Data Movement in Kernelized Systems

Randall W. Dean
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, Pennsylvania 15213

(412) 268-7654

rwd@cs.cmu.edu
Fax: (412) 681-5739

Francois Armand
CHORUS Systemes
6 Avenue G. Eiffel
78182 ST-QUENTIN-EN-Y
CEDEX-FRANCE

+33 (1) 30-64-82-00

francois@chorus.fr

This paper considers how two kernelized systems, Mach 3.0 with the BSD4.3 Single Server and CHORUS/MiX V.4, move data to and from files under a variety of circumstances. We give an overview of the kernel abstractions and system servers and describe in detail the *read()* and *write()* paths of these two systems. We then break down their *read()* and *write()* performance and compare them to two monolithic systems, Mach 2.6 MSD(BSD4.3) and System V R4.0. We then describe the compromises each of the two kernelized systems made in order to achieve a goal of performance comparable to the monolithic systems. We conclude with a description of what techniques each system uses that could benefit both each other and traditional monolithic systems.

1. Introduction

A recent trend in operating system research has been towards small kernels exporting a small set of abstractions that are used by a server or set of servers to implement the services provided by traditional operating systems [7, 9, 14, 18]. This approach to building operating system services gives OS developers a sophisticated environment for building and testing new services and meeting the needs of newer hardware platforms. A kernelized system also allows a developer to mix and match components as needed, minimizing or eliminating unneeded capabilities that are a permanent part of traditional monolithic systems. Kernelized systems have also demonstrated that they are a sound basis on which one can build distributed operating systems [2, 5] and/or provide features such as real-time [15, 19] or Object-Oriented environments [16].

For kernelized systems to gain acceptance, they must be binary compatible with and perform comparably to monolithic systems. Many papers have described efforts and experiences towards achieving these goals [6, 13]. Some authors have asserted that some services, such as, the file system, do not belong outside of the kernel [20]. One of the major fears appears to be the need for costly context switching. To meet the performance goal, a kernelized system must either make context switching faster than in systems where it is not in the critical path, or somehow avoid them entirely where possible. Data movement must also be carefully designed in these systems to avoid extra copying of data.

This paper considers how two systems, Mach 3.0 with the BSD4.3 Single Server and CHORUS/MiX V.4, achieve the performance goals by controlling data movement. We are particularly concerned with how fast these two systems can move data to and from files under a variety of circumstances. First we give an overview of the kernel abstractions and system servers and describe in detail the *read()* and *write()* paths of these two systems. We then break down their *read()* and *write()* performance and compare them to two monolithic systems, Mach 2.6 MSD(BSD4.3) and System V R4.0. Next we describe the compromises each of the two kernelized systems has made to achieve performance comparable to the monolithic systems. We conclude with a description of what techniques each system uses that could benefit both each other and traditional monolithic systems.

2. Microkernel Abstractions

The Mach 3.0 Microkernel and the CHORUS Nucleus supply a similar set of abstractions for building systems servers [10, 17]. Unfortunately, for historical reasons, the two systems often use different names to describe the same thing. The remainder of this section describes the abstractions of Mach 3.0 and CHORUS relevant for understanding the rest of the paper using either the common name or both when necessary.

- A **Task** [4] or **Actor** [8] is an execution environment and the basic unit of resource allocation. Both include virtual memory and threads. The Mach task also includes port rights. An actor includes ports as communication resources. A task or actor can either be in kernel or user space.
- **Threads** are the basic unit of execution. A task or actor can have multiple simultaneous threads of execution. Threads may communicate via **Ports**.
- Both systems are built around Interprocess Communication or IPC.
    - Mach ports are protected communication channels [12] managed by the kernel with separate port rights residing in each task. A thread uses the local name and right for a port residing in its task to send typed data to the task having the unique receive right for that port. **Port Sets** allow a task to clump a group of ports together for the purpose of receiving from multiple ports with only one thread.
    - CHORUS IPC uses a single global name space with each port named by a Unique Identifier (UI) to send untyped data to ports. CHORUS ports, which may belong to different actors, may be grouped into **port groups**. CHORUS IPC offers the capability of broadcasting a message to all ports in a group. Actors running in supervisor space may define **message handlers** to receive messages. Instead of explicitly creating threads to receive the messages, an actor may attach a handler, a routine in its address space, to the port on which it waits for messages. When a message is delivered to the port, the handler is executed within the context of the actor using a kernel provided thread. This mechanism avoids extra copy of data and context switches when both the client and the server run on the same site. The connection of message handlers is transparent to the client of a server.
- Address spaces
    - In Mach, the address space of a task is made up of **VM Objects**. Objects often map secondary storage managed by an **External Pager** [21]. An object, which is represented by a send right to a port, must be entered into the task's address space with **vm_map** and can subsequently accessed through normal memory accesses.
    - The address space of a CHORUS actor is made up of **Regions**. Regions often map secondary storage called **segments** managed by **Mappers**. A segment is represented by a capability, a port UI and a key. CHORUS also allows segments to be read or written directly without mapping them using *sgRead()* and *sgWrite()* Nucleus calls.
- Device Access
    - Mach gives direct access to disks and other devices through *device_read()* and *device_write()*. The *device_read()* call, which, like most Mach calls, is an RPC to the kernel, returns the data in out of line memory directly from the disk driver without any unnecessary copies.
    - By contrast, the CHORUS nucleus doesn't know about any device except the clock. Instead, it allows actors to dynamically connect handlers to interrupts and traps. Device drivers are implemented this way by actors running in the supervisor address space.
- To allow for binary compatibility, Mach and CHORUS each have a mechanism for handling

Unix™ system call traps called **Trap Emulation** or **Trap Handlers**. The Mach kernel enables a task to redirect any trap number back into the user task making the trap. CHORUS has a mechanism for allowing traps to be handled by any actor runing in supervisor address space which has attached a handler to these traps.

3. System Servers

Both Mach 3.0 with the BSD4.3 Single Server and CHORUS/MiX V.4 consist of a small kernel or nucleus described in the previous section, and a server or set of servers running in user mode or possibly supervisor mode supporting Unix semantics. This section gives on overview of both of these systems server components.

3.1. Mach 3.0 with the BSD4.3 Single Server

The BSD4.3 Single Server is a single user application which exports Unix semantics to other user applications. To communicate with this server, an **Emulation Library** is loaded into the address space of all clients beginning with /etc/init and inherited subsequently by all of its children. A typical system call traps into the kernel and is redirected by the **Trap Emulation** mechanism back out into the Emulation Library. The Emulation Library sends a message to the BSD4.3 Single Server which then executes the actual Unix call and returns back through the Emulation Library to the user application. The remainder of this section describes the Emulation Library and BSD4.3 Single Server and how they support Unix files.

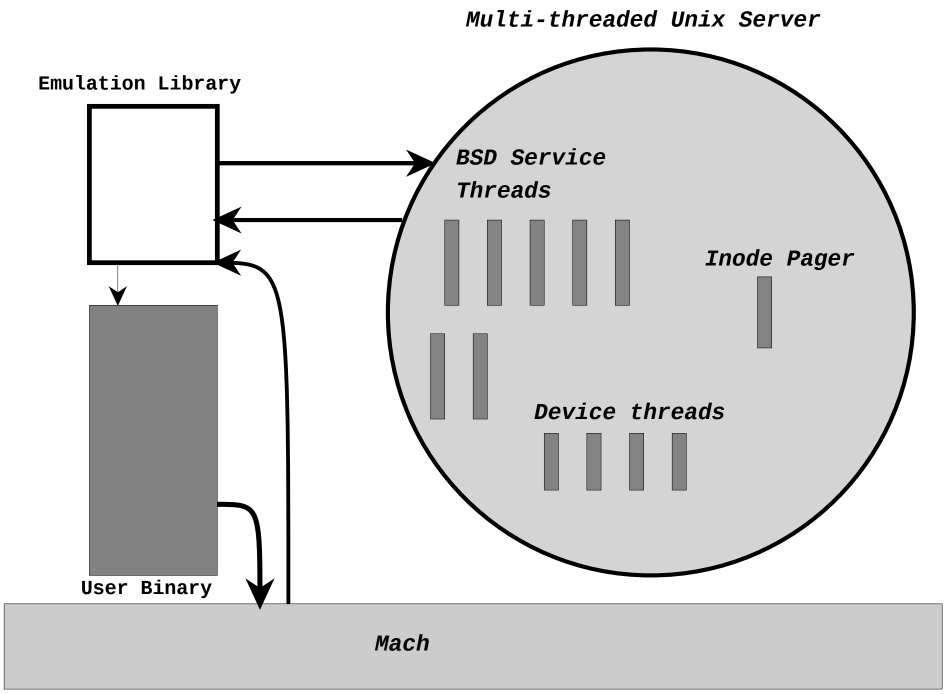

Figure 3-1: A System Call in Mach 3.0 with the BSD4.3 Single Server

3.1.1. BSD4.3 Single Server and Emulation Library Structure

The BSD4.3 Single Server is a single task made up of numerous CThreads [11]. Most of the CThreads are part of a pool which responds to messages from the Emulation Libraries of Unix processes. When they receive a message from a client Emulation Library, they internally take on the identity of that client and execute the appropriate Unix system call. Upon completion, they return the result and reenter the pool of waiting threads. Each of the remaining CThreads provides a particular service. There is the **Device Reply Thread**, which handles all responses from the server to the kernel for device interaction, the **Softclock Thread**, which implements internal timeouts and callbacks, the **Net Input Thread**, which handles network device interactions with the kernel, and the **Inode Pager Thread**, which implements an external pager backing Unix file objects.

The Emulation Library is actually a self-contained module that is loaded into the address space of BSD4.3 Single Server clients. It is entered from the kernel's Trap Emulation code in response to a system call. The Emulation Library then either handles the system call locally or forwards the request as a message to the server to handle it. To allow the Emulation Library to handle some system calls without contacting the BSD4.3 Single Server, the Emulation Library and the BSD4.3 Single Server share two pages of memory. While the Emulation Library can read both pages, it can only write one of them. The read-only page contains information that a client can already get through querying system calls such as *getpid()*, *getuid()*, and *getrlimit()*. The writeable page contains data that the client can already set through system calls such as *sigblock()* and *setsigmask()*. The writeable page also contains an array of special files descriptors used by the mapped files system.

3.1.2. Mapped Files

The BSD4.3 Single Server is capable of mapping files backed by the Inode Pager directly into the address space of clients for direct access by the Emulation Library. These mapped regions are actually windows into the Unix files that can be moved by a request from the Emulation Library. There is exactly one window of 64K bytes in size for each open file. For each mapped region, there is a corresponding file descriptor in the writeable page of shared memory. This file descriptor contains information on the current mapping window and a copy of the real file descriptor in the BSD4.3 Single Server.

To allow for Unix file semantics which permit multiple readers and writers of files and the sharing of the current file pointer, there is a **Token** scheme. The Token protects the mapping window information, the file pointer and the file size. The Token can be in three states. These are *active*, *invalid* and, to limit the number of messages sent to the BSD4.3 Single Server, *passive*. The transitions between these states are covered in section 4.

3.2. CHORUS/MiX V.4

3.2.1. The CHORUS/MiX V.4 subsystem

MiX V.4 is a CHORUS subsystem providing a Unix interface that is compatible with Unix SVR4.0. It is both BCS and ABI compliant on AT/386 and 88K platforms. It has been designed to extend Unix services to distribution such as access to remote files and remote execution.

MiX V.4 is composed of a set of cooperating servers running in independent actors on top of the CHORUS Nucleus and communicating only by means of the CHORUS IPC. The following servers are the most important:

- The **Process Manager** (PM) provides the Unix interface to processes. It implements services for process management such as the creation and destruction of processes and the sending of signals. It manages the system context of each process that runs on its site. When the PM is not able to serve a Unix system call by itself, it calls other servers, as appropriate, using CHORUS IPC.

- The **Object Manager** (OM), also named the **File Manager** (FM), performs file management services. It manages various file system types such as S5, UFS, and NFS. It also acts as a CHORUS Mapper for "mappable" Unix files and as the Default Mapper for swap space management. Disk drivers are generally part of the Object Manager.
- The **Streams Manager** (StM) manages all stream devices such as pipes, network access, ttys, and named pipes when they are opened. It cooperates with the Object Manager which provides the secondary storage for the files' meta-data.

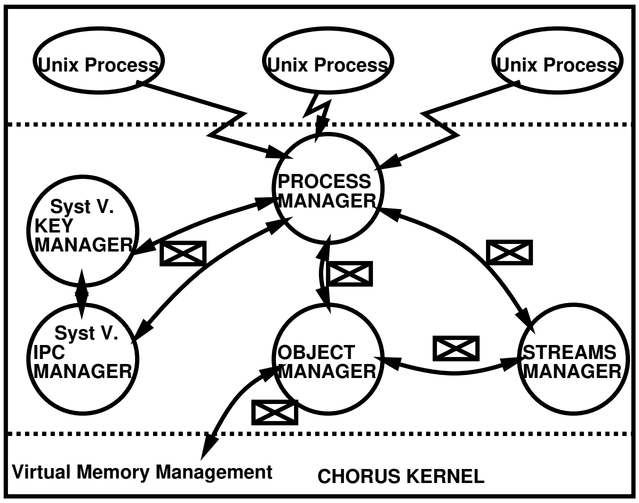

Figure 3-2: CHORUS/MiX V.4 subsystem

All MiX servers are fully independent and do not share any memory and thus can be transparently distributed on different sites. Although they may run in either user space or supervisor space they are usually loaded in the supervisor address space to avoid numerous and expensive context switches each time a new server is invoked. This paper discusses the case where all servers are loaded into supervisor space.

3.2.2. Regular File Management in CHORUS/MiX V.4

The management of regular files is split between three components in MiX V.4: the Object Manager, the Process Manager and the CHORUS Nucleus. Pages of regular files are read, written and cached using the Virtual Memory services, *sgRead()* and *sgWrite()*, without having to map them. This allows the support of mapped files and provides the benefit of the Virtual Memory mechanisms for caching files from local or remote file systems. The Virtual Memory cache replaces the buffer cache, moreover in the distributed case, file consistency are maintained by the same mechanisms that are used to provide distributed shared memory.

In MiX V.4, open files are named by a capability built from a port Unique Identifier and a 64-bit key only meaningful to the server. This capability is returned to the PM by the OM when the *open()* system call is made. All opens of the same file get the same capability. For each open system call, the PM manages an open file descriptor similar to the *file_t* structure of a native Unix, that handles flags and the current offset. In addition, the PM manages an **object descriptor** for each file in use on its site. This object descriptor handles the following information: size of the file, mandatory locks posted against the file (if any), last access and last modification time and, the capability of the file exported by the OM. The PM uses this information to convert *read()/write()* Unix system calls into *sgRead()/sgWrite()* CHORUS Nucleus calls.

3.2.3. File Size Management

Due to the separation between the Process Manager and the Object Manager, a benefit which allows them to potentially be distributed on different nodes, the file size is protected by a **Token**. Since the Token may have been granted to another site or recalled by the OM, the PM must check whether the file size information it holds is valid or not before using the file size. If the information is not valid, the PM must retrieve the information first.

4. *Read()* and *Write()* Path Analysis

This section takes a detailed look at the *read()* and *write()* paths of both Mach 3.0 with the BSD4.3 Single Server and CHORUS/MiX V.4.

4.1. Mach 3.0 with the BSD4.3 Single Server *Read()* Path

This section describes how the *read()* system call works in Mach 3.0 with the BSD4.3 Single Server. When a user makes a *read()* call the kernel redirects that call trap back into the Emulation Library. After making sure that the file is mapped, that it has a valid Token, and that the mapping is into the desired part of the file, the Emulation Library copies the data from the mapped region into the user's buffer and returns.

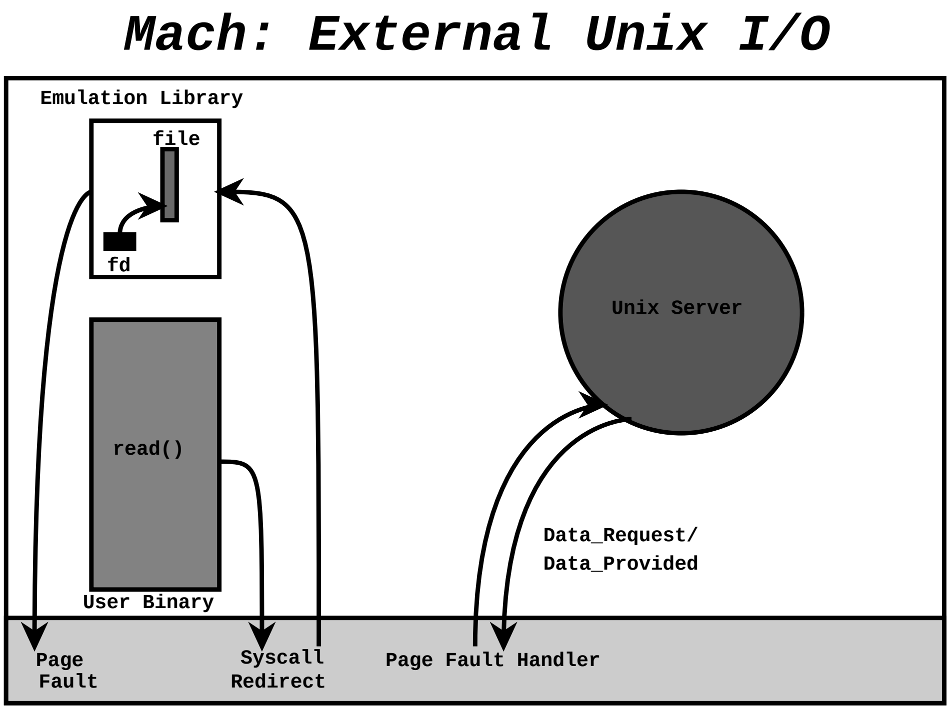

Figure 4-1: Mach 3.0 with the BSD4.3 Single Server Mapped File Read

4.1.1. *Read()* in the Emulation Library

To examine the file descriptor in the writeable shared page, the Emulation Library must first acquire the **share lock** which protects this region. A share lock is different from a simple spin lock in that the BSD4.3 Single Server does not trust the Emulation Library. The Emulation Library resides in user memory and can be over-written at any time by an incorrect or malicious user program. Therefore, the BSD4.3 Single Server must take measures to ensure it does not deadlock on this share lock. The share lock is implemented as a simple spin lock in the Emulation Library and as a test-and-set and block with timeout in the server. When the server blocks, it requests a callback from the Emulation Library when the Emulation Library releases the lock. If the callback is not received in a configurable amount of time, the server assumes the client is malicious or incorrect and kills the task.

Once the Emulation Library has acquired the share lock, it can examine the state of the Token. If the Token is *invalid*, the Emulation Library must send a message to the server to acquire the Token. If the Token is *passive*, it may be switched directly to *active* without contacting the server. Once the Emulation Library has the Token, its file pointer and mapping info are guaranteed to be correct. If the mapping information is not adequate for the current call, the Emulation Library can send a message to the server to change the mapping. With a valid mapping, the Emulation Library simply copies the data into the user's buffer, updates the file pointer, and releases the Token. If the BSD4.3 Single Server has indicated, by setting a bit in the file descriptor, that another task is waiting for the Token, the Emulation Library sends a message to the server to completely release the Token. If no call-back is needed, the Emulation Library moves the Token to *passive*.

4.1.2. *Read()* in the BSD4.3 Single Server

The BSD4.3 Single Server can become involved in a *read()* call in a number of ways. The Emulation Library can call it directly in order to request a Token, request a window remapping, or release a Token. The BSD4.3 Single Server can also be called by the Mach kernel when the Emulation Library faults on mapped data and the Inode Pager must be contacted to satisfy the fault. Each of these operations is described in detail below.

A Token request is straightforward to process. The server keeps track of who has the Token and examines the state of that process's file descriptor. If the holder of the Token has it *passive*, the server invalidates the Token and grants it to the requester. If the Token is *active*, the server leaves a call back request with the holder and waits for it to send a Token release call.

A request to change the mapping window is extremely simple when just reading from a file. All the server has to do is deallocate the existing window into the file and generate a new mapping which covers the range necessary for the *read()*. The handling of the remapping operation for *write()* is covered in detail in section 4.2.

When the Emulation Library first touches a page of data mapped into its address space, a page-fault occurs. The kernel receives this fault and resolves it in one of two ways. If the page is already present in the kernel VM system, but the task does not have a valid mapping, then the kernel enters the page mapping and returns from the fault. If the page is not present in the kernel, the kernel sends a *memory_object_data_request()* message to the External Pager backing this object. In the case of Unix file data, this is the Inode Pager in the BSD4.3 Single Server. The Inode Pager then reads the appropriate data off disk with *device_read()* and provides this page back to the kernel.

For historical reasons, the Inode Pager uses the buffer cache interface to read and write from and to disk.

This leads to unnecessary caching in the BSD4.3 Single Server. This also results in the Inode Pager using a less efficient and now obsolete interface *memory_object_data_provided()* instead of *memory_object_data_supply()* to return the data to the kernel. The former requires a copy within the kernel, whereas the latter uses a page stealing optimization that eliminates the copying. In the page stealing optimization, the VM system just removes the physical page from the server and places it directly in the Memory Object without copying it.

4.2. Mach 3.0 with the BSD4.3 Single Server *Write()* Path

The *write()* path is almost identical to the *read()* path. The differences lie in correctly managing the file size and writing out dirty pages in a timely fashion. As in the *read()* case, the Emulation Library sets up a valid mapping and copies the user's data into the window. A valid mapping may exist past the end of the file, so there is no need to handle file extension as a special case in the Emulation Library.

If the Emulation Library write faults on a page that is in the file, the Inode Pager returns the page from the disk file to be over written. In the case of *write()* faulting on a new page, such as filling a whole or extending the file, the Inode Pager returns *memory_object_data_unavailable()* which causes the kernel to supply a zero filled page to the client. If the *write()* extends the file, the Emulation Library updates the file size field in the local file descriptor.

Actual writes to disk and propagation of the file size occur when the Token is taken away from a writer, when the writer changes its mapping, or when the kernel needs free pages and starts sending the mapped file pages to the Inode Pager to be cleaned. In the first two cases, the mapped file code first updates the file size from the writer's file descriptor. The server then calls *memory_object_lock_request()* to force the kernel, which knows which pages have actually been written by the user, to request cleaning of the dirty pages. This generates messages from the kernel to the pager requesting that the dirty pages be written out. When the Inode Pager receives the dirty pages from the kernel, it writes them out to disk.

Disk block allocation is delayed until the data is actually written out to disk. It would be possible for the Inode Pager to allocate new blocks in a timely fashion since it gets a data request message when the Emulation Library page-faults on the new page to be written. The Inode Pager currently ignores the write fault attribute when returning the empty page to satisfy the page-fault. By not allocating at the time of the initial fault, the semantics for failure in the BSD4.3 Single Server are not the same as Mach 2.6 MSD(BSD4.3).

The difficulty occurs when there was a involuntary request to clean pages from the kernel driven by a memory shortage. In the previous cases where dirty pages and the file size were written out, the writing was initiated by a call from the Emulation Library. In this case the Emulation Library could be in the middle of a *read()* or *write()* call, so the Inode Pager must read the file size from the current holder of the Token without taking that Token away.

4.3. CHORUS/MiX V.4 *Read()* Path

This section illustrates how a Unix *read()* system call is handled in the CHORUS/MiX V.4 system. When a Unix process traps to the system, it executes the trap handler connected by the Process Manager. The PM performs various checks and invokes the CHORUS Nucleus call *sgRead()*. The Nucleus looks for the corresponding page. If the page is found, data is directly copied into the user's buffer. If the page is not found, an upcall to the appropriate mapper is performed by sending a message to the port whose UI is part of the file capability. In this case, the mapper that implements the disk driver reads the data from the disk and replys to the Nucleus which copies out the data into the user's buffer. The overall mechanism is summarized in the figure 4-2.

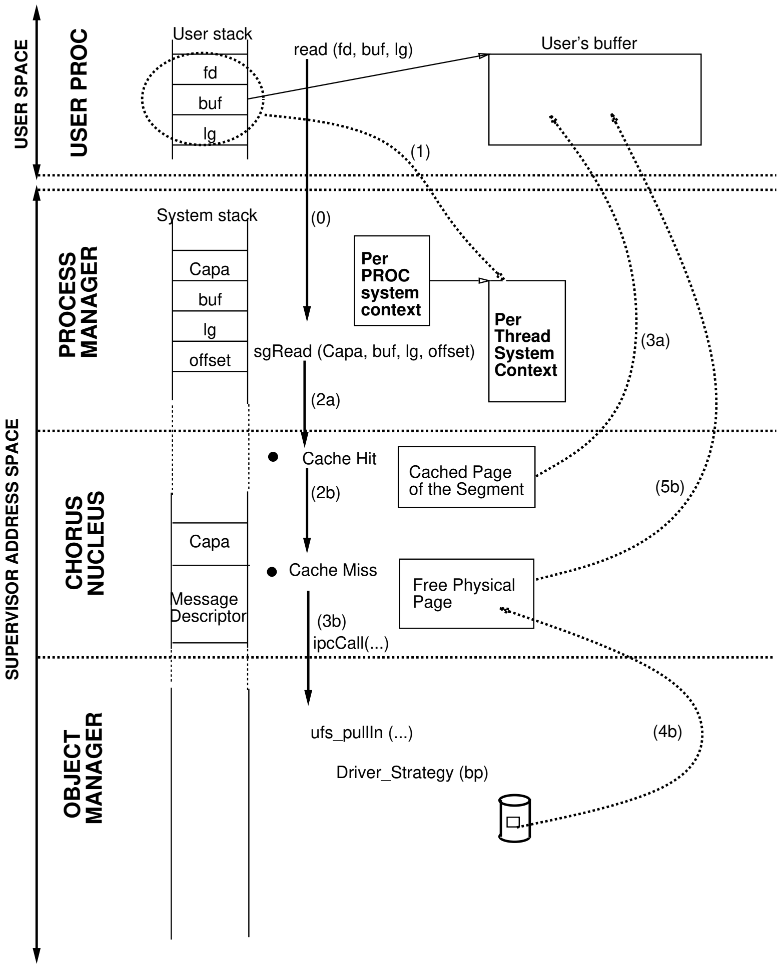

Figure 4-2: CHORUS/MiX V.4 read path

4.3.1. Process Manager

As in any Unix system, the library part of a system call runs in user space and builds a stack frame with the system call arguments and then traps. The trap is handled by the CHORUS Nucleus which redirects the invocation to the trap handler connected by the PM at init time by means of the *svTrapConnect()* Nucleus call (arrow 0 in figure 4-2).

The trap handler executes in supervisor space in the context of the process invoking the system call. The system stack used by the trap handler is the one provided by the CHORUS Nucleus at thread creation time. Using software registers provided by the CHORUS Nucleus, the handler retrieves the thread and process system context. The user stack frame is then copied into the thread structure since multiple threads can run concurrently within a process (arrow 1).

Once the file descriptor has been validated, the PM distinguishes between files that are accessible through the Virtual Memory Interface, such as regular files and mappable character devices, and other files such as directories and streams that are read and written by sending messages to the appropriate server. The PM then acquires the current offset for the file. This requires that it hold a Token in the distributed release of the system. When the system runs on a single site, the PM is assured that the offset information it has is always valid.

The PM then must determine whether the read starts beyond the current end-of-file, or not, and thus must acquire the file size information. This information is protected by a token mechanism which is separate from the Token for the current offset. At the time the *read()* occurs, the Token might not be present in the PM, but may have been recalled by the OM. Finally, after having checked the read operation against mandatory locks (if any), the PM invokes the *sgRead()* CHORUS Nucleus call.

4.3.2. CHORUS Nucleus

The CHORUS Nucleus first tries to retrieve the local cache [1] associated with the segment being accessed. A local cache is created for the segment upon the first access to the segment. Through the local cache descriptor the Nucleus accesses the pages of the segment that are present in main memory, and their associated access rights whether readable or writable. In case the page being read is present (arrow 2a) in the local cache, data is copied out to the user's buffer (arrow 3a). If not present, a free page is acquired (arrow 2b) and the Mapper managing the segment is invoked (arrow 3b). Upon return from the Mapper, the data is copied out into the user's buffer (arrow 5b).

It should be noted that the *sgRead()* operation insures the seriability of concurrent overlapping reads and and writes on a segment.

4.3.3. Mapper

In MiX V.4, regular files are managed by the Object Manager, which acts as the Mapper for these files and thus handles pullIn (read page) and pushOut (write page) requests for these files. When the Object Manager runs in supervisor address space it uses a message handler to handle incoming requests. The client thread is made to run in the server context. Direct access to invoker message descriptors is permitted thus avoiding extra copy of data.

The Object Manager then invokes the corresponding function (e.g.: *ufs_pullin()*) using an extension to the vnode_ops mechanism. The file system specific pullin function converts the page offset in the file into a physical block number and then invokes the disk driver to load data. The buffer area is set up so that it points directly to the physical page provided by the Nucleus and passed as part of the reply message (arrow 4b). The OM only buffers inodes, directory and indirect blocks.

4.4. CHORUS/MiX V.4 *Write()* Path

The basic mechanisms are similar to the ones described for the *read()* system call. However, a *write()* may extend a file or fill a hole in the file. In such a case, the system must immediately allocate disk blocks for the written data.

When the Object Manager returns a page of a file to the Nucleus, it returns also an associated access right enabling the page to be read-only or read-and-written. When a process wants to extend a file by writing after the EOF, the Nucleus requests the entire page for read/write access if there was a cache miss, or only the write permission if there was a cache hit, with no previous write access granted by the Mapper.

Before the Object Manager returns a write permission for a page, it insures that blocks will be available later to write the data on the disk. Blocks are allocated to the file at the time the write permission is required. When the file is closed, blocks that have been allocated but which are not used (past the EOF) are returned to the free block pool.

Finally if the extension of the file has been successful, the PM changes the size of the file. This requires that the PM has the file size token with a write access granted to the size information.

5. Performance

This section contains the performance breakdown for the *read()* and *write()* calls described in the previous sections plus composite numbers for both these systems and from Mach 2.6 MSD(BSD4.3) and System V Release 4.0/386. The micro-benchmarks show where the time is consumed in the full *read()* and *write()* call. The benchmarks for Mach 3.0 with the BSD4.3 Single Server and Mach 2.6 MSD(BSD4.3) were run on a HP Vectra 25C 80386/25Mhz with 16MB and 32K cache, a WD 1007 disk controller and a 340MB disk. The benchmarks for CHORUS/MiX V.4 and System V R4.0 were run on Compaq Deskpro 80386/25Mhz with 16MB of memory and 32K SRAM cache, a WD 1007A disk controller and a 110MB disk drive. The testing methodology was to run a given test three to ten times and to report the median value.

As a number of tests show, the two test platforms, HP Vectra and Compaq Deskpro, have quite different disk and memory throughput performance characteristics. Basically, the HP Vectra memory throughput is between 40% and 50% higher than the Compaq Deskpro throughput (see table 5-1). In comparing the performance of the different systems below, it is essential to remember these differences and relate the figures to the maximum memory and disk throughput of the two different platforms.

To measure the performance of the memory system a simple bcopy was run on both machines. Repeated movement of 4096 bytes was used for the cached result and repeated movement of between 256k and 1M was used for the uncached result. Table 5-1 shows throughput and the time for copying a page of 4096 bytes.

	HP Vectra	Compaq
Cached	21.2MB/sec	14.7MB/sec
	184µs	265µs
UnCached	10.4MB/sec	6.4MB/sec
	379µs	615µs

Table 5-1: Bcopy of 4096 bytes

Two of the primary primitives used by the kernelized systems are trap times and Remote Procedure Call or RPC times. We measured the cost of reaching the Emulation Library or Trap Handler in the kernelized system, versus the cost of a kernel trap in the monolithic systems. For this test we used *getpid()* on all architectures. To show how much overhead there is in reaching the Emulation Library or Trap Handler for a Unix call, including such things as checking for signals, we measured the cost of a null trap into the kernel. Table 5-2 also shows RPC times for the two kernelized systems. The Mach 3.0 RPC times are between two user tasks since this is the type of communication that occurs between the Emulation Library and the BSD4.3 Single Server. The Chorus Null RPC times corresponds to a null *ipcCall()* issued from a supervisor actor to another supervisor actor handling messages via a message handler, which is what is used between PM and OM, and between Chorus Nucleus and OM. CHORUS/MiX V.4 has a more expensive getpid() call than the native SVR4 implementation. This is due, at least partially, to the MiX PM having already implemented support for multithreaded processes and multiprocessor platforms. Thus, MiX has some synchronization mechanisms that are not present within the native Unix SVR4.

	HP Vectra		Compaq	
	Mach 3.0	Mach 2.6	CHORUS/MiX V.4	SVR4.0
Null Trap	40µs	61µs	36µs	85µs
Trap to Unix	61µs	61µs	119µs	85µs
Null RPC	310µs		83µs	

Table 5-2: Trap and RPC times

The primary measurement we are concerned with in this paper is *read()* and *write()* throughput. Table 5-3 measures throughput on all four systems for moving data to and from disk plus the extrapolated time it took for each 4096 byte page. The *read()*'s and *write()*'s were large enough so no caching effects would be seen. As can be seen, the only case where the kernelized system does not perform as well as the monolithic system is the Mach 3.0 *write()* which performs 6% slower.

	HP Vectra		Compaq	
	Mach 3.0	Mach 2.6	CHORUS/MiX V.4	SVR4.0
Read	320KB/sec	320KB/sec	270KB/sec	270KB/sec
	12.5ms	12.5ms	14.8ms	14.8ms
Write	300KB/sec	320KB/sec	250KB/sec	250KB/sec
	13.3ms	12.5ms	16.0ms	16.0ms

Table 5-3: Read and Write Throughput

To approximate the cost of the code path for *read()* and *write()* we measured the cost of *read()*'s without requiring disk access. For this test we did a sequential read of a file of approximately 2 megabytes in size for the kernelized systems and SVR4 and a file as large as possible and still able to fit into the buffer cache for Mach 2.6. Table 5-4 shows these results.

HP Vectra		Compaq	
Mach 3.0	Mach 2.6	CHORUS/MiX V.4	SVR4.0
4.3M/sec	4.3M/sec	3.3M/sec	3.0M/sec
900µs	900µs	1100µs	1300µs

Table 5-4: Cached Read Throughput

The next benchmark measures the cost for repeated calls to *read()* or *write()* of one byte with a *lseek()* between each iteration. So that the *lseek()* time can be removed for further comparisons, a benchmark to measure *lseek()* time is included. Table 5-5 shows the results of these tests. The read time for CHORUS/MiX V.4 is consistent with the equivalent time measured for SVR4: the difference (80µs)corresponds roughly to the additional cost of the trap handling (36µs for the seek call and for the read call). Among the four systems, all of them except SVR4 exhibit equivalent times for reading and writing one byte. The reason why SVR4 writes much faster one byte than it reads it, is quite unclear.

Table 5-4 measured the cost of reading 4096 bytes from a file when reading sequentially through the entire file. The difference between that result and the results shown in table 5-5 should be the difference of

	HP Vectra		Compaq	
	Mach 3.0	Mach 2.6	CHORUS/MiX V.4	SVR4.0
Lseek	95µs	79µs	155µs	121µs
Read	210µs	390µs	710µs	630µs
Write	200µs	380µs	720µs	420µs

Table 5-5: Cached Read and Write Times

an uncached and a cached bcopy and the addition of a fast page-fault resolved in the VM cache. For the Mach 3.0 system table 5-4 also includes the cost of a mapping window change every 16 iterations of 4096 bytes. To account for the mapping window change operation in Mach 3.0 with the BSD4.3 Single Server, a test was run which *lseek()*ed 61440 bytes after each 4096 byte read to force a mapping operation on each *read()*. This resulted in a cost of 2250µs for each iteration including the *lseek()* cost. By looking at 16 reads covering 64KB, we can extrapolate the cost of just the mapping operation and the cost of a page-fault. The measured result from table 5-4 is 900µs times 16 which must be equal to 2250µs plus 15 times the quantity 115µs, from table 5-5, plus 379µs plus the page-fault time. From this, the cost of the fast page-fault time is 316µs. By breaking down the 2250µs measurement, we get the cost of the mapping operation to be 1440µs.

Another test of interest both for understanding the break down of *read()* costs and for comparing between the kernelized systems and monolithic systems is the reading of data directly from raw disk partitions. Table 5-6 shows the throughput and per 4096k byte page performance of Mach 3.0 and Mach 2.6 for different size reads. When these tests were run with block increments of one between each read, the performance was lower than that actually measured for a full *read()*. This result is consistent with the BSD4.3 Single Server's uses of read-ahead which will not consistently miss disk rotations like this test must have. To account for this and better simulate what is happening in the BSD4.3 Single Server, various block increments from one to eight times the data size were used in the benchmark and the increment which produced the best result was reported. An interesting point about the results is that Mach 2.6 reads from raw partitions slower than it reads from Unix files.

Bytes	4KB	8KB	16KB	32KB	64KB
Mach 3.0	10.5ms	10.0ms	7.5ms	7.5ms	6.5ms
device_read()	380KB/sec	410KB/sec	530KB/sec	530KB/sec	620KB/sec
Mach 2.6	14.3ms	14.3ms	14.3ms	14.3ms	14.3ms
read()	280KB/sec	280KB/sec	280KB/sec	280KB/sec	280KB/sec

Table 5-6: Raw Read Performance

The CHORUS Nucleus does not provide any direct access to devices. MiX Object Manager accesses directly the disk driver as a native Unix implementation does, through a bdevsw/cdevsw interface. Tests were done to measure the disk throughput from within the OM using 4KB, 8KB 16KB and 32KB transfers. The test was run in two different ways: the first run was reading the disk sequentially while the second run was always reading the same block of the disk. Reading the same block goes faster than reading the disk sequentially, because the controller has an internal cache.

As it was difficult to get the same measure for Unix SVR4, we measured the disk throughput through the raw disk interface using standard read system calls, the throughput achieved is 530KB/sec for 4096 byte

Bytes	4KB	8KB	16KB	32KB
Sequential	7.1ms	7.1ms	7.1ms	7.1ms
read	560KB/sec	560KB/sec	560KB/sec	560KB/sec
Same Block	4.49ms	4.19ms	4.10ms	4.00ms
read	890KB/sec	952KB/sec	975KB/sec	990KB/sec

Table 5-7: CHORUS/MiX V.4 Raw Read Performance

pages.

The results in table 5-8 were gathered to show the combined effect of reading from disk and supplying the data to the VM system and to show what potential improvements could be made by tuning the file system implementation to the kernelized architecture. The Data_Provided column corresponds to *memory_object_data_provided()* the obsolete call that is used in the current implementation. Data_Supply corresponds to *memory_object_data_supply()*, the new Mach call which has the previously mentioned page stealing optimization. Like the results from table 5-6, the block increment was adjusted to remove disk rotation misses. Both tests reached maximum throughput of 530KB/sec at 16k reads and maintained that speed for larger block sizes.

Comparison of Mach Per Page Fault Cost			
Bytes	4KB	8KB	16KB
Data_Provided	12.1ms	10.2ms	7.5ms
	330KB/sec	400KB/sec	530KB/sec
Data_Supply	11.5ms	10.0ms	7.5ms
	346KB/sec	410KB/sec	530KB/sec

Table 5-8: Mach 3.0 Data_Provided and Data_Supply

The cost of a pullIn operation in the CHORUS Nucleus has been measured by having the Mapper repeatedly load the same disk page into physical memory. With a 4096 byte page, this operation takes 5ms, which is a throughput of 800KB/sec. To determine the overhead introduced by the OM, the cost of reading 4096 bytes from the disk controller's cache must be eliminated. Subtracting this cost, which was measured to be 4.49ms, results in an overhead of 510µs. This time includes the 83µs for an RPC from the Nucleus to the OM. Thus, the actual overhead induced by the OM is 430µs.

To bring together all of the previous results, table 5-9 shows a break down of the *read()* path with associated costs. The micro-benchmark total comes out only 4% higher than the measured result. Since the measurements for *device_read()* and *memory_object_data_provided()* are only an approximation, the measured and projected totals can be considered the same. The Mach 2.6 numbers were included for comparison. Since there is no easy way to measure the internal access time to the disk driver for reading, an extrapolated value was supplied for read which yielded a identical measured and projected total.

While the end measured result for *read()* and *write()* performance is the same in Mach 3.0 with the BSD4.3 Single Server and Mach 2.6 MSD(BSD4.3), table 5-9 shows a result which may question scalability the result under load. Because the disk latency is so high, the 262% increase in processing overhead necessary in the BSD4.3 Single Server is hidden. Further measurements should look at whether the effects of this increased processing outweigh the benefits seen in cached performance where the

Operation	Time (μs)	Time (μs)
	Mach 3.0	Mach 2.6
Trap to Unix	61	61
Misc Costs	54	250
1/16th Remap Window	90	N/A
Pagefault	316	N/A
Read from Disk	10500	11810
Data Provided	1600	N/A
Bcopy Uncached to User	379	379
Total w/o Disk Read	2500	690
Total	13000	12500
Measured Total	12500	12500

Table 5-9: Mach 3.0 with the BSD4.3 Single Server *Read()* Path

BSD4.3 Single Server *read()* overhead is only 54% of the Mach 2.6 MSD(BSD4.3) *read()* overhead.

6. Conclusions

6.1. Compromises

The kernelized systems have achieved their performance in a number of cases by compromising some of their modularity and portability goals.

Mach 3.0 with the BSD4.3 Single Server makes liberal use of the shared memory pages between the BSD4.3 Single Server and Emulation Library. While this works well on a uniprocessor or shared memory multiprocessor, it would not work well on a NUMA or NORMA machine. The mapping information that is used by the Emulation Library could be updated by messages instead of being written directly by the BSD4.3 Single Server. This would eliminate the *passive* Token state optimization since there is no mechanism for an upcall from the BSD4.3 Single Server to the Emulation Library. A dedicated thread in the Emulation Library for upcalls would solve this problem, but with the cost of adding an additional thread creation at each *fork()* call. Another solution would be to have a proxy task running on each node on behalf of the BSD4.3 Single Server to share the memory with the Emulation Library and respond to upcalls. The cost here would be at startup in the creation of an additional task per node.

CHORUS/MiX V.4 is capable of running its system servers in either supervisor or user mode. The configuration used for the previous section's measurements was the version where the servers reside in supervisor mode. This results in no context switches and faster IPC times to read or write from/to the disk. Moreover, CHORUS/MiX V.4 servers do not share any memory, thus it is quite easy to let a component evolve without changing the other servers as long as the protocol between them remains unchanged. The configuration where CHORUS/MiX V.4 is running with its servers in user mode is primarily intended to be used for debugging purposes, thus no particular attention has yet been paid to achieving similar performance in such a configuration. However, some experiments have been done in the previous release of MiX (MiX V3.2) to measure the additional costs that such a configuration will imply. Further details can be found in [3].

One of the key points in the design of CHORUS/MiX V.4 has been the introduction of the message handler mechanism. However, this mechanism should be extended to work whether the receiver of a message is running in supervisor address space or user address space.

6.2. Cross Pollenation

A number of techniques used by CHORUS/MiX V.4 could readily be adopted by Mach 3.0 with the BSD4.3 Single Server. The movement of device drivers, such as, disk drivers, out of the kernel or nucleus into systems servers is one example of this. By locating the device manipulation code in the same task that needs the data, context switches and data copies can be avoided. For this to work on a Mach based system, the device driver needs to be able to run in user mode where systems servers are run. This is feasible on architectures that allow user mode access to device registers such as the R2000/R3000 based DECstation and 80386/80486 based platforms. The current version of the BSD4.3 Single Server for the DECstation platform uses a user mode ethernet driver and preliminary work has been done on moving disk drivers.

Another technique, which CHORUS/MiX V.4 uses, that could be adopted by Mach 3.0 with the BSD4.3 Single Server, is the concept of a handler. By having the kernel redirect the trap directly into another address space, CHORUS avoids the protection problem the BSD4.3 Single Server has with the Emulation Library and avoids the associated RPC needed by the Emulation Library to cross into a trusted server. The question about this technique, is whether the additional cost of always redirecting into another address space outweighs the cost of the occasional RPC. Since Mach 3.0 servers are run in user mode, the cost of this redirection may turn out to be much higher than that seen by CHORUS for the redirection into a supervisor mode actor.

Mach 2.6 MSD(BSD4.3) could benefit from a mapped file system using the Emulation Library technique. This will only work because it already has the complex VM system, a fast IPC, and the Trap Emulation mechanism that resides in Mach 3.0. In general, monolithic systems can not benefit the kernelized techniques because they do not have the necessary mechanisms for building system servers with new structures.

To fully compare the two approaches taken by Mach 3.0 and CHORUS, one could use the CHORUS Trap Handler mechanism to implement an Emulation Library instead of a Process Manager in CHORUS. As long as device drivers are part of CHORUS subsystems' specific actors, it will be difficult to have multiple subsystems running simultaneously sharing the same devices. The isolation of the drivers out of the servers as done in Mach would help to solve this issue. This has been experimented in CHORUS/MiX V3.2 and described in [3].

6.3. Summary

Section 5 shows that Mach 3.0 with the BSD4.3 Single Server and CHORUS/MiX V.4 have achieved performance in the movement of data comparable to the monolithic systems which they are compatible with. In no case was *read()* or *write()* throughput less then 94% of the performance of the monolithic system. Many of the micro-benchmarks clearly indicated better performance on kernelized systems for certain types of cached access.

There is also still room for improvement in Mach 3.0 with the BSD4.3 Single Server. By moving to the *memory_object_data_supply()* call from the obsolete interface and better optimizing read sizes and read-ahead for the performance of the microkernel, disk throughput could approach 600KB/sec or almost a 100% improvement over the existing BSD4.3 Single Server and Mach 2.6 MSD(BSD4.3) systems.

CHORUS/MiX V.4 may also expect some significant improvement since no readahead is used in the systems that have been described and measured. Pages being pushed out to the disk are written synchronously. Making them asynchronous should help to handle multiple disk accesses more gracefully by making the disk sort algorithm much more useful.

The current performance of microkernels has been achieved through multiple iterations. This allows microkernel based systems to now be used in commercial products as opposed to just being interesting research tools. Yet, microkernel based systems are still very young and have not benefited from the thousands of programmer-years that have been spent to make monolithic systems as fast as they are now. Even as young as they are, we believe that kernelized systems have shown themselves to be both flexible and fast enough for the challenging task of building file systems and moving data. It can only be imagined what kernelized systems will look like a few years from now, after receiving a fraction of the effort that has gone into monolithic systems in the past.

7. Availability

Mach 3.0 is free and available for anonymous FTP from cs.cmu.edu. The BSD4.3 Single Server is available free to anyone with an AT&T source license. Contact mach@cs.cmu.edu for information. CHORUS is available from CHORUS Systemes. Contact info@chorus.fr for licensing information. Reports and documentation is freely accessible by anonymous FTP from opera.chorus.fr.

8. Acknowledgements

We would like to thank those who have read and commented on this paper throughout its development. In particular, we thank Dr. J. M. Karohl, Brian Bershad, Jim Lipkis, Michel Gien, Marc Rozier, Brian Zill, and, especially, Peter Stout.

References

[1] Abrossimov V., Rozier M., Shapiro M.
Generic Virtual Memory Management for Operating System Kernels.
In *Proceedings of the Twelvth ACM Symposium on Operating Systems Principles*. December, 1989.

[2] Armand F., Gien M., Herrmann F., Rozier M.
Revolution 89: or Distributing Unix brings it back to its Original Virtue.
In *Proceedings WEDMS I*. October, 1989.

[3] Francois Armand.
Give a Process Manager to your drivers!
In *Proceedings of EurOpen Autumn 1991*. September, 1991.

[4] Baron, R.V. et al.
MACH Kernel Interface Manual.
Technical Report, School of Computer Science, Carnegie Mellon University, September, 1988.

[5] Joseph S. Barrera III.
Kernel Support for Distrubted Memory Multiprocessors.
PhD thesis, School of Computer Science, Carnegie Mellon University, To be published, 1992.

[6] Bricker, A., Gien, M., Guillemont, M., Lipkis, J., Orr, D., Rozier, M.
A New Look at Microkernel-Based UNIX Operating Systems: Lessons in Performance and Compatibility.
In *Proceedings of EurOpen Spring 1991 Conference*. May, 1991.

[7] Cheriton, D.R., Whitehead, G.R., Szynter, E.W.
Binary Emulation of UNIX using the V Kernel.
In *Proceedings of Summer 1990 USENIX Conference*. June, 1990.

[8] Chorus Systemes.
CHORUS Kernel v3 r4.0 Programmer's Reference Manual.
Technical Report CS/TR-91-71, Chorus Systemes, September, 1991.

[9] Rozier, M., et. al.
CHORUS Distrbuted Operating Systems.
Computing Systems 1(4), December, 1988.

[10] Chorus Systemes.
CHORUS Kernel v3 r4.0 Specification and Interface.
Technical Report CS/TR-91-69, Chorus Systemes, September, 1991.

[11] Eric C. Cooper and Richard P. Draves.
C Threads.
Technical Report CMU-CS-88-154, School of Computer Science, Carnegie Mellon University, February, 1988.

[12] Draves, R.P.
A Revised IPC Interface.
In *Proceedings of the USENIX Mach Workshop*, pages 101-121. October, 1990.

[13] Draves, R.P., Bershad, B., Rashid, R.F., Dean, R.W.
Using Continuations to Implement Thread Management and Communication in Operating Systems.
In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*. April, 1991.

[14] Golub, D., Dean, R.W., Forin, A., Rashid, R.F.
Unix as an Application Program.
In *Proceedings of Summer 1990 USENIX Conference*. June, 1990.

[15] Guillemont M.
Real Time in a Distributed Computing Environment.
Computer Technology Review , October, 1990.

[16] Habert, S., Mosseri, L., Abrossimov, V.
COOL:Kernel support for object oriented environments.
In *Proceedings of OOPSLA'90 - Canada Ottawa*. 1990.

[17] Keith Loepere.
MACH 3 Kernel Principles.
Open Software Foundation and Carnegie Mellon University, 1992.

[18] A. S. Tannenbaum and S. J. Mullender.
An Overview of the Amoeba Distributed Operating System.
CWI Syllabus 9.
1982

[19] Tokuda, H., Nakajima, T., Rao, P.
Real-Time Mach: Towards a Predictable Real-Time System.
In *Proceedings of the First Mach USENIX Workshop*, pages 73--82. October, 1990.

[20] Brent Welch.
The File System Belongs in the Kernel.
In *Proceedings of the Second USENIX Mach Symposium*. November, 1991.

[21] Michael Wayne Young.
Exporting a User Interface to Memory Management from a Communication-Oriented Operating System.
PhD thesis, School of Computer Science, Carnegie Mellon University, November, 1989.

Table of Contents

List of Figures

List of Tables