Microkernels as Foundations for Distributed Systems

John Bellardo, Michael Copenhafer, and Greg Hamerly
{bellardo,mcopenha,ghamerly}@cs.ucsd.edu

Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0114

Abstract. There are a number of key issues in constructing operating systems for distributed systems that are generally not present in operating systems for non-distributed or tightly-coupled systems. In particular, a distributed system must provide transparent access to shared resources (e.g. memory, communication channels, I/O devices). In this paper we argue that the design philosophy of microkernels makes them well-suited to the needs of distributed processing. We compare three different microkernels, Mach, Chorus, and QNX, in the areas of process management, interprocess communication, and memory management. We will show how each of these systems implements these mechanisms and how they facilitate distributed processing.

1 Introduction

In this paper we discuss the design philosophy of microkernels and argue that microkernels are well-suited to address problems of distributed processing. We compare three different microkernels, Mach, Chorus, and QNX, in the areas of process management, interprocess communication (IPC), and memory management. Our work is closely related to Tanenbaum's earlier research on microkernels for parallel systems [8], but is different in two respects. First, we believe that microkernels offer benefits not only in tightly-coupled systems, but also in distributed (loosely-coupled) systems. Second, our study compares two research systems (Mach and Chorus) with a commercial system (QNX).

Section 2 discusses distributed processing and the new problems it creates. Section 3 discusses our model of a typical microkernel, and sections 4-6 discuss Mach, Chorus, and QNX in the areas of process management, interprocess communication, and memory management. Section 7 compares these three microkernels with respect to the needs of distributed processing. Section 8 concludes.

2 Distributed Processing

Distributed processing attempts to distribute the computation of a task among several computers on a network. This technique can speedup computation and increase resource utilization. However, these benefits also come with new design problems. These include increased communication and synchronization costs, network transparent access to resources, scalability, remote process control, and fault-tolerance. We explain these problems briefly here, and discuss how microkernels help solve them in Section 7.

- **Communication and Synchronization.** Tasks that run in parallel need to communicate and synchronize with one another. This communication can be very costly, especially over a network.

- **Network Transparency.** It is possible to simplify the programming model for distributed systems by providing network transparent access to resources. This requires a uniform interface that is independent of the location of the resource.
- **Scalability.** It is desirable to have a proportional increase in system performance when there is an increase in the available resources (i.e. additional computers).
- **Process Control.** Distributed systems need to be able to create and control processes on remote machines. This provides concurrency and the ability to load balance computations.
- **Fault-Tolerance.** Since distributed systems have more resources than tightly-coupled systems, there is a greater risk that any one part of the system may fail. Therefore, it is important that distributed systems are able to continue making progress in the face of partial system failures.

3 Typical Microkernel

Microkernels are an operating systems design approach which emphasizes providing only the most essential functions as part of the kernel. While there is no single definition of what these essential functions are, most microkernels contain facilities for process management, interprocess communication, memory management, and device I/O.

- **Process Management.** Fine-grained control of processes is typically provided in the form of threads. Processes may have multiple threads of execution that share the same address space.
- **Interprocess Communication.** Messages provide a way for processes to communicate with one another and with the kernel. Message passing is often implemented with `Send()` and `Receive()` primitives.
- **Memory Management.** Abstract memory objects are typically provided to protect regions of memory. Virtual memory management may be implemented as part of the kernel or as a user-level process.
- **Device I/O.** Since device I/O is a privileged operation, a microkernel provides abstractions for communicating with devices.

Higher-level features such as device drivers, file systems, swapping, and networking are implemented as user-level processes. By including only core functionality, microkernels are very small, and are potentially more flexible and portable than traditional (monolithic) designs which implement a greater number of features in the kernel. The drawback of providing only core functionality is that user-level processes which are traditionally part of the kernel must now communicate with the kernel using the interprocess communication facilities. This can lead to a loss of performance.

4 Mach

The Mach project began at Carnegie Mellon University in 1985 where the first three versions were developed before the Open Software Foundation took over the project. Mach has always been fully BSD compatible. The first two versions (1.x, 2.x) contained the majority of the BSD code in the kernel, hence they would not be considered “true” microkernels. In version

3 the vast majority of the BSD code was moved into user-level processes[4] and the kernel contained only the bare necessities, making it a true microkernel. This will be the version of Mach that we look at in more depth. For more detail see [9].

4.1 Process Management

The Mach kernel provides two abstractions that together represent a UNIX process: the task and the thread.

A Mach task is used as a storage container for resources. All resources are allocated at the task level. Some examples of this are files, memory, ports, and threads. The task is also used as the unit of cleanup. If a violation occurs within a task all the resources in use by that task will be released, including all of its threads. A task must contain at least one thread.

The thread is the basic unit of execution. Each thread is owned by a single task and has access to all of the task's resources (including memory and files). The only data private to a thread is that thread's stack and a program counter. A task with only one thread is equivalent to a process in UNIX. If a task is suspended none of its threads will be considered for scheduling.

Mach makes its scheduling decisions at the thread level. Because all threads are equal (priorities aside) to the kernel, multiple threads from the same task can be scheduled to run simultaneously if there are enough CPUs. In addition, when one thread performs a blocking system call just that thread is blocked, not the thread's task.

Mach provides primitives that operate on tasks and threads. These operations are invoked by sending a message to a special kernel port (see the description of IPC below). They allow tasks and threads to be created, deleted, suspended, and resumed. They also allow a thread to voluntarily relinquish its processor. When a new task is created an existing task is used as a "template" for the new task. The template task specifies how its resources are to be shared when creating a new task. The different levels of sharing allow tasks to share memory objects (see the Memory Management section below).

4.2 Interprocess Communication

Mach provides a set of message passing primitives for IPC. Mach uses ports and capabilities to make message passing secure.

A port can be thought of as a mailbox. Messages are sent to and read from ports. Each task has a kernel-maintained port table in kernel protected memory. The port is an index into this table. Each entry in the table contains a capability to send that particular port. The different types of capabilities are send, send once, and receive. The receive capability is exclusive.

There are two ways a task can get a capability to a port. The first way, inheriting it from the template task when it is created, is described below in the section on memory management. The second way is in a message sent by another process which has a port capability.

Mach defines two different type of messages: simple and complex. Simple messages are passed as-is from the sender to the receiver; the kernel does not have any knowledge of the type of data in the message so they can't be used to send capabilities. Complex messages are composed of an arbitrary number $< type, size, data >$ tuples. During the process of delivering a complex message the kernel scans through all of the tuples looking for one whose type is *capability*. For each tuple it finds that matches this criteria the kernel creates a capability in the receiving task's port table and modifies the message to refer to the new capability. Since receive access to a port is exclusive the kernel may have to remove the receive capability from the sender if the capability is being sent in the message.

All kernel services (with the exception of message passing) are invoked by sending a message to a well known kernel port provided by the template task. A process communicates with a device by obtaining and then sending/receiving messages to a port representing the device being accessed.

Network transparency in Mach is achieved via a network message proxy. The job of the proxy is to forward all of the necessary messages across the network to their destination, and to receive all incoming messages and dispatch them to the correct process.

4.3 Memory Management

Mach is unique in its approach to memory management. It allows a small subsection of one task's virtual address space to be managed by another user-level task. The unit of memory allocation in Mach is called a memory object. A memory object occupies a number of contiguous bytes in the requesting task, and is managed by a task we refer to as an *object pager* (possibly the same task).

When a task requests a new memory object it specifies an object pager (the kernel provides a default object pager) and the virtual address where the new object should appear. The kernel creates a capability to represent the newly requested object and sends it to the object pager in a new object request message. The pager then examines the message and determines if it wants to satisfy the request. The kernel also creates two additional capabilities for the memory object that are used to communicate to/from the object pager.

During the life of the memory object the kernel makes service requests to the object pager. For example, the kernel can request that data be brought into a page frame to satisfy a page fault, and the kernel can request that a page be flushed out to backing store.

Mach provides the object pager with the ability to control the permissions on the objects it administers. This allows the pager to implement features such as copy-on-write sharing without special kernel support. The kernel also allows the pager to send the kernel messages that pertain to a memory object's state. Thus the pager can inform the kernel that a particular page frame is no longer needed by the memory object and can be used for other purposes.

There is no degree of trust between the kernel and object pagers, which creates a security problem for the kernel. It must protect itself from malicious object pagers. Mach does provide kernel mechanisms to deal with this problem, but the security issue still exists.

The interface that is used during the communication between the kernel and the pager is entirely message driven. The pager receives and responds to messages from the kernel, and vice versa. When used with the transparent network interface as described above, Mach can easily do remote paging. That means that the object pager can reside on a different computer.

5 Chorus

Chorus began as a research project in 1979 at INRIA in France. The goal was to build a transparent, distributed microkernel that provided better resource utilization, performance, and fault-tolerance than then-current monolithic kernels.

While at INRIA, Chorus underwent three major revisions. The first version, Chorus-V0, established some of the main concepts still used in the system, particularly the use of message passing within the kernel, and the notion of system processes called “actors.” The subsequent versions, Chorus-V1 and Chorus-V2, refined existing features and added support for binary compatibility with UNIX applications.

The current version, Chorus-V3, is an effort to move Chorus into an industrial setting. The system was rewritten in C++ and augmented with real-time support. Chorus-V3 will be the basis for further discussion. This section is only intended to cover certain key aspects. More complete discussions are available elsewhere in publications [1–3, 5].

5.1 Process Management

A process in Chorus defines a protected address space which encapsulates the following resources: a set of threads that share the resources of the process, a virtual memory context (discussed later), and a set of ports for communication with other processes [5]. There are three types of processes in Chorus, each having different execution privileges.

- *Supervisor processes* execute in the same address space as the microkernel and are permitted to directly execute kernel instructions. They may also execute privileged I/O instructions.
- *System processes* are permitted to execute kernel operations but may not execute privileged I/O instructions. Unlike supervisor processes, system processes execute in a private address space.
- *User processes* may execute neither kernel operations nor privileged I/O instructions. They run in a private address space.

Although Chorus can support multiple simultaneous processes, it is *not* possible to migrate a process and its threads to another site on a distributed system.

Many threads can execute concurrently within a process. Each thread is characterized by the state of the processor (program counter, stack pointer, registers, etc.). The scheduling scheme is very flexible: although the basic scheme is priority-based, Chorus also supports time-slicing and priority degradation on a per-thread basis.

Chorus supports UNIX-like `fork()` and `exec()` system calls for creating new processes. Threads are synchronized using mutexes, semaphores, or spin locks. These synchronization primitives may be used to construct condition variables and monitors.

5.2 Interprocess Communication

Processes communicate by passing messages via *port* objects. Chorus messages are contiguous byte strings which consist of a 63-byte header and a variable length body. Ports are abstract entities which represent the address of a process and a queue of unread messages. Their names are globally unique identifiers, making them location independent. While ports incur a performance penalty by providing an extra level of indirection between communicating threads, they provide a number of useful functions:

- **Flexible Communication.** Threads from different processes on potentially different sites of a distributed network may share messages using ports.
- **Multiple Communication Paths.** A single thread may accept multiple incoming messages by attaching more than one port to itself. Conversely, multiple threads may listen on a single port, allowing concurrent processing of data.
- **Port Groups.** It is also possible to group ports from various threads together into port groups. Messages may be sent to port groups providing a form of multicasting. Like individual ports, port groups are named by a unique identifier.
- **Dynamic Reconfiguration.** Ports may be migrated to different sites. This allows the implementation of service provided by a server to be reconfigured (which may involve removing the server from the network temporarily) without interrupting the clients of that service.
- **Protection.** Ports prevent unauthorized access to threads since the access to a port requires an appropriate capability.

Chorus offers two communication protocols. The first is asynchronous, one-way message passing. Chorus makes no guarantees about the reliability of a one-way message transfer. This protocol provides a highly efficient form of communication for services which do not require an explicit reply from a recipient. It may also serve as a basis for more reliable communications protocols.

The second form of communication is remote procedure calls (RPC). Unlike one-way messages, RPC is synchronous and reliable. More specifically, RPC guarantees that the the response received by a client is that of the server which received the original request.

For transferring large blocks of data, Chorus couples virtual memory and interprocess communication which permits copy-on-write techniques.

5.3 Memory Management

The unit of data abstraction in Chorus is called the *segment*. Segments generally represent some form of secondary storage such as a file. Similarly to other abstractions in Chorus, segments are global and are identified by capabilities.

Each process' address space is divided into *regions*. A region is a contiguous range of virtual addresses within a process which maps a portion of a segment to a given virtual address. Associated with each mapping is a set of access rights.

System processes known as *mappers* are responsible for mapping segments onto regions. If a process makes a request to read or modify data within a region, the mapper returns the appropriate segment containing the data. Segments are swapped on a demand basis by a user-level process called the *External Mapper*.

6 QNX

QNX is a commercial microkernel-based operating system provided by the QNX Software Systems corporation. It currently supports Intel x86-based systems, and is being ported to the Motorola PowerPC 7400. QNX is targeted for real-time systems and embedded systems as well as workstation-class platforms. The microkernel is quite small (12 Kb of code) and requires one service, the process manager. More information can be found in [6, 7].

6.1 Interprocess Communication

QNX handles IPC via message passing. Communication endpoints are specified by process IDs. Communication is network transparent in QNX: any process may communicate with any other process on a network if it has the correct permissions. The communicating processes are not aware of the location of the other process. When the kernel recognizes that a communication request is for a non-local destination, it invokes the kernel's network interface.

QNX has two optimizations to make message passing efficient: synchronous communication and multipart messages. The message passing primitives in QNX (send/receive/reply) only allow synchronous communication. Because of this, the kernel can copy memory directly from the sender's address space to the receiver's without buffering the message. This saves one copy operation per message, and reduces the size of the kernel's buffers.

QNX messages come in two flavors: single and multipart messages. Single messages are typical contiguous-buffer data transfers; the sender sends a block of data, and the receiver receives it.

Multipart messages are useful when message data is not in a contiguous block, but in disjoint locations. Without multipart messages, the sending process would have to create and copy the data into a contiguous buffer, and the receiving process would have to create a contiguous buffer, receive the message, and unpack the data by copying it out. Instead, the sender and receiver can each create an *MX control structure*. This structure specifies where the different parts of the message are located (in the sender's address space), and where they should be placed (in the receiver's address space). The MX structure is basically a block of pointers to different portions of memory. The kernel uses this information to copy the data directly from the sender to the receiver.

In addition to messages, QNX also supports IPC through *signals* and *proxies*. Both signals and proxies are forms of *asynchronous* communication, where the receiver does not need to interact with the sender. A proxy can simply be thought of as a non-blocking message, and a signal is a traditional UNIX-type signal where no data is transferred.

Synchronization is handled in QNX via message passing (since the primitives are synchronous) and shared memory semaphores.

6.2 Process Management

The QNX microkernel handles process scheduling, but process management is handled by the process manager, which is a separate process. However, the process manager runs in the kernel's address space; no other process or OS service does this.

The process manager supports process creation through the standard `fork()` and `exec()` primitives, and adds `spawn()`. Spawn merges the functionality of fork and exec (for efficiency), and can create a child process on any other node on the network.

The kernel handles process scheduling based on process priorities and three scheduling classes: FIFO, round-robin, and adaptive. The highest priority process that is ready to run will be given the processor. If there are multiple processes at the same highest priority, the scheduling classes of each is used to determine which should be run. Each priority has an associated ready-to-run queue. FIFO and round-robin schedule processes similarly, but FIFO has no notion of a timeslice – a process runs until it is preempted or it yields control (by making any kernel call). Adaptive scheduling is a form of priority degradation; it lowers a process' priority by one when it consumes its timeslice, and raises it again when it blocks. This is proposed as a good policy for compute-bound processes mixed with interactive processes. The scheduler is run whenever a process becomes unblocked, a timeslice expires, or a running process is preempted (by a higher priority process).

6.3 Memory Management

There was no information available about how QNX performs memory management. The kernel does not provide swapping by itself, however, so we presume that while the kernel performs memory allocation, a user-level process provides swapping.

7 Comparisons

In this section we draw conclusions about which microkernels offer better support for each of the five design problems of distributed systems described in Section 2: (1) network transparency, (2) communication, (3) process control, (4) scalability, and (5) fault-tolerance. For certain problems, we found that a particular microkernel offered key advantages. More generally, we found that the fundamental design of microkernels offers solutions to these problems.

7.1 Network Transparency

All three microkernels examined in this paper implement facilities for sending messages across the network transparently. This service is implemented in the form of user-level network proxies at each node.

The flexibility of this scheme is demonstrated by the network paging service offered by Mach and Chorus. The addition of the network paging facilities came at little cost because the existing paging facilities already used messages to communicate with the microkernel. The only necessary addition was a network messaging server.

Network paging is not the only service that benefits from the presence of a messaging server; the entire design of microkernels is centered around messaging, so the majority of the services are easily distributed in this manner.

7.2 Communication

Distributed applications on a network require message passing for communication. The efficient message passing of microkernels can support this requirement of distributed applications.

All three microkernels include interesting optimizations for communication. For example, both Chorus and Mach offer copy-on-write semantics for message passing. QNX, however, performs fewer memory copies due to a combination of synchronized messages and the MX structure. Although we lack performance data, we believe this enhancement gives QNX an advantage in shared memory message passing.

7.3 Remote Process Management

We found no properties inherent to microkernels that address the issue of remote process management. While all three systems we investigated have the ability to create and communicate with remote processes, which is valuable, we do not believe that these features are inherent to microkernels.

Additionally, we found that none of the systems allowed process migration across a network, which we thought could hinder computational load balancing. Since microkernels have a decoupled design with an emphasis on network transparency, we believe that process migration should be easier to implement. We would like to explore this area further to determine why process migration has not been implemented and what its associated costs and benefits are.

7.4 Scalability

Since many of the services of a microkernel are implemented as user-level processes, it is possible to configure nodes of a distributed network with different services. This minimizes the number of resources dedicated to services that may never be used by a node. Furthermore, it is possible to achieve this without compiling a different version of the kernel for different nodes.

The port abstraction can also provide a measure of scalability. Consider a set of servers which share and listen on a single port. It is easy to add additional servers that listen on the same port. Even though both Mach and Chorus offer ports, only Chorus allows multiple processes to receive on a single port.

It is interesting to note that QNX scales *down* very well. Its small memory footprint makes it useful in embedded systems where the other kernels would not fit.

7.5 Fault-Tolerance

Microkernel design offers fault-tolerance in several ways related to the decoupled structure. Because most operating system services run in their own protected address spaces, the failure of one will not affect the rest of the system. Additionally, because these services run as user-level processes, they can be stopped and restarted if failures do occur, while the rest of the system can make progress. Finally, the simplicity of microkernels lends to their correctness and reliability.

The port abstraction of both Mach and Chorus provides further potential for fault-tolerance. In the event of a failure in either Mach or Chorus, the kernel can redirect all future messages to the failed process to another process that provides the same service. QNX lacks this ability because it uses process IDs to determine the destination of messages.

8 Conclusion

In this paper, we compared Mach, Chorus, and QNX in the areas of process management, IPC, and memory management, to see how these systems dealt with certain problems of distributed processing. While we found some interesting differences in how the three microkernels solved these problems, we also found that the three microkernels did not differ greatly from our model of a typical microkernel. The strengths of each system drew more from sound microkernel implementation than unique optimizations.

As a subject for future work, we are interested in studying the potential benefits and costs of process migration in the context of distributed systems.

We would also like to briefly mention a few of the shortcomings of our paper. First, we chose not to cover some important problems of distributed computing, particularly security and device I/O management. Second, due to lack of funding, we were unable to gather relevant performance data to substantiate some of our claims.

References

1. F. Armand, R.W. Dean. Data Movement in Kernelized Systems. In *Proceedings of the Usenix Workshop on Micro-kernels and Other Kernel Architectures*, April 1992.
2. F. Armand, M. Guillemont, and P. Leonard. Towards a distributed UNIX System – the CHORUS Approach. In *EUUG Autumn Conference*, pp. 413-431, 1986.
3. A. Bricker et al. A new look at micro-kernel-based UNIX operating systems: Lessons in performance and compatibility. In *Proc. of the EurOpen Spring'91 Conference*, May 1991.
4. D. Golub, R. Dean, A. Forin, and R. Rashid. Unix as an Application Program. In *Proceedings of the USENIX Summer Conference*, June, 1990
5. F. Herrmann et al. Chorus, a New Technology for Building UNIX Systems. In *EUUG Autumn Conference*, 1988.
6. QNX System Architecture. Available WWW <URL: http://www.qnx.ca/literature/qnx_sysarch/index.html> (1999).
7. Symmetric Multi-Processing (SMP) with the QNX Neutrino Microkernel RTOS. Available WWW <URL: http://www.qnx.ca/literature/whitepapers/smp.html> (1999).
8. A. Tanenbaum. A Comparison of Three Microkernels. In *The Journal of Supercomputing*, July 1995.
9. A. Tanenbaum. Case Study 4: MACH. In *Modern Operating Systems*, pp. 637-680. ISBN 0-13-588187-0.
10. M. Young et. al. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proceedings of the 11th Operating Systems Principles*, November, 1987