Jan. 25-29), pp. 449-468.