A Unix Interface for Shared Memory and Memory Mapped Files Under Mach

Avadis Tevanian, Jr., Richard F. Rashid, Michael W. Young,
David B. Golub, Mary R. Thompson, William Bolosky and Richard Sanzi

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract

This paper describes an approach to Unix shared memory and memory mapped files currently in use at CMU under the Mach Operating System. It describes the rationale for Mach's memory sharing and file mapping primitives as well as their impact on other system components and on overall performance.

1. Introduction

The 4.2 BSD mapped file interface (*mmap*) was designed to address two shortcomings of previous Unix systems: a lack of shared memory between processes and the need to simplify processing of file data. Early Unix systems had provided no shared memory access and a stylized way of accessing sequential file data through *read* and *write* system calls. Applications that desired random access to data would use Unix's *seek* operation or buffer their data themselves, often incurring unwanted system overhead. A mapped file facility could allow a user to treat file data as normal memory without regard to buffering or concerns about sequential versus random access. It would also provide an obvious mechanism for sharing memory by allowing more than one process to map a file read/write simultaneously.

The BSD file mapping facility was proposed as early as 1982. Since then, similar mapped file interfaces have been implemented by several vendors, both as part of 4.2 BSD Unix (e.g., Sequent Dynix [Dynix]) and as part of a System V modified to contain 4.2 BSD enhancements (e.g., IBM's AIX). A shared memory facility not based on mapped files is available in AT&T's System V and has also been adapted to a variety of 4.2 BSD based systems such as DEC's Ultrix. There is currently a lively debate going on within the Unix community about the appropriate Unix interface to virtual memory and the relationship between mapped files, memory sharing and other virtual memory concerns such as copy-on-write memory mapping.

This paper describes the somewhat atypical approach to shared memory and file mapping currently in use at CMU under the Mach Operating System. It describes the rationale for Mach's memory sharing and file mapping primitives, their impact on other system components and on overall performance and the experiences of the Mach group in implementing and using them.

2. The Problems of a Mapped File Interface

Despite its obvious convenience, the notion that all memory sharing should be channeled through a mapped file interface presents a number of problems:

1. A Unix file is (in principle) a permanent on-disk data structure which must be maintained consistent against crashes. The use of disk files to exchange temporary data can put an unnecessary I/O load on the system and impact performance.

2. A mapped file facility must take into account the sharing of remote (network) files. In order to handle remote file systems (e.g. SUN NFS), the operating system must be intimately involved in maintaining network data consistency. This can increase its complexity considerably by introducing within the OS kernel many of the same concerns that complicate transaction processing systems.

3. Sharing semantics are limited to those supplied by the kernel. In particular, an application program cannot use domain specific knowledge to allow less then full consistency in sharing access to file data. This can result in inefficiency in the handling of data sharing across node boundaries.

These problems typically have led to compromises in the actual mapped file semantics provided. Most have either assumed that modifications to read/write mapped files are not guaranteed to be consistent in the face of multiple writers, or they guarantee consistency only for those processes which share files on a single network node.

3. The Uses of Shared Memory

Many of the potential uses of shared memory do not require a file mapping interface. In fact, such an interface may present problems. Memory sharing is often suggested as a way of overcoming traditional Unix deficiencies by providing for:

- fine granularity multiprocessing,
- ultra-fast IPC,
- database management support and/or
- reduced overhead file management.

But of these potential uses of shared memory, only two require some kind of mapped file facility and of these only one fits the traditional *mmap* model of file access and shared data consistency.

3.1. Fine grain multiprocessing

The need to support fine grain multiprocessing has forced several multiprocessor manufacturers to adopt some form of memory sharing in their multiprocessor versions of Unix, e.g. in Sequent's Dynix [Dynix] and Encore's UMax [Umax]. This kind of shared memory often takes the form of an *mmap*-like primitive which acts on a special device or file. Fine grain multiprocessing is accomplished by creating as many processes as there are available processors which then *mmap* the shared memory object and synchronize through it. The only reason such a facility would want a mapped file interface is the benefit provided by using Unix's filesystem name space for referring to shared data. There is no need for the shared data to be disk resident or permanent.

3.2. Fast IPC

Shared memory can be used as a kind of ultra-fast IPC facility, especially where large data structures are built in shared memory by one process and then managed or manipulated by another. An example of a potential use of this kind can be found in the relationship between multiphase program components such as the typical C language preprocessor and compiler. Already such programs use files or pipes to accomplish their goals. The advantages of such a fast IPC facility are actually diminished by tying it to a similar shared file construct which would require some form of file system creation/destruction cost as well as disk I/O.

3.3. Database management

Designers of database management systems have argued against Unix at least partly because of its inability to share data between potential database client programs and transaction managers, data managers and recovery logs. Systems of this sort need both sharing between processes and sharing of data pages in files to accomplish their ends. Unfortunately, an mmap-like construct does not, by itself, resolve the problems posed by database systems. For example, it may be important for a database system to know when data is going to be moved from volatile storage to disk so that a database recovery manager can update crucial portions of the recovery log in advance [TABSSOSP] (i.e., write-ahead logging). In addition, the consistency of shared memory must either be absolute, or the consistency model must be well understood and manageable by the database transaction manager -- a fact often remarked by database builders on other systems with shared file constructs such as Apollo’s Aegis [Leach83].

3.4. Efficient file access

By far the most compelling general argument for linking shared memory with memory mapped files is the need in Unix for reducing the overhead of file management. Partly because Unix was originally designed at a time when primary memory was a scarce commodity, traditional Unix programs are I/O intensive. Even the Unix pipe facility was once implemented as file I/O to conserve memory. As the relationship between the costs of memory and secondary storage have changed, large memory Unix systems are limited more by their I/O capacity than by memory. A mapped file facility could reduce the cost of I/O operations by eliminating a copy operation from the Unix buffer cache to process memory and also provide for better memory utilization by allowing more than one process to share the same physical memory when accessing the same file.

4. Mach Memory Primitives

Rather than support sharing only through an *mmap* model of shared memory through shared files, Mach provides a number of non-file based mechanisms for sharing data among computational entities:

- Unrestricted, fine grain sharing between processors in a tightly coupled multiprocessor can be achieved by using the Mach notion of *thread*. A thread can be thought of as a lightweight process which shares an address space with other lightweight processes. The Unix notion of *process* has been split into *task* and *thread*. A task defines an address space and resource domain in which a number of program control flows (threads) may coexist. Using this multiple thread per task mechanism, an application may easily share a single address space among separate executing entities.

- In addition to unrestricted sharing using threads, Mach allows tasks to read/write share protected ranges of virtual addresses through inheritance. A Mach task can specify any portion of its address space to be shared read/write with its children as the result of a *task_create* operation. The fact that memory is shared only through inheritance guarantees that the shared memory is always located within a single host (or cluster within a host). This allows the kernel to guarantee cache consistency for such memory. Another advantage of this method of data sharing is that it ensures that shared memory is always located at the same virtual address in each inheriting task. This avoids the often difficult programming problems caused by pointer address aliasing in shared data structures.

- Physical memory can be shared copy-on-write by taking advantage of Mach's integration of IPC and virtual memory management. Applications not requiring read/write memory sharing can use this feature to transfer large amounts of data between tasks without actually copying data. In effect, a multiphase application can effectively forward between components the actual physical memory containing important data. The sender in such an exchange is always protected because data is logically sent by value. The kernel uses memory management tricks to make sure that the same physical page is available to both sender and receiver unless or until a write operation occurs.

- Finally, applications may define their own sharing semantics within a distributed system of Mach hosts using the Mach *external pager* facility. This external pager mechanism allows an application to control many aspects of virtual memory management for regions of virtual memory. An *external pager* may implement fully coherent network shared memory, or a shared memory paradigm that requires clients to maintain their own cache consistency (if consistency is even desired). It allows a database recovery manager to be advised of the kernel's need to flush data to disk in advance and thus permit efficient write-ahead logging.

4.1. Mach virtual memory operations

Table VMOPS lists the set of operations that can be performed on the virtual address space of a task. Mach calls are specified to act on object handles called *ports* which are simplex communication channels on which messages are sent. A more complete description of Mach ports and calling conventions can be found in [USENIX86].

A task address space consists of an ordered collection of mappings to memory objects; all threads within a task share access to that address space. A Mach *memory object* (also called a *paging object*) is a data repository, provided and managed by a server. The size of an address space is limited only by the addressing restrictions of the underlying hardware. For example, an IBM RT PC task can address a full 4 gigabytes of memory under Mach, while the VAX architecture allows at most 2 gigabytes of user address space.

The basic memory operations permit both copy-on-write and read/write sharing of memory

Virtual Memory Operations

vm_allocate (task, address, size, anywhere)	*Allocate and fill with zeros new virtual memory either anywhere or at a specified address on demand.*
vm_copy (task, src_addr, count, dst_addr)	*Virtually copy a range of memory from one address to another.*
vm_deallocate (task, address, size)	*Deallocate a range of addresses, i.e. make them no longer valid.*
vm_inherit (task, address, size, inheritance)	*Set the inheritance attribute of an address range.*
vm_protect (task, address, size, set_max, protection)	*Set the protection attribute of an address range.*
vm_read (task, address, size, data, data_count)	*Read the contents of a region of a task's address space.*
vm_regions (task, address, size, elements, elements_count)	*Return description of specified region of task's address space.*
vm_statistics (task, vm_stats)	*Return statistics about the use of memory by task.*
vm_write (task, address, count, data, data_count)	*Write the contents of a region of a task's address space.*

Table 4-1:

All VM operations apply to a *task* (represented by a port) and all but vm_statistics specify an *address* and *size* in bytes. *anywhere* is a boolean which indicates whether or not a vm_allocate allocates memory anywhere or at a location specified by address.

regions between tasks. Copy-on-write sharing between unrelated tasks is usually the result of large message transfers. An entire address space may be sent in a single message with no actual data copy operations performed. Read/write shared memory within a task creation tree can be created by allocating a memory region and setting its inheritance attribute. Subsequently created child tasks share the memory of their parent according to its inheritance value. The only restriction imposed by Mach on the nature of the regions that may be specified for virtual memory operations is that they must be aligned on system page boundaries. The system page size is a boot time parameter and can be any power of two that is a multiple of the hardware page size.

4.2. Managing external pagers

The basic task virtual memory operations allow memory sharing through inheritance between tasks in the same task creation subtree. Read/write shared memory between unrelated tasks can be implemented through the use of external pagers -- tasks which allocate and manage secondary storage objects.

The Mach interface for external pagers can best be thought of as a message protocol used by a pager and the kernel to communicate with each other about the contents of a memory object. The external pager interface to the kernel can be described in terms of operations requested by the kernel (messages sent to a *paging_object* port) and calls made by the external pager on the kernel (messages sent to the kernel's *pager_request_port* associated with a memory object). Tables EXPOPS and POPS describe these two interfaces.

Kernel to External Pager Interface

pager_init (paging_object, pager_request_port, pager_name)	*Initialize a memory object.*
pager_data_request (paging_object, pager_request_port, offset, length, desired_access)	*Requests data from an external pager.*
pager_data_write (paging_object, offset data, data_count)	*Writes data back to a memory object.*
pager_data_unlock (paging_object, pager_request_port, offset, length, desired_access)	*Requests that data be unlocked.*
pager_create (old_paging_object, new_paging_object, new_request_port, new_name)	*Accept ownership of a memory object.*

Table 4-2:

Calls made by Mach kernel to a task providing external paging service for a memory object.

A memory object may be mapped into the address space of a task by exercising the *vm_allocate_with_pager* primitive, specifying a paging object port. This port will then be used by the kernel to refer to that object. A single memory object may be mapped more than once (possibly in different tasks). The Mach kernel provides consistent shared memory access to all mappings of the same memory object on the same uniprocessor or multiprocessor. The role of the kernel in paging is primarily that of a physical page cache manager for objects.

When asked to map a memory object for the first time, the kernel responds by making a *pager_init* call on the paging object port. Included in this message are:

- a *pager request* port, which the pager may use to make cache management requests of the Mach kernel,
- a *pager name* port, which the kernel will use to identify this memory object to other tasks in *vm_regions* calls.[1]

The Mach kernel holds send rights to the paging object port, and send, receive, and ownership rights on the paging request and paging name ports.

In order to fulfill a cache miss (i.e. page fault), the kernel issues a *pager_data_request* call specifying the range (usually a single page) desired. The pager is expected to supply the requested data using the *pager_data_provided* call on the specified paging request port. To flush modified cached data, the kernel performs a *pager_data_write* call, including the data to be

[1]The paging object and request ports cannot be used for this purpose, as access to those ports allows complete access to the data and management functions.

written and its location in the memory object. When the pager no longer needs the data (e.g. it has been successfully written to secondary storage), it is expected to use the *vm_deallocate* call to release the cache resources.

Since the pager may have external constraints on the consistency of its memory object, the Mach interface provides some functions to control caching; these calls are made using the pager request port provided at initialization time.

External Pager to Kernel Interface

vm_allocate_with_pager (task, address, size, anywhere, paging_object, offset)	*Allocate a region of memory at specified address backed by a memory object.*
pager_data_provided (paging_object_request, offset, data, data_count, lock_value)	*Supplies the kernel with the data contents of a region of a memory object.*
pager_data_lock (paging_object_request, offset, length, lock_value)	*Prevents further access to the specified data until an unlock.*
pager_flush_request (paging_object_request, offset, length)	*Forces physically cached data to be destroyed.*
pager_clean_request (paging_object_request, offset, length)	*Forces modified physically cached data to be written back to a memory object.*
pager_cache (paging_object_request, should_cache_object)	*Notifies the kernel that it should retain knowledge about the memory object even after all references to it have been removed.*
pager_data_unavailable (paging_object_request, offset, size)	*Notifies kernel that no data is available for that region of a memory object.*

Table 4-3:

Calls made by a task on the kernel to allocate and and manage a memory object.

A *pager_flush_request* call causes the kernel to invalidate its cached copy of the data in question, writing back modifications if necessary. A *pager_clean_request* call asks the kernel to write back modifications, but allows the kernel to continue to use the cached data. A pager may restrict the use of cached data by issuing a *pager_data_lock* request, specifying the types of access (of read, write, execute) which may be permitted. For example, a pager may wish to temporarily allow read-only access to cached data. The locking on a page may later be changed as deemed necessary by the pager.

When a user task requires greater access to cached data (e.g. a write fault on a read-only page) than the pager has permitted, the kernel issues a *pager_data_unlock* call. The pager is expected to respond by changing the locking on that data when it is able to do so.

When no references to a memory object remain, and all modifications have been written back to the paging object port, the kernel deallocates its rights to the three ports associated with that memory object. The pager receives notification of the death of the request and name ports, at which time it can perform appropriate shutdown.

In order to attain better cache performance, a pager may permit the data for a memory object to be cached even after all address map references are gone by calling *pager_cache*. Permitting such caching is in no way binding; the kernel may choose to relinquish its access to the memory object ports as it deems necessary for its cache management.

The Mach kernel may itself need to create memory objects, either to provide backing storage for zero-filled memory (*vm_allocate*), or to implement virtual copy operations. These memory objects are managed by a *default pager* task, which is known to the kernel at system initialization time. When the kernel creates such a memory object, it performs a *pager_create* call (on the default pager port); this call is similar in form to *pager_init*. Since these kernel-created objects have no initial memory, the default pager may not have data to provide in response to a request. In this case, it should perform a *pager_data_unavailable* call.

Since interaction with pagers is conducted only through ports, it is possible to map the same memory object into tasks on different hosts in a distributed system. While each kernel keeps its own uses of the cached data consistent, the pager is responsible for any further coordination. Since each Mach kernel will perform a *pager_init* call upon its first use of a memory object, including its own request and name ports, a pager can easily distinguish the various uses of its data.

5. A Unix Interface for File Mapping

Shared memory can be obtained in Mach either through the use of memory inheritance or external pagers. Given these mechanisms for sharing data, there is no need to overload the Unix filesystem in order to provide shared memory. Nevertheless, the potential performance advantages of mapped files make them desirable for Unix emulation under Mach. In addition, the ease of programming associated with mapped files is attractive in both the Unix and Mach environments.

At present, Mach provides a single new Unix domain system call for file mapping:

```
map_fd(fd, offset, addr, find_space, numbytes)
        int             fd;
        vm_offset_t     offset;
        vm_offset_t     *addr;
        boolean_t       find_space;
        vm_size_t       numbytes;
```

Map_fd is called with an open Unix file descriptor (*fd*) and if successful results in a virtual copy of the file mapped into the address space of the calling Unix process. *Offset* is the byte offset within the file at which mapping is to begin. The offset may be any byte offset in the file, page alignment is not required. *Addr* is a pointer to the address in the address space of the calling process at which the mapped file should start. This address, unlike the offset, must be paged aligned. If *find_space* is TRUE, the kernel will select an unused address range and return it in **addr*. The number of bytes to be mapped is specified by *numbytes*.

The implementation of *map_fd* was a straightforward application of internal Mach primitives for virtual copying regions of memory and external pagers [MACH-ASPLOS, MACH-SOSP]. When a request is made for a file to be mapped into a user address space, the kernel creates a temporary internal address space into which the file is mapped. This mapping is accomplished with the *vm_allocate_with_pager* primitive. The kernel specifies that new memory is to be allocated and that the new memory will be backed by the internal kernel *inode pager*. Then the file data is moved to the process address space by a call to *vm_copy*. Once this is done, the kernel can deallocate the temporary map.

6. Uses of Mapped Files in Mach

Files mapped using *map_fd* can be used in a variety of ways. Mach itself uses file mapping internally to implement program loading. File mapping can also be used as a replacement for buffer management in the standard I/O library.

6.1. File Mapping and Shared Libraries

Mach uses the mapped file interface to implement both program loading and a general form of shared libraries. In the current Mach system, there are two types of program loaders. The first program loader executes in the kernel and implements the Unix *exec* system call. This loader handles both *a.out* and *COFF* format binary files for binary compatibility with existing systems. The second loader executes in a user task and handles *MACH-O* format binary files. Both loaders use mapped files.

The MACH-O format was devised to be flexible enough to be used as a single file format for fully resolved binaries, unresolved object files, shared libraries and "core" files. It provides enough backward compatibility with older formats (e.g., a.out) to salvage most existing code for debuggers and related applications.

The MACH-O format can roughly be though of as a sequence of commands to be executed by a program loader. The layout of a MACH-O file is summarized as:

```
start
        header
        command_id, command_info
        command_id, command_info
                .
                .
                .
        command_id, command_info
ENDMARKER
```

Each command consists of a command identifier followed by a command-dependent number of arguments. Some of the commands supported are:

READ_ONLY	Map in data read-only (e.g. a text segment).
WRITEABLE	Map in data read/write (e.g. a data segment).

ZEROFILL	Allocate zero-fill memory (e.g. a bss segment).
REGISTER	Create a thread in the task and set its register state.
LOADFILE	Map in data from another file (e.g. a shared library).
RELOCATE	Relocate a specified address.
END_LOAD	Loading complete.

The *header* contains a magic number indicating MACH-O format. It also contains other useful information such as version information, and a machine-type specifier. Finally, the header specifies the type of file represented, e.g. executable, object file or shared library.

The MACH-O program loader operates by scanning a load file and executing commands as necessary. In the typical case, it uses the *map_fd* call to map portions of files into its address space. It then places the data in the image to be executed using the *vm_write* operation. Since copy-on-write is used at the base of the virtual memory primitives it is possible to share both code and writable data. Each task that writes data within a shared library will get a new copy as each page is written for the first time. Pages that are not written will be physically shared by all tasks.

6.2. File Mapping and Standard I/O

The Mach mapped file mechanism has been used to build a new version of the C library buffered I/O package. When a file is *fopen*ed it is mapped in its entirety into the caller's address space. The semantics of the buffered i/o package are not changed. The existing stdio buffer has, in effect, been enlarged to the size of the file. When a write takes place only the data buffer is changed. The file is not guaranteed to change on disk until a *fflush* or *fclose* takes place. As with normal buffered I/O, if two processes have the same file open for reading and writing, there is no guarantee how the reads and writes will intermix. A read may get new information off the disk copy of the file, or it may use information that was already buffered.

The primary rationale for this change is improved performance. Table STDIOPERF shows the time for simple buffered I/O operations both with and without the change. In addition to improved performance, the use of file mapping also has the effect of reducing the memory load on the system. In a traditional Unix implementation *fopen* would allocate new memory to the calling process and copy the data from the Unix buffer cache into that new memory at the time of a *read*. Using this new package and Mach file mapping, each new call to *fopen* will reuse any physical memory containing file data pages, reducing the number of I/O operations. (See table STDIOPERF2.)

In addition to traditional buffered I/O calls, the mapped file version of buffered I/O has had added to it a new call which allows an application program to directly access the mapped file data and thus further improve performance by eliminating the copying of data by *fread* and *fwrite*. The new routine is called *fmap* and is a buffered I/O compatible version of *map_fd*.

To read map a file with fmap the user calls:

Unmapped vs. Mapped Buffered I/O Performance

Test program	First time user system elapsed			Second time user system elapsed		
old_read	6.1u	0.62s	0:08	6.1u	0.62s	0:08
new_read	6.0u	0.71s	0:08	6.0u	0.21s	0:06
map_read	2.8u	0.76s	0:04	2.7u	0.17s	0:03

Table 6-1:

Time to read a 492544 byte file using standard I/O.
(Mach, 4K file system, MicroVAX II)

old_read performs *fopen* followed by a loop of getc calls.
new_read is identical to old_read with new mapped file package.
map_read uses fmap and reads data by array reference, not getc.

Multiple Access File I/O Performance

Test program	user	system	elapsed	I/O
old_read[1]	25.4u	1.8s	1:23	217io
old_read[2]	25.3u	2.0s	1:26	326io
old_read[3]	25.1u	2.2s	1:26	439io
new_read[1]	24.0u	1.6s	1:17	89io
new_read[2]	24.0u	1.8s	1:18	194io
new_read[3]	24.2u	1.6s	1:18	197io

Table 6-2:

Time to read a 1970176 byte file using standard I/O.
(Mach, 4K file system, MicroVAX II)

Each program is run 3 times in parallel and times are listed.
Instance numbers for each invocation are in brackets.
Each program accesses the same file simultaneously.

old_read performs *fopen* followed by a loop of getc calls.
new_read is identical to old_read with new mapped file package.

```
stream = fopen("filename", "r");  /* existing call */
data = fmap(stream, size);        /* new call */
```

where data is a pointer to a region of virtual memory where the file contents are buffered, and size is the suggested size for the data buffer; if that size is zero, then the implementation will choose a suitable size. As before,

```
bufsize = fbufsize(stream)
```

returns the actual size of the buffer. Once *fmap* is called, the user can reference file data by using the data pointer and any offset less than bufsize. The user may also mix *fseek,* and *fread* calls with direct data references. Once the user is finished with the file the call

```
fclose(stream);                 /* existing call */
```

should be used to deallocate the virtual address space used by the mapped file.

To write map a file the user would:

```
    stream = fopen("filename", "w");/* existing call */
    data = fmap(stream, size);      /* new call */
```

where size is used as an initial buffer size; if that size is zero, the implementation will choose a suitable size. Initially, the buffer will be zero-filled. Once *fmap* is called, the user may write into any part of the file with an offset less than bufsize. An *fwrite* or *fseek* call with an offset greater than bufsize will cause an error. To expand the buffer size, the user may call *fmap* again with a larger size parameter. The calls

```
    fflush(stream);                 /* existing call */
    fclose(stream);                 /* existing call */
```

continue to work as before. Similarly, files opened for append and read/write may be *fmap*ed.

Table STDIOPERF3 shows the time advantage which can be gained by using fmap rather than conventional I/O.

Unmapped vs. Mapped Buffered I/O Performance

Test program	First time user system elapsed I/O	Second time user system elapsed I/O
old_read	11.5u 3.1s 0:21 481io	11.5u 3.0s 0:21 482io
map_read	11.2u 2.9s 0:15 480io	11.0u 0.9s 0:12 0io

Table 6-3:

Time to read a 1970176 byte file.
(Mach, 4K file system, MicroVAX II)

old_read performs *open*, *malloc*s buffer, calls *read* for whole file and then reads data by array reference.

map_read uses *fopen* and *fmap* and reads data by array reference.

7. The Effect of Mach Memory Mapping on Performance

File mapping is hardly free[2]. Even when a page is already in physical memory, a page fault must be taken on the first process access to validate the corresponding hardware map entries. Currently such a fault takes approximately 1.0-1.4 milliseconds on a MicroVAX II with a 4K page size. There are also several ways in which mapped files can adversely affect performance:

- If the file to be mapped is smaller than a single page, file mapping will always result in a full page being allocated in physical memory with excess data filled with zeroes.
- Mapped files compete with program text and data for physical memory. In a traditional Unix system, user programs maintain a fixed-size buffer, so the buffer cache limits the amount of memory which can be consumed in accessing a file.

Nevertheless, as the performance of the new standard I/O library points out, useful performance gains can be achieved using Mach memory mapping. In fact, because the Mach kernel uses

[2]Unless the output is going to /dev/null!

mapped files internally to implement *exec*, overall performance of vanilla 4.3 BSD programs is often improved when run on Mach. Particularly dramatic performance gains are seen on machines where the processor speed is high, memory is plentiful and disk is a bottleneck. For example, performance gains of over 20% have been achieved on a VAX 8650. Improvement can be found, however, even on small memory systems with moderately heavy loads. The multiuser benchmark load used to study the performance of the CMU ITC VICE/VIRTUE file system [SATYA85] ran 10-15% faster under Mach than a comparable BSD derived kernel on an IBM RT PC with 4 megabytes of memory.

These performance improvements are especially surprising because many basic operating system overheads are actually larger in Mach than in 4.3 BSD. The use of special purpose scheduling instructions has, for example, been eliminated in the VAX version of Mach. The Mach equivalent of the Unix u-area is not at a fixed address so as to allow multiple threads of control. This increases the cost of task and thread data structure references significantly [MACH-THREADS]. In addition, VAX Mach is run as a multiprocessor system even on uniprocessors at CMU, so virtually all kernel operations have had their costs increased by locking concerns.

8. Conclusion

Mach's basic memory primitives provide applications with several mechanisms for sharing memory. As such, a mapped file interface under Mach is not required for shared memory. Mach does provide a non-shared interface for mapped files. This interface is not only appropriate for implementing various applications (e.g. shared libraries and program loading), but has increased both the performance and functionality of the system.

The internal implementation of Mach VM does not preclude shared read/write file mapping. Mach does, in fact, support the 4.2 *mmap* call for the purposes of mapping special device memory (typically used for frame buffers). The *mmap* call will also work on normal files but will not map files shared between processes. This restriction was not based on technical issues, but was an intentional modification of the *mmap* semantics. The Mach designers felt it was important to discourage programmers from writing programs which depended on sharing data which might or might not be consistently maintained in a loosely coupled environment.

9. Acknowledgements

The Mach VM implementation was done primarily by Avie Tevanian, Michael Young and David Golub. The implementation has been ported five different machine types and has yet to need modification to accommodate new architectures.

The authors would also like to acknowledge others who have contributed to the Mach kernel including Mike Accetta, Robert Baron, David Black, Jonathan Chew, Eric Cooper, Dan Julin, Glenn Marcy and Robert Sansom. Bob Beck (of Sequent) and Charlie Hill (of North American Philips) helped with the port to the Balance 21000. Fred Olivera and Jim Van Sciver (both formerly of Encore) helped with the port to the MultiMax.

References

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, Michael Young.
Mach: A New Kernel Foundation for UNIX Development.
In *Proceedings of Summer Usenix*. July, 1986.

[2] Sequent Computer Systems, Inc.
Dynix Programmer's Manual
Sequent Computer Systems, Inc., 1986.

[3] Encore Computer Corporation.
UMAX 4.2 Programmer's Reference Manual
Encore Computer Corporation, 1986.

[4] Leach, P.L., P.H. Levine, B.P. Douros, J.A. Hamilton, D.L. Nelson and B.L. Stumpf.
The Architecture of an Integrated Local Network.
IEEE Journal on Selected Areas in Communications SAC-1(5):842-857, November, 1983.

[5] Rashid, R., Tevanian, A., Young, M., Golub, D., Baron, R., Black, D., Bolosksy, W. and Chew, J.
Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures.
Technical Report , Carnegie-Mellon University, February, 1987.

[6] Satyanarayanan, M., et.al.
The ITC Distributed File System: Principles and Design.
In *Proc. 10th Symposium on Operating Systems Principles*, pages 35-50. ACM, December, 1985.

[7] Alfred Z. Spector, Dean S. Daniels, Daniel J. Duchamp, Jeffrey L. Eppinger, Randy Pausch.
Distributed Transactions for Reliable Systems.
In *Proceedings of the Tenth Symposium on Operating System Principles*, pages 127-146. ACM, December, 1985.
Also available in *Concurrency Control and Reliability in Distributed Systems*, Van Nostrand Reinhold Company, New York, and as Technical Report CMU-CS-85-117, Carnegie-Mellon University, September 1985.

[8] Young, M., Tevanian, A., Rashid, R., Golub, D., Eppinger, J., Chew, J., Bolosky, W., Black, D. and Baron, R.
The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System.
Technical Report , Carnegie-Mellon University, February, 1987.

[9] Tevanian, A., Rashid, R., Golub, D., Black, D., Cooper, E., and Young, M.
Mach Threads and the UNIX kernel: The Battle for Control.
Technical Report , Carnegie-Mellon University, April, 1987.

Table of Contents

List of Tables