From RIG to Accent to Mach: The Evolution of A Network Operating System

Richard F. Rashid

Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pa. 15213

Abstract

This paper describes experiences gained during the design, implementation and use of the CMU Accent Network Operating System, its predecessor, the University of Rochester RIG system and its successor CMU's Mach multiprocessor operating system. It outlines the major design decisions on which the Accent kernel was based, how those decisions evolved from the RIG experiences and how they had to be modified to properly handle general purpose multiprocessors in Mach. Also discussed are some of the major issues in the implementation of message-based systems, the usage patterns observed with Accent over a three year period of extensive use at CMU and a timing analysis of various Accent functions.

1. Background

Mach is a multiprocessor operating system kernel currently under development at Carnegie-Mellon University. In addition to binary compatibility with Berkeley's current UNIX 4.3 bsd release, Mach provides a number of new facilities not available in 4.3, including:

- Support for tightly coupled and loosely coupled general purpose multiprocessors.
- An internal symbolic kernel debugger.
- Support for transparent remote file access between autonomous systems.
- Support for large, sparse virtual address spaces, copy-on-write virtual copy operations, and memory mapped files.
- Provisions for user-provided memory objects and pagers.
- Multiple threads of control within a single address space.
- A capability-based interprocess communication facility integrated with virtual memory management to allow transfer of large amounts of data (up to the size of a process address space) via copy-on-write techniques.
- Transparent network interprocess communication with preservation of capability protection across network boundaries.

As of May 1986, Mach runs on most uniprocessor VAX architecture machines: VAX 11/750, 11/780, 11/785, 8200, 8600, 8650, MicroVAX I, and MicroVAX II. Mach also runs on two multiprocessor VAX machines, the four (11/780 or 11/785) processor VAX 11/784 with 8 MB of shared memory the VAX 8300 (with up to 4 processors). Mach has already been ported to the IBM RT/PC and work has begun on ports to the uniprocessor SUN 3 and multiprocessor Encore MultiMax. The current version of the system, Mach-1, includes all of the features listed above and is in production use by CMU researchers on a number of projects including a multiprocessor speech recognition system called Agora [5] and a project to build parallel production systems.

Mach is the logical successor to CMU's Accent [16, 17] kernel -- an operating system designed to support a large network of uniprocessor scientific personal computers. The design and implementation of Accent was in turn based on experiences gained during the development of the University of Rochester's RIG system [3, 14], a message-based network access machine. Both RIG and Accent have seen considerable use over the years. RIG provided a variety of functions including terminal support and remote file access within the Rochester environment until early this year when the last RIG machine was decommissioned. Accent continues in use at CMU as the basic operating system for a network of 150 PERQ workstations and has seen commercial use in printing and publishing workstations as well as engineering design systems. As a third generation network operating system Mach, benefits from the lessons learned in over ten years of design, implementation and use of RIG and Accent. This paper summarizes the lessons of those systems and their impact on the design and implementation of Mach.

2. The Evolution of Accent from RIG

Implementation of RIG began in 1975 on an early version of the Data General Eclipse mini-computer. The first usable version of the system came on-line in the fall of 1976. Eventually the Rochester network included several RIG Eclipse nodes as network servers and a number of Xerox Altos acting as RIG client hosts. RIG provided clients network file services, ARPANET access, printing services and a variety of other functions. Active development continued well into the 1980's but obsolescence of its Data General Eclipse and Xerox Alto hardware base eventually dictated its demise in the Spring of 1986.

2.1. The RIG Design

The basic system structuring tool in RIG was an interprocess communication (IPC) facility which allowed RIG processes to communicate by sending packets of information between themselves. RIG's IPC facility was defined in terms of two basic abstractions: *messages* and *ports*.

CH2345-7/86/0000/1128$01.00 © 1986 IEEE

A RIG port was defined to be a kernel-provided queue for messages and was referenced by a global identifier consisting of a dotted pair of integers *<process number.port number>*. A RIG port was protected in the sense that it could only be manipulated directly by the RIG kernel, but it was unprotected in the sense that any process could send a message to a port. A RIG port was tied directly to the RIG abstraction of a *process* -- a protected address space with a single thread of program control.

A RIG message was composed of a header followed by data. Messages were of limited size and could contain at most two scalar data items or two array objects. The type tagging of data in messages was limited to a small set of simple scalar and array data types. Port identifiers could be sent in messages only as simple integers which would then be interpreted by the destination process.

Due largely to the hardware on which it was implemented, RIG did not allow either a paged virtual memory or an address space larger than 2↑16 bytes. RIG did, however, use simple memory mapping techniques to move data [3]. The largest amount of data which could be transferred at a time was 2K bytes.

2.2. Problems with RIG

The RIG message passing architecture was originally intended more as a means for achieving modular decomposition (much like Brinch-Hansen's RC4000) rather than as the basis for a distributed system. It was discovered early on, though, that RIG's message passing facility could be adapted as the communication base for a network operating system. Unfortunately, as RIG became heavily used for networking work at Rochester a number of problems with the original design became apparent:

- **Protection**

  The fact that ports were represented as global identifiers which could be constructed and used by any process implied that a process could not limit the set of processes which could send it a message. To function correctly, each process had to be prepared to accept any possible message sent to it from any potential source. A single errant process could conceivably flood a process or even the entire system with incoherent messages.

- **Failure notification**

  Another difficulty with global identifiers was that they could be passed in messages as simple integers. It was therefore impossible to determine whether a given process was potentially dependent on another process. In principle any process could store in its data space a reference to any other process. The failure of a machine or a process could therefore not be signaled back to dependent processes automatically. Instead, a special process was invented which ran on each machine and was notified of process death events. Processes had to explicitly register their dependencies on other processes with this special "grim reaper" process in order to receive event-driven notifications.

- **Transparency of service**

  Because ports were tied explicitly to processes, a port defined service could not be moved from one process to another without notifying all parties. Transparent network communication was also compromised by this naming scheme. A port identifier was required to explicitly contain the network host identifier as part of its process number field. As the system expanded from one machine to one network to multiple interconnected networks this caused the port identifier to expand in size -- usually resulting in considerable reimplementation work.

- **Maximum message size**

  The limited size of messages in RIG resulted in a style of interprocess interaction in which large data objects (such as files) had to be broken up into chunks of 2K bytes or less. This constraint impacted on the efficiency of the system (by increasing the amount of message communication) and on the complexity of client/server interactions (e.g., by forcing servers to maintain state information about open files).

2.3. The evolution of RIG

CMU's Spice [8] distributed personal workstation project provided an oportunity to effectively "redo" a RIG-like system taking into account that system's limitations. The result was the Accent operating system kernel for the PERQ Systems Corporation PERQ computer.

The Accent solution to the problems present in the RIG design was based on two basic ideas:

1. **Define ports to be capabilities as well as communication objects.**

   By providing processes with capabilities to ports rather than a global identifier for them, it was possible to solve at one time the problems of protection, failure notification and transparency:

   - Protection in Accent is provided by allowing processes access only to those ports for which they have been given capabilities.

   - Processes can be notified automatically when a port disappears on which those processes are dependent because the kernel now has complete knowledge of which processes have access to each port in the system. There is no hidden communication between processes.

   - Transparency is complete because the ultimate destination of a message sent to a port is unknown by the sender. Thus transparent intermediary processes can be constructed which forward messages between groups of processes without their knowledge (either for the purpose of debugging and monitoring or for the purpose of transparent network communication).

2. **Use virtual memory to overcome limitations in the handling of large objects.**

   The use of a large address space (a luxury not possible in the design of RIG) and copy-on-write memory mapping techniques permits processes to transmit objects as large as they can directly access themselves. This allows processes such as file servers to provide access to large objects (e.g., files) through a single message exchange -- drastically reducing the number of messages sent in the system [9].

The first line of Accent code was written in April 1981. Today Accent is used at CMU in a network of 150 PERQ workstations. In addition to network operating system functions such as distributed process and file management, window management and mail systems, several applications have been built using Accent. These include research systems for distributed signal processing [10], distributed speech understanding [5] and distributed transaction processing [18]. Four separate programming environments have been built -- CommonLisp, Pascal, C and Ada -- including language support for an object-oriented remote procedure call facility [12].

3. The Accent Design

Accent is organized around the notion of a protected, message-based interprocess communication facility integrated with copy-on-write virtual memory management. Access to all services and resources, including the process management and memory management services of the operating system kernel itself, are provided through Accent's communication facility. This allows completely uniform access to such resources throughout the network. It also implies that access to kernel provided services is indistinguishable from access to process provided resources (with the exception of the interprocess communication facility itself).

3.1. Interprocess communication

The Accent interprocess communication facility is defined in terms of abstractions which, as in RIG, are called *ports* and *messages*.

The *port* is the basic transport abstraction provided by Accent. A port is a protected kernel object into which messages may be placed by processes and from which messages may be removed. A port is logically a finite length queue of messages sent by a process. Ports may have any number of senders but only one receiver. Access to a port is granted by receiving a message containing a port capability (to either send or receive).

Ports are used by processes to represent services or data structures. For example, the Accent window manager uses a port to represent a window on a bitmap display. Operations on a window are requested by a client process by sending a message to the port representing that window. The window manager process then receives that message and handles the request. Ports used in this way can be thought of as though they were capabilities to objects in a object oriented system(Jones78). The act of sending a message (and perhaps receiving a reply) corresponds to a cross-domain procedure call in a capability based system such as Hydra [2] or StarOS [11].

A *message* consists of a fixed length header and a variable size collection of typed data objects. Messages may contain both port capabilities and/or imbedded pointers as long as both are properly typed. A single message may transfer up to 2↑32 bytes of by-value data.

Messages may be sent and received either synchronously or asynchronously. A software interrupt mechanism allows a process to handle incoming messages outside the flow of normal program execution.

Figure 3-1 shows a typical message interaction. A process A sends a message to a port P2. Process A has send rights to P2 and receive rights to a port P1. At some later time, process B which has receive rights to port P2 receives that message which may in turn contain send rights to port P1 (for the purposes of sending a reply message back to process A). Process B then (optionally) replies by sending a message to P1.

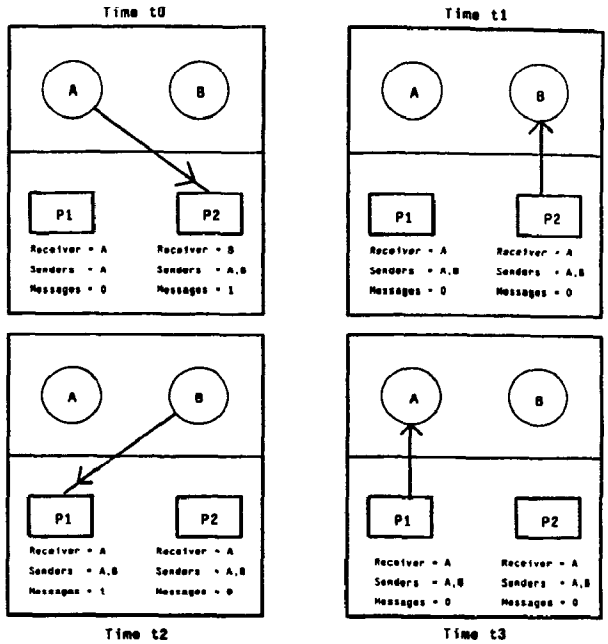

Figure 3-1: Typical message exchange

Should port P2 have been full, process A would have had the option at the point of sending the message to: (1) be suspended until the port was no longer full, (2) have the message send operation return a port full error code, or (3) have the kernel accept the message for future transmission to port P2 with the proviso that no further message can be sent by that process to P2 until the kernel sends a message to A telling it the current message has been posted.

3.2. Virtual memory support

Accent provides a 2↑32 byte paged address space for each process in the system and a 2↑32 byte paged address space for the operating system kernel. Disk pages and physical memory can be addressed by the kernel as a portion of its 2↑32 byte address space. Accent maintains a virtual memory table for each user process and for the operating system kernel. The kernel's address space is paged and all user process maps are kept in

paged kernel memory. Only the kernel virtual memory table, a small kernel stack, the PERQ screen, I/O memory and those PASCAL modules required for handling the simplest form of page fault need be locked in physical memory, although in practice parts of the kernel debugger and symbol tables for locked modules are also locked to allow analysis of system errors. The total amount of kernel code and symbol table information locked is 64K bytes [9].

Whenever large amounts of data (the threshold is a system compile-time constant normally set at 1K bytes) are transmitted in a message, Accent uses memory mapping techniques rather than data copying to move information from one process to another within the same machine. The semantics of message passing in Accent imply that all data sent in a message are logically copied from one address space to another. This can be optimized by the kernel by mapping the sent data copy-on-write in both the sending and receiving processes.

Figure 3-2 shows a process A sending a large (for example 24 megabyte) message to a port P1. At the point the message is posted to P1, the part of A's address space containing the message is marked copy-on-write -- meaning any page referenced for writing will be copied and the copy placed instead into A's virtual memory table. The copy-on-write data then resides in the address space of the kernel until process B receives the message. At that point the data is removed from the address space of the kernel. By default, the operating system kernel determines where in the address space of B the newly received message data is placed. This allows the kernel to minimize memory mapping overhead. Any attempt by either A or B to change a 512 byte page of this copy-on-write data results in a copy of that page being made and placed into that process' address space.

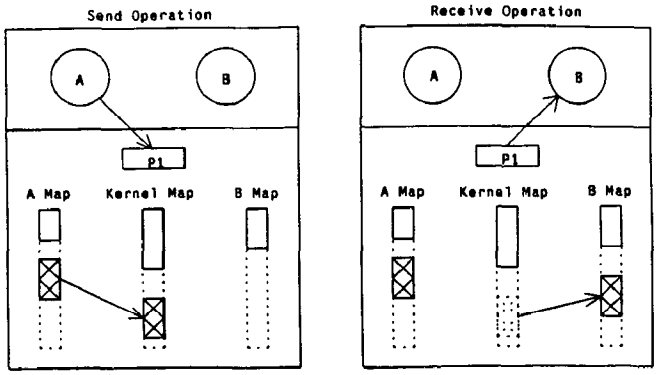

Figure 3-2: Mapping operations during message transfer

3.3. Network communication

The abstraction of communication through ports permits the distinction between access to local and remote resources to be completely invisible to a client process. In addition, Accent exploits the integration of memory management and IPC to provide a number of options in the handling of virtual memory, including the ability to allow memory to be sent copy-on-reference across a network. Each entry of an Accent virtual memory table maps a contiguous region of process virtual memory to a contiguous portion of an Accent *memory object*. A memory object is the basic unit of secondary storage in Accent. Memory objects can be contiguous physical memory (as used for the PERQ screen or I/O buffers) or a randomly addressed disk file. A memory object can also be backed not by disk or main memory, but by a process through a port. Initial references to a page of data mapped to a port are trapped by the kernel and a request for the necessary data is forwarded in a message on that port. This feature allows processes to provide the system with virtual memory that they themselves maintain (either locally or over a network connection to another machine). In this way network communication servers can provide copy-on-reference network transmission of pages in a large message.

4. Key Implementation Issues in Accent

Many of the implementation decisions made in Accent were based on experiences with RIG. Nevertheless, the addition of virtual memory and capability management to the RIG design made it unclear how the RIG experiences would extrapolate to the Accent environment.

4.1. IPC Implementation

The actual implementation of the message mechanism relied on several assumptions about the use of messages:

- the average number of messages outstanding at any given time per process would be small,
- the number of port capabilities needed by a process could vary from two to several hundred, and
- the use of simple messages (meaning messages which contained port capabilities only in their header and which contained less than a few kilobytes) would so dominate complex messages that simple messages would be an important special case.

Each of these assumptions had held true for RIG [4, 14]. It was hoped that although Accent provided a substantially different application environment than RIG, the RIG experiences would provide a reasonable prediction of Accent performance.

Given these expectations, the implementation was optimized for anticipated common cases, including:

- The assumption that there would seldom be more than one message waiting for a process at a time led to an implementation in which messages are queued in per-process rather than per-port queues.
- To allow large numbers of ports per process and fast lookup, port capabilities are represented as indexes into a global port record array stored in kernel virtual memory. Port access is protected through the use of a bitmap of process access rights kept per port (the number of processes is much less than the number of ports).
- The assumption that simple messages would be an important special case led to the addition of a field to the message header so that user processes can indicate whether or not a message is simple and thus allow special handling by the kernel.

These usage assumptions did in fact prove true for Accent. Table 4-1 demonstrates the properties of Accent message passing as measured during an active day of use.

1.01	Average probes to requested message
33.42	Average port rights held per process
14.38	Average ports owned per process
0.094	Ratio of complex to simple messages

Table 4-1: Message use statistics

4.2. Virtual Memory Implementation

The lack of sophisticated virtual memory management in RIG (and in fact in nearly all message-based systems of that era) meant that Accent could not benefit from previous experience with virtual memory use resulting from message operations. Instead, the design of Accent's virtual memory implementation grew out of simple assumptions based purely on intuition. These initial assumptions influenced the design of the Accent virtual memory implementation:

- process maps had to be compact, easy to manipulate and support sparse use of a process address space,
- the number of contiguously mapped regions of the address space would be reasonably small, and
- large amounts of memory would frequently be passed copy-on-write in messages.

The Accent process virtual memory map is maintained as a two-level indirect table terminating in linked lists of entries (see Figure 4-1). Each entry on the linked list maps a contiguous portion of process virtual memory into contiguous regions of Accent memory objects. The map is organized so that large portions can be validated, invalidated or copied without having to modify the linked lists of map entries. This is accomplished by having valid, copy-on-write and write-protect bits at each level of the table. During lookup, these bits are "ored" together. Thus all of memory can be efficiently made copy-on-write by just setting the copy-on-write bits of valid entries in level one of the process map table. Figure 4-1 illustrates the translation of a virtual address to an offset within a memory object.

Physical memory in Accent is used as a cache of secondary storage. There are no special disk buffers. Access to all information (e.g., files) is through message passing (and subsequent page faulting if necessary).

This scheme is flexible enough to be used internally by the kernel to remap portions of its own address space. An entire process virtual memory map, for example, is copied in a fork

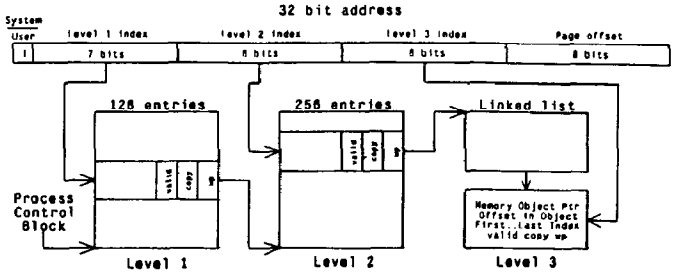

Figure 4-1: Mapping a virtual address in Accent

operation without physically copying the map by using Accent's copy-on-write facility. To reduce map manipulation overheads, changes caused by copy-on-write updates are recorded first in a virtual to physical address translation table (kept in physical memory) and are not incorporated into a process map until the relevant page must be written out to secondary storage.

Copy-on-write access to memory objects is provided through the use of *shadow* memory objects which reflect page differences between a copied object and the object it shadows (which could in turn be a shadow). Disk space for newly created pages or pages written copy-on-write is allocated on an as-needed basis from a special paging area. No disk space is ever allocated to back up a process address space unless the paging algorithms need to flush a dirty page. See figure 4-2.

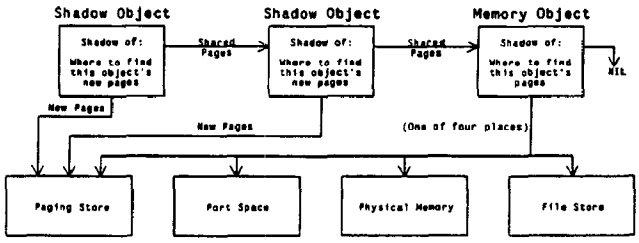

Figure 4-2: An example of memory object shadowing

Most shadow memory objects are small (under 32 pages). Most large shadows contain only a few pages of data different from the objects they shadow. These facts led to an allocation scheme in which small shadows are allocated contiguously from the paging store and larger shadows use a page map and are allocated as needed.

Overall, the basic assumptions about the use of process address space in Accent appear to hold true. The typical user process table:

- is between 1024 and 2048 bytes in size,
- contains 34-70 mapping entries, and
- maps a region of virtual memory approximately eight megabytes in extent (in PERQ PASCAL each separately compiled module occupies a distinct 128K byte region of memory) and about one to two megabytes in size.

Although all memory is passed copy-on-write from one process to another, the number of copy-on-write faults is typically small. A typical PASCAL compile/link/load cycle, for example, requires only slightly more than one copy-on-write fault per second. Clearly most of the data passed by copy in Accent is read and not written. The result is that the logical advantages of copy-on-write are obtained with costs similar to that of mapped shared memory [6].

4.3. Programming Issues

One of the problems with message based systems has traditionally been the fact that existing programming languages do not support their message semantics. In RIG, a special remote

procedure call function was provided called "Call" [13] which took as its arguments a message identifier, a process-port identifier, and operation arguments along with their type information. One of the early decisions in the implementation of Accent was to define all interprocess message interfaces in terms of a high-level specification language. The properties of ports allow them to be viewed as object references. The interprocess specification language is defined in terms of operations on objects. Subsystem specifications in this language are compiled by a program called Matchmaker into remote procedure call stubs for the various programming languages used in the system -- currently C, PASCAL, ADA and Common LISP. The result is that all interprocess interfaces look to the programmer as though they were procedural interfaces in each of these languages. In PASCAL, for example, the interface procedure for writing a string to a window of the screen would look like:

WriteString(window,string-to-be-written)

All Matchmaker specified calls take as their first argument the port object on which the operation is to be performed. The remote procedure call stub then packages the request in a message, sends it to the port, and waits for a reply message (if necessary). Initial access to server ports is accomplished either through inheritance (by having the parent process send port rights to its children) or by accessing a name server process (a port for which is typically passed to a process by inheritance). A complete description and specification of Matchmaker can be found in [12].

Matchmaker's specification language allows both synchronous and asynchronous calls as well as the specification of timeouts and exception handling behavior. It supports both by-value and by-value-result parameters. It allows types to be defined as well as the specification of their bit packing characteristics in the message. For the server process, Matchmaker produces routines which allow incoming messages to be decoded and server subroutines automatically invoked with the proper arguments.

The support provided by Matchmaker is similar to some of the features which have been introduced in modern languages for managing multiple tasks such as the ADA rendezvous mechanism [1]. Matchmaker, however, supports a number of different programming languages and provides a much greater range of options for synchronous and asynchronous behavior in a distributed environment.

Despite the obvious simplicity of simple "remote procedure call" style interfaces, a suprisingly high percentage of network operating system interfaces take advantage of the asynchronous form of Matchmaker interfaces. Of 225 system interfaces:

- 170 (approximately 77 percent) are synchronous,
- 45 (approximately 19 percent) are asynchronous and
- 10 (approximately 4 percent) represent exceptions.

Runtime statistics show that over 50 percent of messages actually sent during normal system execution are sent as part of asynchronous Matchmaker specified operations -- normally due to the behaviour of I/O subsystems (such as handlers for the PERQ keyboard and display) or basic system servers (such as network protocol servers).

Matchmaker server interfaces account for approximately 10 percent of the total network operating system code -- roughly 75.5k bytes out of 757k bytes. For the Accent kernel itself, the Matchmaker interface is 10280 bytes out of approximately 115k bytes. Runtime costs are considerably less. During a PASCAL compilation, for example, less than 2 percent of CPU time is due to Matchmaker interface overheads.

4.4. Key Statistics

4.4.1. Hardware and basic system performance of Accent

Table 4-2 compares the relative performance of PERQ and VAX-11/780 CPUs. Timings were performed in PASCAL on the PERQ and in C on a VAX running UNIX 4.1bsd.

PASCAL programs written for the PERQ range in overall speed from 1/5 to 1/3 the speed of comparable programs on the VAX 11/780, depending on whether 16-bit or 32-bit operations predominate. In fairness to the PERQ hardware, the underlying microengine is much faster than the PASCAL timings in table 4-2 would indicate. Microcoded operations often run as fast as or faster than equivalent VAX 11/780 assembly language. Note, for example, the relative speeds of the microcoded context switch and kernel trap operations. Moreover, instruction sets better tuned to the PERQ hardware, such as the Accent CommonLisp instruction set, run at speeds closer to 50 percent of the VAX. Nevertheless, for the purpose of gauging the performance of the Accent kernel code, which is written in PASCAL and makes heavy use of 32-bit arithmetic, pointer chasing and packed field accessing, the CPU speed of a PERQ is about 1/5 that of a VAX 11/780.

Perq	Vax	Ratio	Operation
2300ns	720ns	.31	Tick (32-bit stack local)
12us	4us	.25	Simple loop (16-bit integer)
20us	3us	.17	Simple loop (32-bit integer)
35us	20us	.57	Null procedure call/return
75us	25us	.33	Procedure call with 2 arguments
80us	400us	5.00	Context switch
132us	264us	2.00	Null kernel trap
30s	9s	.30	Baskett Puzzle Program (16-bit)
50s	10s	.20	Baskett Puzzle Program (32-bit)

Table 4-2: Comparison of Perq and Vax-11/780 operation times

4.4.2. IPC Costs

Table 4-3 shows the costs of various forms of message passing in Accent. As was previously described, Accent distinguishes between *simple* and *complex* messages to improve performance of common message operations. Simple messages are defined to be those with less than 960 bytes of in-line data that contain no pointers or port references (other than those in the message header). Other messages are considered complex. The times for complex messages listed in the table were measured for messages containing one pointer to 1024 bytes of data. The observed ratio of simple to complex messages in Accent is approximately 12-to-1.

Time	IPC Operation
1.15	Simple message send
1.35	Simple message receive
10.	Complex message send (1024 bytes)
10.	Complex message receive (1024 bytes)

Table 4-3: IPC operation times in milliseconds

The average number of messages per second observed during periods of heavy standard version use (e.g., compilation) is less than 30. There were 67378 simple messages and 4279 complex messages sent during one measurement of three hours of editing, network file access, and text formatting, an average of less than eight per second [9].

4.4.3. Accessing file data

One of the reasons for the relatively low message rate of message exchange in Accent is the heavy reliance on virtual memory mapping techniques for transferring large amounts of data in messages. A process making a request for a large file typically receives the entire file in a single message sent back from a file server process. As a result, all file access in Accent is mediated through the memory management system. There are no separate file buffers maintained by the system or special operations required for file access versus access to other forms of process mapped memory. By contrast, in RIG the same operation would have required as many message exchanges between client and server as there were pages in the file.

Table 4-4 shows the costs associated with reading a 56K byte file under UNIX 4.1bsd on a VAX 11/780 with a 30 millisecond average access time Fujitsu disk and under the standard version of Accent with a 30 millisecond average access time MAXSTORE drive.

The measured cost of a file access in Accent as shown in table 4-4 is due, in part, to the cost of a disk write to update the file access time. This disk write is unbuffered in Accent and thus is included in the file request time. The Unix disk write associated with an open is buffered and is excluded from the open/close time.

Accent file access speed is limited by the basic fault time of about four milliseconds (see table 4-5), the average number of consecutive file pages on a disk track and the cost of making new

System	Time	Operation
Accent	66	Request file from server
UNIX 4.1	5-10	Open/close
Accent	5-10	Read a page (512 bytes)
UNIX 4.1	16-18	Read a page (1024 bytes)
UNIX 4.2	16-18	Read a page (4096 bytes)

Table 4-4: File access times in milliseconds

VP entries. Its page size is only 512 bytes, in contrast to 1024 bytes for 4.1bsd and 4096 or 8192 for 4.2bsd.

Once mapped, file access in Accent ranges from somewhat faster than 4.1bsd to slightly slower, depending on the locality of file pages. 4.2bsd file access [15] is considerably faster than either 4.1bsd or Accent. This increase in speed appears to be due almost entirely to the larger (typically 4096 byte) file page size. The actual number of disk I/O operations per second under 4.2 is almost identical to 4.1, about 50-60 per second, and appears to be bounded by the rotational speed of the disk (60 revolutions per second).

4.4.4. Fault handling and copy-on-write

Table 4-5 summarizes the results from test programs that caused 100,000 instances of a variety of memory fault types. It shows the average total times required to handle single faults.

Total	Type of fault
0.623	Null fault
3.355	Read fault, zero fill
3.704	Write fault, zero fill
3.760	Read fault, memory fill, small file
4.504	Read fault, memory fill, large file
3.833	Write fault, CopyOnWrite copy

Table 4-5: Fault handling times in milliseconds

Overall, the costs of copy-on-write memory management are nearly identical to that of by-reference memory mapping. Less than 0.01 percent of the total time associated with an entire rebuilding of the operating system and user programs from source is used to handle copy-on-write faults [9].

5. Mach: Adapting Accent to Multiprocessors

Accent went beyond demonstrating the feasibility of the message passing approach to building a distributed system. Experience with Accent showed that a message based network operating system, properly designed, can compete with more traditional operating system organizations. The advantages of this approach are system extensibility, protection and network transparency.

By the fall of 1984, however, it became apparent that, without a new hardware base, Accent would eventually follow RIG into oblivion. Hastening this process of electronic decay was Accent's inability to completely absorb the ever burgening body of UNIX developed software both at CMU and elsewhere -- despite the existence of a "UNIX compatibility" package.

Mach was conceived as an Accent-like operating system which would provide complete UNIX compatibility. It was also designed to better accommodate the kind of general purpose shared-memory multiprocessors which appear to be on their way to becoming the successors to traditional general purpose uniprocessor workstations and timesharing systems.

5.1. The design of Mach

The design of Mach differs from that of Accent in several crucial ways:

- The Accent notion of a process, which like RIG is an address space and single program counter, was split into two new concepts:
  1. a *task*, which is the basic unit of resource allocation including a paged address space, protected access to system resources (such as processors, ports and memory), and
  2. a *thread*, which is the basic unit of CPU utilization.
- A facility for handling a form of structured sharing of read/write memory between tasks in the same family tree was added to allow finer granularity synchronization than could be achieved with a kernel provided mechanism.
- The Mach IPC facility was further simplified. This came about as the logical result of using thread mechanisms to handle some forms of asynchrony and error handling (much as was done in the V Kernel [7]).
- The notion of memory object was generalized to allow general purpose user-state external pager tasks to be built.

These design modifications are a consequence of handling shared-memory multiprocessor architectures. Accent provided no tool for fine grain synchronization or lightweight processes. Both are important for effective use of multiprocessor cycles in a variety of applications.

Despite these changes, the basic features which allowed Accent to provide uniform access to both local and network resources are still in place. This allows networks of multiprocessors or of multiprocessors and uniprocessors to be built using the same basic system abstractions. As in Accent, operations on all Mach objects other than messages are performed by sending messages to ports which are used to represent them. For example, the act of creating a task or thread returns access rights to the port which represents the new object and which can be used to manipulate it. A thread can suspend another thread by sending a suspend message to that thread's *thread port*, even across a network boundary.

Tasks are related to each other in a tree structure by task creation operations. Virtual memory may be marked as inheritable to a task's children. Memory regions may be inherited read-write, copy-on-write or not at all. A standard UNIX fork operation, for example, takes the form of a task with one thread creating a child task with a similar single thread of control and all its memory shared copy-on-write.

The notions of multiple threads of control within a task and limited sharing between task allows Mach to provide three levels of synchronization and communication: fine grain, intra-application interprocess communication and inter-application interprocess communication.

Fine grain communication is performed on memory shared either within a task or between related tasks. Mach provides a library to support synchronization on shared memory to avoid the cost of kernel trap operations on short-term locks. Network read/write shared memory is not provided by the kernel, but is potentially implementable by a user-state process acting as an external object pager (see discussion of object pagers below).

Intra-application inter-thread communication is performed using the standard Send and Receive ports primitives but can be implemented more efficiently in the presence of shared libraries and memory. By the nature of the abstractions, threads can ignore the difference between intra-application communication and inter-application communication.

Inter-application communication requires the intervention of the Mach kernel to provide protection. As in Accent, large amounts of data in messages may be mapped copy-on-write from one address space to another rather than copied. Data forwarded in messages over the network can be transmitted on reference rather than all at once at the discretion of the network server.

5.2. Implementation

5.2.1. Virtual memory modifications

While system analysis indicated that the basic Accent virtual memory scheme worked well, it also demonstrated that the data structure used to represent an Accent process map -- a two-level indirect table terminated in linked lists of mapping descriptors -- was unnecessarily complicated. Because nearly all operations on maps are sequential and maps seldom get very large, Mach implements task address maps as simple ordered lists of mapping descriptors. Each descriptor maps a range of virtual addresses to a range of bytes in a memory object. The only non-sequential operation -- lookup events due primarily to memory faults -- is sped by the use of hints based on previous lookup requests.

Another innovation of Mach over Accent is in the use of *sharing maps* to represent read/write shared regions between tasks. A Mach mapping descriptor may point either directly to a memory object (which can then only be shared copy-on-write) or indirectly to memory objects through a sharing map. A sharing map is simply an address map which maps a range of virtual addresses shared by at least two task address maps. All operations on tasks maps in a shared range of addresses are performed through indirection on sharing maps.

Overall, the Mach data structures are simpler, more compact and more expressive than those of Accent. A Mach address map can be thought of as a simple run-length encoding of a process address space. A typical UNIX-style process can be expressed in less than 100 bytes.

5.2.2. Mach IPC

The introduction of the notion of tasks and threads into Mach necessitated some changes to Accent's basic IPC facility. Port access rights in Mach are owned by a task. All threads within a task may therefore send or receive messages on that task's ports.

The availability of threads to manage asynchronous activities simplified handling of software interrupts. Moreover, several message options, such as message priorities and the ability to preview the contents of a message before it had to be received, had been found to be largely unused for their intended purpose in Accent and have been removed.

5.2.3. Managing hardware diversity

Mach was intended from the outset to handle a wide diversity of both uniprocessor and multiprocessor hardware. For example, Mach provides a task memory sharing and a thread memory sharing model for multiprocessor memory synchronization. This allows Mach to support both multiprocessors which support full memory sharing with cache consistency as well as machines with only partial sharing or explicit memory caching. In practice, the system already is configured to handle a wide range of uniprocessor and multiprocessor VAX configurations. The same binary kernel image is used on both uniprocessor and multiprocessor systems.

Mach also handles another form of diversity. Messages, because they contain tagged data, are transformed from one machine data format to another by network servers. Properly typed Matchmaker interfaces allow programs written on an RT PC to communicate with VAX applications despite different byte ordering, data packing and data format conventions. There are, however, limits on this form of machine independence. For example, no attempt is made to preserve precision of floating point numbers converted from one form to another.

5.2.4. Confronting UNIX

One mechanism for ensuring Mach's survival in the face of a flood of UNIX based software is to make certain that it is compatible with an existing UNIX environment. This was achieved by building Mach to allow UNIX 4.3bsd system calls to be handled in much the same way they would be handled in a completely native system. The Mach kernel effectively supplants the basic system interface functions of the UNIX 4.3bsd kernel: trap handling, scheduling, multiprocessor synchronization, virtual memory management and interprocess communication. 4.3bsd functions are provided by kernel-state processes which are scheduled by the Mach kernel and share communication queues with it. Work is now underway to remove non-Mach UNIX functionality from kernel-state and provide these services through user-state processes.

6. Conclusions

The evolution of network operating systems from RIG through Mach was, in a sense, driven by the evolution of distributed computer systems from small networks of minicomputers in the middle 1970s to large networks of personal workstations and mainframes in the early 1980s to networks of uniprocessor and multiprocessor systems today. Not suprisingly, the basic software primitives of Mach -- task, thread, port, message and memory object -- parallel the hardware abstractions which characterize modern distributed systems -- nodes, processors, network channels, packets and primary and secondary memory. Experiences, both good and bad, with RIG and Accent have played an important role in determining the exact definition of the Mach mechanisms and their implementation.

7. Acknowledgements

In addition to anything the author may have done, the heroes of the RIG kernel development were Gene Ball and Ilya Gertner. Jerry Feldman was in large part responsible for the initial RIG design and the system's name. The Accent development team included George Robertson and Gene Ball as well as the author. Keith Lantz and Sam Harbison made notable contributions to the design. Mary Shaw contributed the name. Others contributed greatly to Accent's evolution: particularly Doug Philips, Jeff Eppinger, Robert Sansom, Robert Fitzgerald, David Golub, Mike Jones and Mary Thompson. Matchmaker could not have come into existence without the aid of Mary Thompson, Mike Jones, Rob MacLachlin and Keith Wright. Mach was the brainchild of many including Avie Tevanian, Mike Young and Bob Baron. Dario Giuse came up with the name.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-81-K-1539.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

References

[1] Department of Defense.
Preliminary Ada Reference Manual
1979.

[2] Almes, G. and G. Robertson.
An Extensible File System for Hydra.
In *Proc. 3rd International Conference on Software Engineering*. IEEE, May, 1978.

[3] Ball, J.E., J.A. Feldman, J.R. Low, R.F. Rashid, and P.D. Rovner.
RIG, Rochester's Intelligent Gateway: System overview.
IEEE Transactions on Software Engineering 2(4):321-328, December, 1976.

[4] Ball, J.E., E. Burke, I. Gertner, K.A. Lantz and R.F. Rashid.
Perspectives on Message-Based Distributed Computing.
In *Proc. 1979 Networking Symposium*, pages 46-51. IEEE, December, 1979.

[5] Bisiani, R., Alleva, F., Forin, A. and R. Lerner.
Agora: A Distributed System Architecture for Speech Recognition.
In *International Conference on Acoustics, Speech and Signal Processing*. IEEE, April, 1986.

[6] Bobrow, D.G., Burchfiel, J.D., Murphy, D.L. and Tomlinson, R.S.
TENEX, a paged time sharing system for the PDP-10.
Communications of the ACM 15(3):135-143, March, 1972.

[7] Cheriton, D.R. and W. Zwaenepoel.
The Distributed V Kernel and its Performance for Diskless Workstations.
In *Proc. 9th Symposium on Operating Systems Principles*, pages 128-139. ACM, October, 1983.

[8] Spice Project.
Proposal for a joint effort in personal scientific computing.
Technical Report , Computer Science Department, Carnegie-Mellon University, August, 1979.

[9] Fitzgerald, R. and R. F. Rashid.
The integration of Virtual Memory Management and Interprocess Communication in Accent.
ACM Transactions on Computer Systems 4(2):, May, 1986.

[10] Hornig, D.A.
Automatic Partitioning and Scheduling on a Network of Personal Computers.
PhD thesis, Department of Computer Science, Carnegie-Mellon University, November, 1984.

[11] Jones, A.K., R.J. Chansler, I.E. Durham, K. Schwans and S. Vegdahl.
StarOS, a Multiprocessor Operating System for the Support of Task Forces.
In *Proc. 7th Symposium on Operating Systems Principles*, pages 117-129. ACM, December, 1979.

[12] Jones, M.B., R.F. Rashid and M. Thompson.
MatchMaker: An Interprocess Specification Language.
In *ACM Conference on Principles of Programming Languages*. ACM, January, 1985.

[13] Lantz, K.A.
Uniform Interfaces for Distributed Systems.
PhD thesis, University of Rochester, May, 1980.

[14] Lantz, K.A., K.D. Gradischnig, J.A. Feldman and R.F. Rashid.
Rochester's Intelligent Gateway.
Computer 15(10):54-68, October, 1982.

[15] McKusick, M.K., W.N. Joy, S.L. Leach and R.S. Fabry.
A Fast File System for UNIX.
ACM Transactions on Computer Systems 2(3):181-197 , August, 1984.

[16] Rashid, R.F. and G. Robertson.
Accent: A Communication Oriented Network Operating System Kernel.
In *Proc. 8th Symposium on Operating Systems Principles*, pages 64-75. ACM, December, 1981.

[17] R.F. Rashid.
The Accent Kernel Interface Manual.
Technical Report , Department of Computer Science, Carnegie-Mellon University, January, 1983.

[18] Spector, A.Z. et al.
Support for Distributed Transactions in the TABS Prototype.
In *Proceedings of the Fourth Symposium on Reliability in Distributed Software and Database Systems*, pages 186-206. October, 1984.