Unix as an Application Program

David Golub, Randall Dean, Alessandro Forin, Richard Rashid

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

1. Abstract

Since March of 1989 we have had running at CMU a computing environment in which the functions of a traditional Unix system are cleanly divided into two parts: facilities which manage the hardware resources of a computer system (such as CPU, I/O and memory) and support for higher-level resource abstractions used in the building of application programs, e.g. files and sockets. This paper describes the implementation of Unix as a multithreaded application program running on the Mach kernel. The rationale, design, implementation history and performance of the system is presented.

2. Introduction

It is both possible and rational to think of Unix[1] not as an operating system kernel but as an application program -- a server or set of servers that can provide client programs with specific programming abstractions. For the last twelve months we have had running at CMU a computing environment in which the functions of a traditional Unix system are cleanly divided into two parts: facilities which manage the hardware resources of a computer system (such as CPU, I/O and memory) and support for higher-level resource abstractions used in the building of application programs, e.g. files and sockets. The hardware management duties of Unix -- virtual memory, scheduling and device management -- are provided in an operating system environment independent way by the Mach kernel [1]. The key, recognizable Unix facilities -- such as Berkeley files, sockets and ttys -- are literally provided by an application program running on top of Mach. Even to the sophisticated Unix systems programmer the result is a seamless environment in which Berkeley 4.3BSD, Ultrix or SunOS 3.x binaries (depending on the machine type used) continue to run. Performance is comparable in a number of respects to that of earlier Mach releases and commercial Unix implementations.

The notion that Unix can be implemented by an application program acting as a server is the logical culmination of a longstanding trend in OS design. Over the last decade system programmers have increasingly relied upon the client/server implementation model as they have expanded the functionality of computer environments. Network file services, name services and database services are just a few of the server-provided facilities which have been added to traditional operating systems. Fueling this trend has been the recognition of the considerable advantages to be gained by the client/server approach, such as modularity and network transparency.

Beyond the obvious advantages common to all client/server systems, treating Unix as an application program has a number of interesting implications:

- *tailorability* --

[1] Unix is a trademark of AT&T Bell Laboratories

Versions of Unix such as 4.3BSD, Posix and System V.4 can be treated simply as different applications which can be purchased separately and potentially run simultaneously or even side by side with other OS environments.

- *portability* --

  Nearly all of the code which constitutes Unix is independent of a machine's instruction set, architecture and configuration.

- *network accessibility* --

  A Unix Server need not reside on the same machine as a Unix client.

- *extensibility* --

  New versions of Unix can potentially be implemented or tested alongside existing versions.

- *real-time* --

  Traditional barriers to real-time support in Unix can be removed both because the kernel itself does not have to hold interrupt locks for long periods of time to accommodate Unix system services and because the Unix services themselves are preemptable.

Our work does not represent the first time that Unix or another operating system has been implemented as an application program. Other such implementations have, however, frequently started from a rather different notion of the relationship between the system kernel and the supported OS environment. The approach we have taken with Mach was not to implement a virtual machine (as in IBM's CP/67 [4]) or to layer the kernel on a simple message engine (as in AT&T's MERT [6]) or to use a global shared communication area (as in Taos [8]) or to load operating system environment specific emulation-assist code into the kernel (as in Chorus [2]). Rather, we have taken advantage of the fact that Mach provides for the manipulation of system resources through a small set of machine-independent abstractions and for the integration of memory management and communication functions. All functionality pertaining to the implementation of Unix services is performed by a multithreaded Mach task which takes advantage of the Mach IPC, scheduling and virtual memory services. Unix facilities are provided by an application program running on top of Mach and no Unix-specific functionality exists within the kernel.

This paper describes our implementation of 4.3BSD as an application program. In particular, it examines the Unix Server, the key features of Mach on which our implementation depends and some of the challenges we encountered in the development of the system. The functionality and performance of the resulting system is also presented.

3. The Organization of Unix as an application program

The implementation of Unix binary compatibility on Mach divides cleanly into two parts: a *Unix Server* which provides the system services and resources commonly associated with a 4.3 BSD Unix environment and a *Transparent System Call Emulation Library* which operates within the address space of a Unix application. The Mach kernel provides the key support facilities upon which both the Unix Server and the Transparent Emulation Library depend. In particular Mach provides for interprocess communication, memory object creation and data management, scheduling and trap redirection.

3.1. Key Mach Features

It is difficult to describe the Unix Server and Emulation Library without briefly describing the most important details of the Mach kernel on which they both depend. A more detailed description on Mach and its abstractions can be found in [1]. The key features of Mach used in the emulation of Unix services are:

- interprocess communication,
- memory object management,
- scheduling,
- system call redirection,
- device support, and
- multiprocessing support.

3.1.1. Interprocess communication

Mach defines an address space and protection domain to be a *task* and a CPU flow of control to be a *thread*. Mach provides interprocess communication between threads through constructs called *ports*. Ports are protected with a capability mechanism and only Mach tasks with appropriate send or receive capabilities can access a port. All services, resources and facilities within the Mach kernel and between Mach tasks are represented as ports. Mach tasks, threads and memory objects are, for example, all manipulated by sending messages to ports which represent them. As such, the Mach port facility can be thought of as an object reference mechanism.

3.1.2. Memory object management

The address space of a Mach task is represented as a collection of mappings from linear addresses to offsets within Mach *memory objects*. The primary role of the kernel in virtual memory management is to manage physical memory as a cache of the contents of memory objects. The kernel's representation for the backing storage of a memory object is a Mach port to which messages can be sent requesting or transmitting memory object data. Memory object backing store can thus be implemented by user-state programs such as the Unix Server. The Unix Server makes use of this facility to act as an "external inode pager".

3.1.3. Scheduling

Unix application processes are implemented by the Unix Server as Mach tasks with a single thread of control. These threads are scheduled by the Mach kernel using a multi-level feedback queue scheduler with 32 priority levels. Because all Unix application tasks are created by the Unix Server, the Unix Server has direct access to their task and thread ports. This allows it to directly manipulate the priority and schedulability of these tasks to approximate traditional Unix scheduling and priority management.

3.1.4. System call redirection

The Mach kernel allows a designated set of system calls or traps to be handled by code running in user mode within the calling task. The set of emulated system calls needs to be set up only once; it is inherited by child tasks on fork operations.

3.1.5. Device Support

The Mach kernel provides all low-level device support. Each device is represented as a port to which messages can be sent to transfer data or control the device. Data is transferred through read and write operations; the request and reply messages are exported separately, allowing both synchronous and asynchronous styles of I/O. The external memory object protocol allows a user to map the frame buffer for a graphics device directly into its address space.

3.1.6. User Multiprocessing

A user-level multithreading package, the C Thread library [3], eases the use of multiple threads within an address space. It exports mutual exclusion *mutex* locks and condition variables for synchronization via *condition_wait* and *condition_signal* operations.

3.2. Unix Server

The bulk of Unix services are provided by the Unix Server. It is implemented as a Mach task with multiple threads of control managed by the Mach C Threads package. Internal synchronization and process-switching within the Unix Server (e.g., sleep, wakeup, spl) are implemented by using the C Threads package’s mutex, condition_wait and condition_signal functions. A typical system configuration will have dozens of C Threads allocated within the Unix Server. Most threads belong to a common pool which handle incoming requests from user processes. Several threads are dedicated to routines that, in a BSD kernel, would be driven by hardware interrupts (device IO completion, timeout, network code). All communication with hardware devices is done through Mach’s IPC (Interprocess Communication) facility. Figure 3-1 shows the organization of the Unix Server and its relationship to the Mach kernel.

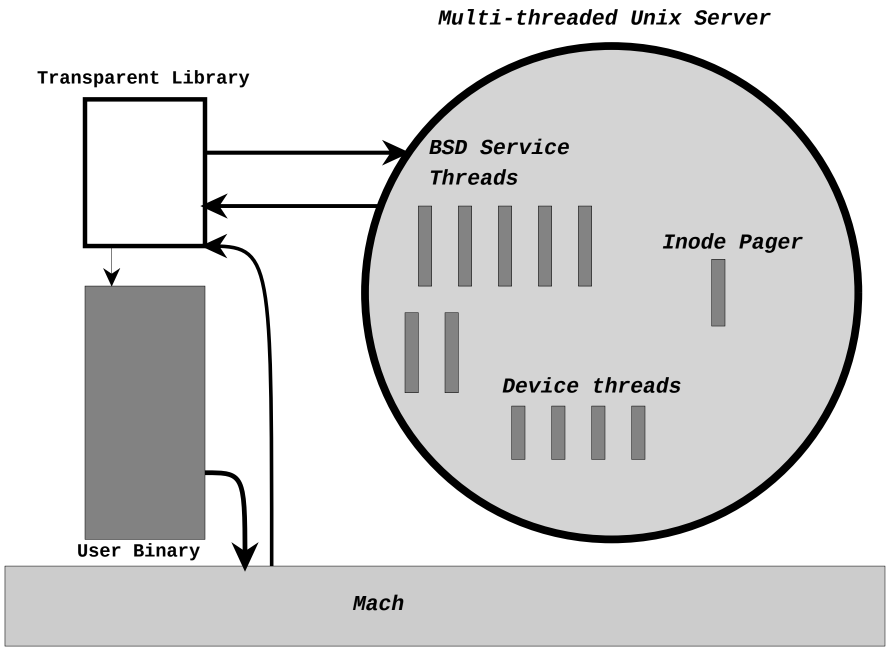

Figure 3-1: Organization of Unix services

The primary tool for communication between the Unix Server and a Unix application program is Mach IPC. Most requests for service arrive in the form of Mach messages requesting that a Unix operation or service be performed. For each incoming message a C Thread is dispatched from the pool to handle that operation. That thread then determines which Unix process requires service, what operation is to be performed and finally parses the message to obtain the arguments for that operation. Many, but not all, messages to the Unix Server correspond directly to system calls normally present in 4.3BSD.

The main departure from this style of interaction between the Unix Server and a Unix application can be found in the handling of 4.3BSD file access. Access to Unix files can be provided either through a pure message passing interface or through the Mach memory object facility. The decision of which interface to use can be made either by the Emulation Library or the Unix Server. The primary reason to choose a pure message passing interface would be performance in a network environment where a message passing interface corresponds more precisely to the natural implementation technology of a network. In a tightly coupled multiprocessor or a uniprocessor a memory object interface is a more efficient way to transfer large amounts of data.

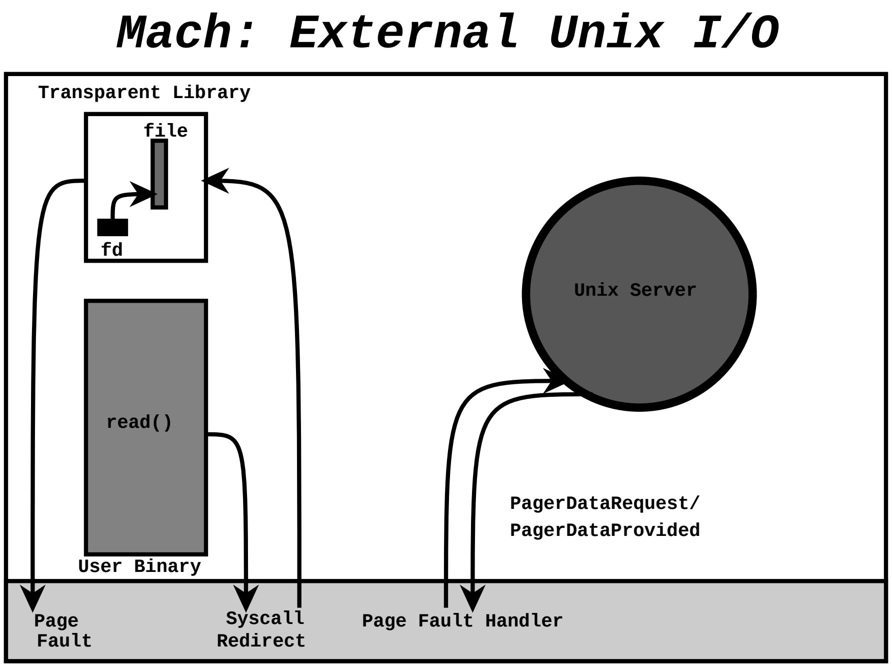

Figure 3-2: Implementation of Unix files as memory objects

In the case of a memory object implementation of file access, the Unix Server acts as the memory object manager (or "inode pager") for 4.3BSD files. When a file is opened by a Unix application its data is mapped directly into the portion of the Unix application address space occupied by the Transparent Emulation Library. That library then directly provides read, write, lseek, etc. system call access to the file's data. In order to ensure Unix file sharing semantics, the Transparent Library must hand-shake with the Unix Server through messages whenever conflicts with other applications could arise. See figure 3-2.

3.3. Transparent Emulator Library

The Transparent System Call Emulation Library is loaded into the address space of the first user process (/etc/init). Mach allows memory to be inherited either read/write or copy-on-write from parent to child task. The Unix Server takes advantage of this function to allow the Transparent Library to be inherited by each child process from its parent on a fork operation. Execve and similar operations which reload an address space with a new application program are careful to preserve the Transparent Library portion of the address space. One advantage of this technique is that it allows multiple Transparent Libraries with perhaps different behavior (such as the support of somewhat different Unix variants) to co-exist with the same Unix Server.

The Transparent Emulation Library contains the equivalent of the Unix system call handler and the glue routines necessary to transform system calls into remote procedure calls to the Unix Server. It intercepts system calls using the Mach system call redirection facility and transforms them into remote procedure calls to the server process. Most of the machine dependencies in the BSD code are handled by the Library. These include manipulating the application’s stack for signal handling and forking a new process.

In some cases Unix requests can be handled exclusively by the Transparent Library. For example, a Unix file which has been mapped into the Transparent Library memory region as part of an open call can be directly read by the Transparent Library without requiring the intervention of the Unix Server. Likewise some Unix signal management (e.g., sigvec), memory management (e.g., obreak) and state management (e.g., getpid) can be performed solely by the Emulation Library.

It is important to note that Transparent Emulation Library code is always executed with its own stack and does not use the stack of the application making the system call. This allows the Emulation Library to be compatible even with applications which may be doing their own stack management or providing their own lightweight process mechanisms.

It is also worth noting that the Transparent Emulation Library is vulnerable to tampering by an application program. It resides in a portion of the user application’s address space and it is within the capabilities of a user program to either read/write or remove that region of memory. As a result, care has been taken to ensure that the correct functioning of the Unix Server cannot be affected by malicious or unintentional tampering with the Emulation Library. It is also important that information managed by the Transparent Library not be more security sensitive than information otherwise available to the user. In this regard the Transparent Library must operate under restrictions similar to that of the standard C runtime library.

4. Implementation History

We started the implementation of the Unix Server in January of 1989. The single-server Unix system ran multi-user by March. Much of the speed with which this was done can be attributed to the fact that we took as much advantage as possible of earlier Mach implementations. Although earlier versions of Mach -- in particular Mach Release 2.5 -- provided Unix compatibility in kernel state, they did so by redefining those operations in terms of Mach primitives. As a result, much of the code providing Unix services was already structured in a way which allowed its use outside of the Mach kernel.

The first versions of the Emulation Library and Unix Server converted every Unix system call to a message. The Mach memory object facility was only used to implement the text and data segments of application address spaces. The result was a system which performed better than we had anticipated but still needed to be tuned considerably. For example, a simple compilation test which would normally take 28.5 seconds to execute under Mach Release 2.5 on an 8MB Sun 3/60 ran, instead, in about 38 seconds.

Some of the problems of the Unix Server were in evidence in specific tests of system call and read/write I/O performance. The cost of a getpid call was about 1.4 milliseconds on a Sun 3/60 and an 8192 byte read operation was over 5 milliseconds. These compared with 118 microseconds for getpid and 2 milliseconds for a read on Mach 2.5.

We examined in detail the behavior of the system running a variety of benchmarks and determined four areas in which substantive improvements would have the greatest effect:

1. the Mach IPC facility,
2. the Mach C Thread library,
3. management of signals and other forms of shared Library/Server state,
4. management of Unix file data (read/write/lseek in particular).

4.1. Mach IPC performance enhancements

At the time we began our work, the performance of the Mach IPC facility was substantially better than other Unix communication facilities [1] but was not tuned to the kind of intense use required by the Unix Server. In March of 1989 the minimum round trip remote procedure call times (RPC) for Mach on the MicroVAX III was measured as 800 microseconds. On the Sun 3/60 minimum RPC times were 1100 microseconds. For a compilation test requiring approximately 25-30 seconds to compete and utilizing 2600 system calls, this amounted to approximately 10% of elapsed time.

There were two ways to attack this problem and we chose to do both. We sped up the IPC facility by approximately 30% through the use of hand-off scheduling. By the summer of 1989 our MicroVAX III minimum RPC costs were 450 microseconds and our Sun 3/60 costs were 800 microseconds. In addition, we reduced the total number of messages to somewhat less half of all system calls (using techniques described below). The result of these enhancements was a considerable reduction in the total time devoted to minimum-path IPC costs.

4.2. C Thread library enhancements

We found two problems with the Unix Server's use of the C Thread library. First, the total number of C threads needed by the Unix Server was potentially much larger than the number of kernel threads it was reasonable to dedicate to it. Secondly, the cost of synchronization using the then existing version of the C thread mutex package was too high. Both problems were caused by a one-to-one binding of C threads to Mach kernel threads in that version of the C thread package.

We addressed these concerns by modifying the C thread package to detach the implementation of C threads from kernel threads. The package was changed to allow a specified pool of kernel threads to animate a much larger pool of C threads. Mutex, condition_wait and condition_signal operations were in turn modified to hand off execution from one C thread to another directly rather than attempt to use kernel scheduling facilities. The number of kernel threads used by the Unix Server could be set to allow any specified level of kernel managed multiprocessing. In addition, specific C threads were allowed to be bound to kernel threads where necessary, for example to act as device management threads.

4.3. Management of signals and shared state

Another problem area which became evident immediately upon instrumentation of the system was the frequent calls by Unix applications (especially the Unix shells) which are intended merely to set or read state information shared with the operating system. These calls include operations to modify signal handling state, verify the identity of the current process and modify the current address space by allocating new memory.

After some experimentation we determined that these operations could either be cached in a memory area accessible to both the Emulation Library and to the Unix Server or they could be performed directly by the Emulation Library using the underlying Mach kernel mechanisms. Thus, along with each Unix fork operation, the Unix Server allocates two memory objects which it maps into the newly created process. The first of these is a read-only object which contains information about the process which the Server is willing to communicate to it. The second is a read-write object accessible to both the Library and the Server. This object is used to allow the Emulation Library to update state information which is not immediately needed by the Server.

Although attractive on a tightly coupled multiprocessor or a uniprocessor, this use of shared state is not necessary appropriate to other architectures. We therefore made it optional with the version of the client Emulation Library. A full message passing interface to the server was preserved.

4.4. Management of Unix file data

One of the major costs in any Unix system is the cost of data movement to and from the disk subsystem. For example, we found that over 75% of all operations in a measured compilation suite were either open, close, read, lseek or write. Within this class, read, write and lseek dominated all file system operations.

The costs of data movement in turn dominate read and write system call times. On a Sun 3/60 the basic cost of an 8192 byte data copy is 1.2 milliseconds. Because Unix read and write system calls specify their target byte addresses it is not possible, in general, to use virtual memory page management to move data directly into a client address space. The simplest way to move 8K of data from the Unix Server to a client application is by Mach messages which actually contain 8192 bytes of copied data. Unfortunately this approach can result in as many as four data copies per call and minimum read or write times of greater than 5 milliseconds.

We experimented with a variety of techniques for reducing these data copy costs. Ultimately the best strategy was to effectively eliminate the cost of data copy altogether. This was accomplished by having the Unix Server map files directly into the address space managed by the Transparent Emulation Library. This allowed read, write and lseek operations to be performed completely by the Library rather than the Server. State information pertaining to the Library's access to a file is shared with the Unix Server through a shared memory object page (see the previous section).

The role of the Unix Server in this approach is that of a memory object manager for inodes and a synchronization manager for files which are shared by multiple simultaneously executing Unix processes.

This eliminates the costs of message passing for many operations as well as the cost of extra data copies. Only one copy is ever made on a read or write call: the necessary copy from the application specified byte address to the actual page of the file in memory. Another advantage of this approach is greater potential parallelism. Because the operation is confined to the client application there is typically no lock contention or serialization.

The major drawback of this approach is increased costs for open and close operations on files. These operations can require memory mapping and deallocation costs which, in practice, are reasonably expensive. Our investigations have shown that open/close costs are important, but not nearly as important as the cost of read/write/lseek operations.

The results of our modifications to the file system have been gratifying. Instead of taking over 5 milliseconds for an 8192 byte lseek/read the costs for an lseek/read loop on the Sun 3/60 were reduced to less than 1.4 milliseconds. This compares favorably to the 2.12 milliseconds for the same test running under Mach Release 2.5.

5. Performance

We examined the performance of our multithreaded Unix Server and Emulation Library through the use of two system oriented benchmarks, one a simple file compilation benchmark and the other a more comprehensive file system test. We also timed specific system calls both individually and in frequently used combinations. In the following sections all tests were made running either Mach Release 2.5 or the Mach kernel with the Unix Server described in this paper. All tests were made on a Sun 3/60 with 8MB of memory and a Priam 300MB disk drive. The same disk with the same binaries and user environment was used in all tests.

5.1. A compilation test suite

We devised a relatively simple compilation test suite and used it to measure the performance of both systems. This compilation test consisted of a shell command file which ran nine compilations of small C source files. These files contained relatively few header file inclusions. Each compilation was separately timed using the "time" command.

The resulting test stressed process creation/termination, program load and startup, file open/close and read/write costs for small files. During the roughly 30 seconds required to complete the test it performed approximately 2600 Unix system calls, forked 57 processes, attempted to open 240 files and close 350 file descriptors, unlinked 100 files and called execve 160 times. Read, write and lseek operations accounted for a large fraction of all system calls. Roughly 750 lseek operations, 450 read and 230 write operations were performed. The table below shows the results of the benchmark. The runtime for SunOS 4.0 on the same machine is shown for purposes of comparison.

Compile Test Peformance	
Operating System	*Elapsed Time*
SunOS 4.0	49 seconds
Mach Release 2.5	28.5 seconds
Mach w/Unix Server	28.4 seconds

5.2. File system benchmarks

In order to get a more complete evaluation of the system's file system performance we took advantage of a benchmark originally developed by M. Satyanarayanan for his peformance evaluation of the Andrew File System [5]. Specifically we used a version of the Andrew Benchmark modified by John Ousterhout [7]. This benchmark stresses directory and file creation, file copy, file search (using "find") and compilation activity. A complete examination of this benchmark can be found in the cited papers.

Modified Andrew Benchmark						
Operating System	*Directory Creation*	*File Copy*	*Recursive file stats*	*Find*	*Compile*	*Elapsed*
Mach Release 2.5	4 sec	20 sec	13 sec	26 sec	336 sec	399 sec
Mach w/Unix Server	1 sec	20 sec	24 sec	34 sec	332 sec	411 sec

Probably the greatest differences revealed by this test were in the the costs of directory scanning and the "stat" operation. Neither have been examined or tuned in the current Unix server. The overall difference in performance was about three percent.

5.3. Basic File System Costs

In addition to our examination of various application benchmarks we looked at the cost of specific operations. In particular, we examined the cost of creating, writing and deleting files of various lengths, the throughput for both cached and uncached file read operations and for file write operations. The table below summarizes these results.

File System Throughput		
Operation	*Mach Release 2.5*	*Mach w/Unix Server*
create write (0 bytes) Delete	84.4ms	83.4ms
create write (10K bytes) delete	154.0ms	152.0ms
create write (100K bytes) delete	634.1ms	596.0ms
write (1M bytes)	0.26 megabytes/sec	0.27 megabytes/sec
read (cached)	3.98 megabytes/sec	3.93 megabytes/sec
read (uncached)	0.41 megabytes/sec	0.34 megabytes/sec

Overall, we found the behavior of the Mach w/Unix Server system to be very similar to that of our Mach 2.5 Release for these operations. In particular, the costs of cached read operations and of write operations were nearly identical.

Measured separately, the costs for open, close, lseek and read operations expose the differences in implementation techniques and the tradeoffs between an in-kernel and out-of-kernel Unix environment. Repeated read and write operations of various lengths are considerably faster in our out-of-kernel system. The cost of an "open" call, however, has increased due to the required message handshakes and to the cost of memory mapping the referenced inode.

File System Operation Costs		
Operation	*Mach Release 2.5*	*Mach w/Unix Server*
lseek	160us	170us
lseek + read (1 byte)	0.44ms	0.35ms
lseek + read (8K bytes)	2.12ms	1.38ms
open "foo" + read (8K) + close	3ms	5.5ms
open "/usr/rfr/tests/foo" + read (8K) + close	5.2ms	6.68ms
failed open of "xxx"	1.66ms	1.42ms

5.4. Process Management Costs

Our raw measurements of process creation and termination demonstrate a problem we have not yet resolved with the use of the Transparent Emulation Library. As can be seen from the table below, fork and exec costs range from 2 to 3 times greater for the Mach w/Unix Server than for Mach Release 2.5. The cost of the fork operation itself is the primary culprit.

Process Management Costs			
Operation	*Mach Release 2.5*	*Mach w/Unix Server*	*SunOS 4.0*
getpid	118us	102us	75us
fork + exit	16ms	48ms	27.6ms
fork + wait + exec + exit	32ms	74ms	106ms

The explanation for this increase in fork cost can be found in a large number of additional page faults which result from the use the Transparent Library. At fork, a copy-on-write copy of the parent is created. In a normal in-kernel Unix implementation, the parent modifies a single stack page (resulting in a copy-on-write fault) and then does a Unix "wait" for the child to exit. In our out-of-kernel Unix environment both the parent and child processes modify the top-of-stack page of their Transparent Library as well as their own process top-of-stack page. In addition references are made to various code and data pages in the Transparent Library. As a result, twenty faults are taken in the out-of-kernel version. Of these faults, five are copy-on-write faults and three are fill zero faults. This contrasts with only five faults altogether for the in-kernel Unix of which two are copy-on-write faults. The high cost of program execution in the early SunOS 4.0 system (included for comparison) was similarly the result of its use of shared libraries.

6. Status

All 4.3BSD binaries, Ultrix binaries or Sun 3.x binaries which ran on earlier versions of Mach continue to run. The switch from older releases of Mach (or Ultrix on the DECStation) can be accomplished simply by installing a new kernel, the Unix Server and Emulator Library on an existing disk. The system includes support for X11R4, full Berkeley networking, the Andrew Nationwide File System and CMU's internal remote file system.

At the time of this writing (April, 1990) the pure Mach kernel with Unix as an application program was running on multiprocessor and uniprocessor Vax systems, the DECStation 3100, the Sun 3 and i386 PC-clones. We are working with the Open Software Foundation on a version for the Encore Multimax. We expect to support the Macintosh and other machines in the future.

The multithreaded Unix Server described in this paper is but one of two ongoing implementations of Unix functionality at CMU. A second group [9] has been engaged in an implementation of Unix based on multiple Mach server tasks each of which performs specific functions such as file access, authentication, etc. The advantage of this approach over a single server Unix implementation is flexibility (since each server can provide services for various operating system environments) and security (since each can be individually secured and verified).

Work using Mach to implement non-Unix environments is also in progress. A virtual Macintosh OS environment is already running on top of Mach Release 2.5 at CMU and is scheduled to be ported to the pure kernel. We are also providing support for DOS applications on the i386.

7. Conclusions

We believe we have demonstrated the practicality of implementing Unix as an application program. Our performance measures demonstrate differences in the underlying cost profiles for our in-kernel and out-of-kernel Unix systems but do not indicate serious problems. We continue to make improvements but we have already achieved near parity with Mach Release 2.5 and can outperform some commercial Unix implementations.

8. Author Biographies

8.1. David Golub

David Golub is a Senior Research Programmer and has worked on the Mach project since December 1985. He received the B. A. in mathematics from Brown University in 1975. His research interests are in distributed computing.

8.2. Randall Dean

Randall Dean is a Research Systems Programmer working with the Mach Project since May, 1989. His primary interests are rock climbing and volleyball.

8.3. Alessandro Forin

Dr. Forin received the B.S. degree in Electrical Engineering in 1982 and the Ph.D. degree in Computer Science in 1987, both from the University of Padova, Padova, Italy. He is currently a Research Computer Scientist at the Carnegie Mellon University. Dr. Forin is member of the Association for Computing Machinery, SIGPLAN and SIGOPS.

8.4. Richard Rashid

Professor Rashid is Director of the Mach Project and has been on the faculty of Carnegie-Mellon University since September, 1979. He received his M.S. (1977) and Ph.D. (1980) degrees in Computer Science from the University of Rochester. He had previously graduated with Honors in Mathematics from Stanford University (1974).

References

[1] Accetta, M.J., Baron, R.V., Bolosky, W., Golub, D.B., Rashid, R.F., Tevanian, A., and Young, M.W.
Mach: A New Kernel Foundation for UNIX Development.
In *Proceedings of Summer Usenix*. July, 1986.

[2] Armand F., Gien M., Guillemont, M. and Leonard, P.
Towards a Distributed UNIX System - The CHORUS Approach.
In *Proceedings of the European UNIX Systems User Group Conference*. September, 1986.

[3] Cooper, E. and Draves, R.
C Threads.
Technical Report CMU-CS-88-154, Computer Science Department, Carnegie Mellon University, June, 1988.

[4] Parmelee, R. P, T. I. Peterson, C. C. Tillman and D. J. Hatfield.
Virtual Storage and Virtual Machine Concepts.
IBM Systems Journal 11(2):99-130, 1972.

[5] Howard, J.H., Kazar, M.L., Menees, S.G., Nichols, D.A., Satyanarayanan, M., Sidebotham, R.N., West, M.J.
Scale and Performance in a Distributed File System.
ACM Transactions on Computer Systems 6(1), February, 1988.

[6] Lycklama, H. and Bayer, D. L.
The MERT Operating System.
Bell System Technical Journal , July, 1978.

[7] Ousterhout, J.
Why Aren't Operating Systems Getting Faster as Fast as Hardware?
In *Proceedings of Summer Usenix*. June, 1990.

[8] Thacker, C. P. and Stewart, L. C. and Satterthwaite, Jr., E. H.
Firefly: A Multiprocessor Workstation.
IEEE Transactions on Computers 37(8):909-920, August, 1988.

[9] Rashid, R., Baron, R., Forin A., Golub, D., Jones, M., Julin, D., Orr, D., Sanzi, R.
Mach: A Foundation for Open Systems; a Position Paper.
In *Proceedings of the 2nd Workshop on Workstation Operating Systems*. IEEE, September, 1989.

Table of Contents

1. Abstract 0
2. Introduction 0
3. The Organization of Unix as an application program 1
    - 3.1. Key Mach Features 2
        - 3.1.1. Interprocess communication 2
        - 3.1.2. Memory object management 2
        - 3.1.3. Scheduling 2
        - 3.1.4. System call redirection 2
        - 3.1.5. Device Support 3
        - 3.1.6. User Multiprocessing 3
    - 3.2. Unix Server 3
    - 3.3. Transparent Emulator Library 5
4. Implementation History 5
    - 4.1. Mach IPC performance enhancements 6
    - 4.2. C Thread library enhancements 6
    - 4.3. Management of signals and shared state 7
    - 4.4. Management of Unix file data 7
5. Performance 8
    - 5.1. A compilation test suite 8
    - 5.2. File system benchmarks 9
    - 5.3. Basic File System Costs 9
    - 5.4. Process Management Costs 11
6. Status 11
7. Conclusions 12
8. Author Biographies 12
    - 8.1. David Golub 12
    - 8.2. Randall Dean 12
    - 8.3. Alessandro Forin 12
    - 8.4. Richard Rashid 12

List of Figures

Figure 3-1:	**Organization of Unix services**	**3**
Figure 3-2:	**Implementation of Unix files as memory objects**	**4**