An Implementation of Memory Sharing and File Mapping

Ken Shirriff

Master's Report

ABSTRACT

Due to demand for shared memory in order to implement databases efficiently, I added the `mmap` mapped file interface of 4.2 BSD Unix to Sprite, providing shared memory, file mapping, and user-level control over paging. This report first discusses the design issues for a shared memory and file mapping interface and provides an overview of the support for these functions in other operating systems. Details of the Sprite virtual memory system and the changes to support shared memory and file mapping are given. Performance measurements show that in this implementation, mapped files have comparable performance to block I/O and are faster than buffered I/O for sequential access. For random access to locally cached files, the relative performance of mapped files versus block I/O depends on the access patterns. Access to files across the network is limited by network transmission time, both for mapped files and block I/O.

January 4, 1993

† The work described here was supported by NASA and the Defense Advanced Research Projects Agency under Contract No. NAG2-591, by the National Science Foundation under Grant No. MIP-8715235, and by the IBM Fellowship program.

An Implementation of Memory Sharing and File Mapping

Ken Shirriff

Master's Report

1. Introduction

This report describes the implementation of shared memory and file mapping in the Sprite operating system. In order to port the Postgres database system to Sprite, the Postgres research group required extensions to the virtual memory capabilities of Sprite. We selected the shared memory and mapped file interface of 4.2 BSD Unix† as providing the required functionality and I added these functions to the Sprite kernel. The functions in this interface permit memory to be shared between processes and allow files to be mapped into memory. Additionally, processes can lock pages into memory, force pages to disk, or check the residency status of pages. Memory can be shared transparently among processes on a machine and can be shared across a local network if explicit consistency synchronization is used.

I made performance measurements of the mapped file interface, comparing block I/O, buffered I/O, and mapped file I/O. The measurements show that for sequential access mapped files have comparable performance to block I/O and are faster than buffered I/O. For random access the relative performance of mapped files versus block I/O depends on the access patterns. If small accesses are made to many data pages, block I/O will generally be faster. If multiple references are made to the data pages, mapped files will generally be faster.

Section 2 of this report gives an overview of the uses of shared memory and file mapping. Section 3 discusses the goals of the implementation under Sprite. Section 4

† Unix is a trademark of Bell Laboratories.

describes some previous implementations of shared memory and file mapping under other operating systems. Section 5 discusses design issues in the implementation of shared memory. Section 6 describes the functions added to Sprite to provide shared memory and file mapping. Section 7 discusses the data structures in the virtual memory system in Sprite. Section 8 gives performance measurements of file mapping and discusses my experiences with shared memory and file mapping in Sprite. Section 9 concludes this report.

2. Uses of shared memory and file mapping.

There are many applications for shared memory and file mapping. The paragraphs below discuss three common uses of shared memory: interprocess communication and synchronization, sharing of data, and database memory management.

Fast interprocess communication and synchronization are important applications of shared memory, especially on multiprocessors. By using hardware test-and-set instructions on a semaphore flag in shared memory, processes can perform fast synchronization without kernel intervention in the non-blocking case.† Shared memory can also be used for interprocess communication functions such as message passing. A message can be written into shared memory and then read by the recipient without explicitly copying the data. For instance, the lightweight remote procedure call facility of the Taos operating system [BAL89] uses a shared message buffer to transmit messages efficiently across domains on a machine.

Shared memory can make memory usage more efficient if several processes are using the same information, since only one copy of the information needs to be kept in memory. An example of this is code sharing: multiple processes executing the same

† If the process must wait on the flag, it could either busy-wait on the flag or call the kernel to wait.

program use a single copy of the program.

Another application of shared memory is database memory management. Many database management systems (DBMS) organize memory as a user-level buffer pool [Sto81]. In this scheme, data blocks are controlled by a buffer pool manager process instead of being cached by the operating system. By organizing memory in this way, the DBMS can fetch data from the buffer without the overhead of a kernel call. Another advantage of a user-level buffer pool is that the DBMS can use a buffer management strategy optimized for databases, since standard page replacement strategies usually perform poorly with databases. In addition, the database can control the order in which data is written to disk, to ensure consistency for crash recovery.

Shared virtual memory and file mapping simplify memory organization in a DBMS. Shared memory allows the database to run as multiple processes sharing a buffer pool. One process can manage the buffer pool and allocate memory for other processes using the pool. File mapping can be used as an alternative to a buffer pool. Database files can be mapped into the address space of the DBMS. This simplifies the DBMS, since the files will be read and written automatically by the operating system paging functions, instead of through explicit system calls to perform the reads and writes. One problem with file mapping is that databases often use very large files. When these files are mapped into memory, the page table may be large compared to the size of physical memory. This may require the page table to be paged, resulting in two page-ins to handle a page fault, instead of one.

File mapping has long been used to access files with the power and simplicity of accessing memory. The Unix standard I/O routines view a file as being a linear stream of data. Systems usually provide more powerful abstractions for data in memory, such as arrays, types, structures, and random access. File mapping permits data to be handled directly with these more powerful abstractions. Another advantage of file mapping is that the file pages in memory function as a user-level file cache, with paging managed by the kernel. For instance, suppose a user is making small random accesses to a large file. When regular I/O functions are used, each access requires a kernel call even if the data is in the file system cache.† If a mapped file is used, references to a data page are handled without any kernel intervention once the page is in memory.

File mapping can sometimes be used for controlling devices: a device is mapped into a process's address space and then the process controls the device directly by writing to the device in memory. This allows devices to be used without having to write device drivers in the kernel and without the cost of a kernel call. However, this only works if the device can be controlled at user level, since the memory accesses are made at user level, not at kernel level. User-level control of a device also requires a mechanism for the user program to handle interrupts, if the device is interrupt driven. One example of a device that is commonly controlled by mapping is a frame buffer. By mapping the frame buffer into memory, programs can directly write to the frame buffer without having to go through the kernel.

3. Goals of the implementation

The primary motivation for the implementation of shared memory in Sprite was the wish to run the Postgres database under Sprite. The Postgres database requires shared memory for its buffer pool manager, as described in Section 2. The Postgres group wanted shared memory functionality similar to that in 4.3 BSD or System V Unix. Only sharing on a single machine was required, not sharing across a network.

The characteristics of the Sprite operating system influenced the design of the shared memory implementation. Sprite is being developed for medium-sized networks

† With buffered I/O, references will be handled without a kernel call if the data is in the user-level I/O buffer. However, with the Unix `stdio` routines, the I/O buffer is only 4-Kbytes. Thus, random accesses to a large file will almost never reference data already in the I/O buffer. In contrast, physical user memory and the file cache may each be several megabytes.

of heterogeneous high-performance workstations [Nel88], [OCD88]. Sprite has a kernel interface and functionality similar to 4.3 BSD Unix. A single system image is provided, with files shared transparently among the machines in a Sprite cluster. Files are stored on a file server and are accessed by client machines using remote procedure calls across the network, with file caching on both clients and servers.

These characteristics motivated several goals for the implementation. Since the Sprite operating system is designed to be portable to a variety of systems, the shared memory implementation must also be portable and must isolate any machine-specific features into small machine-dependent routines. The implementation had to handle mapping files across the local network because Sprite uses a client/server structure. Another important consideration was that the addition of file mapping and shared memory shouldn't cause a performance penalty to programs not using these new features. Other design goals were that the shared memory and file mapping implementations should be efficient and reasonably compact.

There were several reasons to use a standard shared memory interface instead of designing a new one. Past experience of the Sprite group has shown that using a non-standard interface causes many difficulties since programs must be rewritten to use the new interface. As well, a familiar interface is usually simpler for programmers. In the case of shared memory, there didn't seem to be any significant benefits from designing and using a new interface.

4. Other implementations of file mapping and shared memory

File mapping and shared memory have been implemented in many operating systems. Multics is described here first since it is one of the earliest systems to provide shared memory and file mapping and it has influenced many later designs. The 4.2 BSD Unix, System V Unix, and Mach shared memory and file mapping interfaces are currently in use on many systems and are described here. Finally the sharing mechanisms of Sprite are described.

4.1. Multics

Multics is interesting historically, as it was one of the first operating systems to provide shared memory and file mapping [DaD68], [BCD72]. Two design goals of Multics were to provide direct access to all data and to permit controlled sharing of information. These goals were attained by file mapping and shared memory. File mapping is the basic method for accessing information in Multics. Processes can directly access any data in the system as if it were resident in memory.†

File mapping is transparent under Multics: there is no distinction between accessing a file and accessing memory. All on-line information in the Multics system is organized into the file system as directories of symbolically named segments [Org72]. These segments include not only normal files, but also dynamic blocks of data created by a process, such as its heap segment.‡ A process's address space is a collection of segments mapped into memory. A segment can be referenced by a symbolic generalized address, composed of the segment's path name in the file system and a symbolic offset within the segment. When the generalized address is first used, the operating system maps the appropriate segment into the process's address space and substitutes a segment number and numeric offset for the symbolic address. Sharing is achieved by having multiple processes refer to the same file, which causes the segment associated with the mapped file to be shared in memory.

† A process must have the proper permissions to access data, of course.

‡ Each process has its own temporary directory for holding segments created during execution, such as heap and stack segments.

4.2. BSD Unix

An interface for shared memory and file mapping was defined in 4.2 BSD Unix [JFL86, McK86] but was not implemented in BSD Unix until Version 4.4 (which is currently unreleased) [McK90]. This interface has been implemented in various operating systems derived from BSD Unix, such as SunOS [GMS87] and DYNIX† [Seq86]. The Sprite implementation of shared memory and file mapping is based on the 4.2 BSD design. The functions implemented in Sprite are summarized in Table 1. In the 4.2 BSD file mapping interface, a file is mapped into memory by opening the file and then calling the `mmap` function with the file's file descriptor. If the same file is mapped into two address spaces, the associated memory segment will be shared between the two address spaces. In this way, the `mmap` function provides both file mapping and shared memory. The segment can be unmapped from the address space using the `munmap` function. The 4.2 BSD interface also provides the functions `mlock`, `munlock`, and `mincore`, which lock pages into memory, unlock pages from memory, and return the residency status of pages. The 4.2 BSD interface is described in more detail in Section 6.

In contrast to the interface designed for 4.2 BSD Unix, the actual implementations of 4.2 and 4.3 BSD Unix have limited support for sharing memory and no support for mapped files. The implementation permits sharing of read-only code, but not heap or stack memory. This read-only sharing of code improves performance when several processes are using the same code, but doesn't provide any read-write sharing of data. This sharing is invoked automatically through the routines used to create new processes. The `execve` function replaces the current process with a new process, running the specified code image. If any other process is running the same code image, the two processes will share code in a read-only fashion. The `fork` function creates a new process which shares the code segment of the parent process.

---

† DYNIX is a trademark of Sequent Computer Systems, Inc.

`mmap(address, length, protection,`
`   sharing, file_descriptor, offset);`
` caddr_t address;`
` int length, protection, sharing;`
` int file_descriptor;`
` off_t pos;`
	Maps a file into memory, sharing memory if multiple processes map the file.

`munmap(address, length);`
` caddr_t address;`
` int length;`
	Unmaps part or all of a mapping from the process's address space.

`msync(address, length);`
` caddr_t address;`
` int length;`
	Forces mapped pages of memory to disk.

`mlock(address, length);`
` caddr_t address;`
` int length;`
	Locks pages into memory.

`munlock(address, length)`
` caddr_t address;`
` int length;`
	Unlocks pages from memory.

`mincore(address, length,`
`   return_vector);`
` caddr_t address;`
` int length, return_vector[];`
	Returns status (invalid, absent, clean, dirty) of specified pages.

Table 1: 4.2 BSD Unix virtual memory functions added to Sprite.

4.3. System V Unix

System V Unix has support for shared memory but not for file mapping [ATT86]. The System V shared memory interface is summarized in Table 2. This interface has also been implemented in other Unix-based operating systems such as SunOS. The Postgres group has implemented a simplified version of the System V interface at user level on top of Sprite's BSD `mmap` interface. Shared segments in System V are identified by a systemwide integer identification key. To use a shared memory segment, a user process invokes the `shmget` system call, passing it the size of the segment and the identification key. This system call creates a shared segment if none already exists for the given key and returns an integer shared memory identifier. The shared memory identifier is used to identify the shared segment in subsequent operations. The segment can be mapped into the address space with `shmat` and unmapped from the address space with `shmdt`. The segment can be locked into memory, unlocked from memory, or destroyed with `shmctl`.

`shmget(key, size, flag)`	Creates shared segment from a key.
`shmat(id, address, flag)`	Maps shared segment into process's address space.
`shmdt(address)`	Removes shared segment from process's address space.
`shmctl(id, command, buffer)`	Performs various operations on a shared segment.

Table 2: Shared memory functions in System V Unix.

4.4. Mach

The Mach operating system has extensive, flexible support for shared memory and file mapping [TRY87]. One method of sharing is the concept of a thread, which is a lightweight process that shares its entire address space with other processes. Another method of sharing is inheritance, which allows a process to share regions of memory with its child processes. Mach permits user-defined network sharing semantics through the capabilities of an external memory pager. File mapping is provided by the Mach function `map_fd`, which is similar to `mmap`, but doesn't provide sharing.

Mach provides flexible control of the virtual memory system through the abstractions of memory objects and pagers [RTY88]. A memory object is an abstract block of data which can be mapped into a process's address space. The data can be provided by a default paging routine or by an external user-level memory pager. A pager is a collection of routines to handle page faults and to store paged-out pages. The kernel communicates with the designated pager when paging is required. Since the pager can generate the data to fill memory in an arbitrary way, it has tremendous flexibility and control over the virtual memory system. For instance, mapped files are implemented by a memory pager which reads and writes data from a file. Consistent shared memory across a network can be implemented at user level by a pager that provides the same memory object to several different machines and makes sure the data is consistent among the machines [YTR87]. A different shared memory consistency policy can be obtained by modifying the pager.

4.5. Sprite

Prior to the addition of the 4.2 BSD `mmap` interface, Sprite had two limited sharing capabilities. Sprite permitted processes to share code segments, similar to Unix. In addition, Sprite processes could share both their code and heap segments, while having separate stacks [Nel86]. This is similar to the Mach model of multiple threads of execution in a single address space. The motivation for heap sharing was to provide fast and efficient interprocess communication for Sprite applications running on the SPUR multiprocessor. Code and heap sharing is provided by the `vfork` function, which starts a new process sharing its parent's code and heap, but with a separate stack. The implementation of the Sprite virtual memory system is summarized in Section 7 of this report.

5. Design issues

There are several design issues for a shared memory and file mapping interface. This section discusses various design issues and the ways these issues have been resolved by different operating systems.

5.1. Name space for shared memory

Processes need some way to refer to shared memory segments so that different processes can inform the operating system that they are to share the same segment. Many of the Mach sharing mechanisms and the original Sprite heap sharing mechanism depend on inheritance, where a process must create subprocesses which can share memory with it. A more general method is to have a name space for the segments and allow processes to select arbitrary names for segments they are to share. This works as long as cooperating processes can pick the same name for a segment to be shared and disjoint processes don't conflict with each other by picking the same name for unrelated segments. Multics, BSD Unix, Mach, and Sprite all use the file system name space to indicate shared segments, while System V Unix uses 4-byte integer keys as the name space. Because there is no structure on the System V name space, unlike the file system's directory structure, unrelated System V processes could easily pick conflicting keys for shared segments. The recommended System V solution is to generate keys using the library routine `ftok`, which takes a file system path and a single character identifier and returns a unique integer identification key.

Access to a segment can be made more efficient by having the operating system translate the user-selected segment name into an internal segment identifier, which is then used for future operations. In BSD Unix and Sprite, the segment name is translated to an integer file descriptor by the standard file system `open` command, and the file descriptor is used in any functions on the shared segment. In System V, the integer segment key is converted into a shared memory identifier by the `shmget` function and the shared memory identifier is used for further operations on the segment.

One advantage of using the file system as a name space is that shared memory segments can be examined using normal file system functions. For instance, under BSD Unix, `ls` will show what segments exist in the file system and `rm` can be used to remove unwanted segments. System V has no method to determine what keys are in use except examining the kernel's virtual memory (`/dev/kmem`). Two user-level programs are provided to do this: `ipcs` displays a list of keys in use and `ipcrm` removes a shared memory segment, given its key.

5.2. Persistence of segments

The lifetime of a shared memory segment is another design decision. Once a shared segment has been created, it could either be automatically removed at some point or persist in the system indefinitely. Automatically removing shared segments prevents unwanted data from lingering around in the system. However it also prevents processes from being able to create segments that will last across process lifespans or machine crashes. In Multics, BSD Unix, and Sprite, shared segments reside in the file system and thus will exist until they are explicitly removed. Since System V Unix manages shared segments in kernel memory, the segments will be lost if the host machine is rebooted. In normal operation, though, System V segments will persist until they are explicitly removed.

One problem with storing shared segments in the file system is that in many cases the applications share the data in memory but don't require the data to be saved to disk, so writing it to a file is wasted effort. The 4.3 BSD Unix interface specification has a solution to this: a flag can be given to `mmap` to create an anonymous segment, which resides only in memory and is not associated with a file. Since Mach allows user-level pagers, Mach applications can control shared segments and store them in the file system, or not, as desired.

5.3. Sharing in a network

A network environment adds several complications to shared memory. The main issue is whether (and, if so, how) to maintain consistency when data is shared across two or more hosts. This is related to the cache consistency problem; the main memory of a host can be considered to hold a cached copy of the shared data and some semantics for consistency must be provided. One possibility is to limit sharing to processes on the same machine. This eliminates the consistency problems of multiple machines, but at the cost of limiting the functionality provided.

Another alternative is to implement sharing transparently across the network. This would permit processes on different machines to share a memory segment as if the processes were on a single machine. Transparent sharing can be very costly since every write to a shared segment must be propagated across the network to any other users of the segment. Consider the case of one machine writing to a shared page while another machine reads the shared page. The reading machine must continually obtain the modified data from the writing machine if the page is to be shared transparently.

A third alternative, limited consistency, was selected for implementation in Sprite. In this model, sharing can occur across the network but changes to a shared page on one processor will not be visible to processes on other processors until the modified page is returned to the server by `msync` or by being paged out. At this point, the changes will be propagated to any other processors sharing the page. This does not provide transparent sharing, since changes to segments are not visible immediately, but it gives the user the capability to implement transparent sharing or other desired consistency semantics on top of the supplied functionality. A similar method of supplying consistency is provided in SunOS. In Mach, the user-supplied pager routine can implement whatever consistency model is desired.

Process migration poses a potential problem for shared memory. The Sprite operating system permits processes to be migrated from one processor to another across the local network. This raises the possibility of multiple processes sharing memory on a single processor and then having some of the processes migrate to a different processor while others stay behind. If this happened and sharing across the network is not transparent, the sharing semantics would change and the application might no longer work correctly. To avoid this, processes using shared memory in Sprite are marked as not migratable, so they will remain on their original machine. An alternative would be to migrate all processes sharing memory on a machine together, so the sharing would still be local.

5.4. Assigning a virtual address range

One last issue is where in a process's address space to place a segment for shared memory or file mapping. In a segmented architecture such as used by Multics, each memory segment is assigned a separate segment number and thus resides in a separate region of the address space. However, in a flat address space, a shared segment must share the address space with other segments without conflicting with them. For instance, suppose a shared memory segment is mapped after the end of the heap. This would complicate memory allocation functions such as `malloc`, since there will be holes in the usable heap as the heap expands over the shared segment.

Many shared memory interfaces allow the user to supply an address for a segment, but may ignore this address. Mach permits a virtual memory object to be created either at a user-specified address or at a free address selected by the kernel. 4.3 BSD Unix and System V permits the user to provide a page-aligned starting address when creating a shared memory segment, but the system is free to use this address as a suggestion or ignore it totally, depending on the implementation.

Sprite currently reserves part of the address space above the stack for shared memory segments in order to simplify memory management. This eliminates any problems with shared segments interfering with the heap or stack. The main problem with this method is that a large region (currently 32 Megabytes) of the address space must be reserved. Reserving this region both limits the total size of a process's shared memory segments and prevents anything else from using the reserved address space. If the fixed sharing region proves to be insufficient, I may modify shared memory allocation to allocate arbitrarily large segments† between the heap and the stack.

6. Functions added to Sprite

The functions added to the virtual memory system of Sprite are `mmap`, `munmap`, `msync`, `mlock`, `munlock`, and `mincore`. These functions are summarized in Table 1, and are are basically the same as the corresponding functions in 4.3 BSD.

The function `mmap(address, length, protection, sharing, file_descriptor, offset)` is used to provide file mapping and shared memory.

† The size of shared memory or mapped file segments will be limited by the size of the processor's virtual address space, in any case.

It maps the specified file into the specified region of the process's address space. If two or more processes map the same file, the processes will share the mapped region. The file to be mapped is specified by `file_descriptor` and an offset into the file is given by `offset`. The region of memory is specified by `address` and `length`, and must be page aligned. In the current implementation, the value of `address` is ignored. The `mmap` function returns the address in the process's virtual memory at which the segment is mapped. The access mode for the segment (read/write) is given by `flags`. If the file is mapped with write access, the file functions as a swap file for the shared region.

The function `munmap(address, length)` is used to remove part or all of a mapping from the process's virtual address space. The values of `address` and `length` specify the region to be unmapped and must be page aligned. Any modified pages that are no longer mapped by any process will be written back to the file. The `munmap` function can result in a mapped region being shrunk, split into two parts, or entirely removed.

The `msync`, `mlock`, and `munlock` functions can be used to force the desired consistency between the data in memory and the data on disk. They provide the capability, required by some databases, of ensuring pages are written to disk in the right order and at the right time. The function `msync(address, length)` forces the pages in the specified address range to disk and returns after the pages have been written. The function `mlock(address, length)` locks the specified pages into memory, so they cannot be paged out to disk. The function `munlock(address, length)` unlocks the specified pages from memory. If `mlock` isn't used, data will be stored in memory and will be written back when the segment is no longer in used. Additionally, data will be written to disk if the pages are paged out by the operating system. By using `mlock`, the specified pages will be locked down into memory and will not be paged out. Thus a locked page will not be written to disk until it is forced to disk by `msync` or released by `munlock`.

The function `mincore(address, length, return_vector)` returns in `return_vector[]` a list of integers, one for each page, indicating the status of the pages in the specified address range. The return value indicates if the page is clean, dirty, paged out, or invalid.

7. The virtual memory system in Sprite

Section 7.1 discusses the data structures in the virtual memory system in Sprite prior to the addition of shared memory and file mapping. A higher level description of the original Sprite virtual memory system is given in [Nel86]. Section 7.2 describes the data structures used to provided shared memory and file mapping. Section 7.3 describes the machine-dependent aspects of shared memory.

7.1. Original virtual memory system

A process's virtual memory in Sprite consists of four segments, where a segment is a contiguous group of pages sharing the same swap file. These segments are an operating system abstraction on top of the machine's address space, as opposed to being part of the machine architecture as segments are in Multics. There are four types of segments used in Sprite, each with different properties: code, heap, stack, and system. A code segment is read-only and of fixed size. A heap segment is readable and writable, and can be expanded or contracted through a system call in order to provide memory allocation functions. Code and heap segments can be shared among processes. A stack segment automatically grows downwards as references are made outside the allocated part of the stack. The system segment is locked into memory and is used only by the kernel; it is unused by user-level processes.

There are four key data structures in the virtual memory system: the processes' virtual memory data blocks, the segment table of segment control blocks, the operating system page tables, and the core map. The first two data structures influenced the design of shared memory. The last two data structures were unaffected by the addition of shared memory and are described in this section for completeness. Figure 1 illustrates the data structures. These data structures are machine-independent operating system structures. Machine-dependent structures, such as the machine's page table or translation buffers, are managed separately by machine-dependent subroutines.

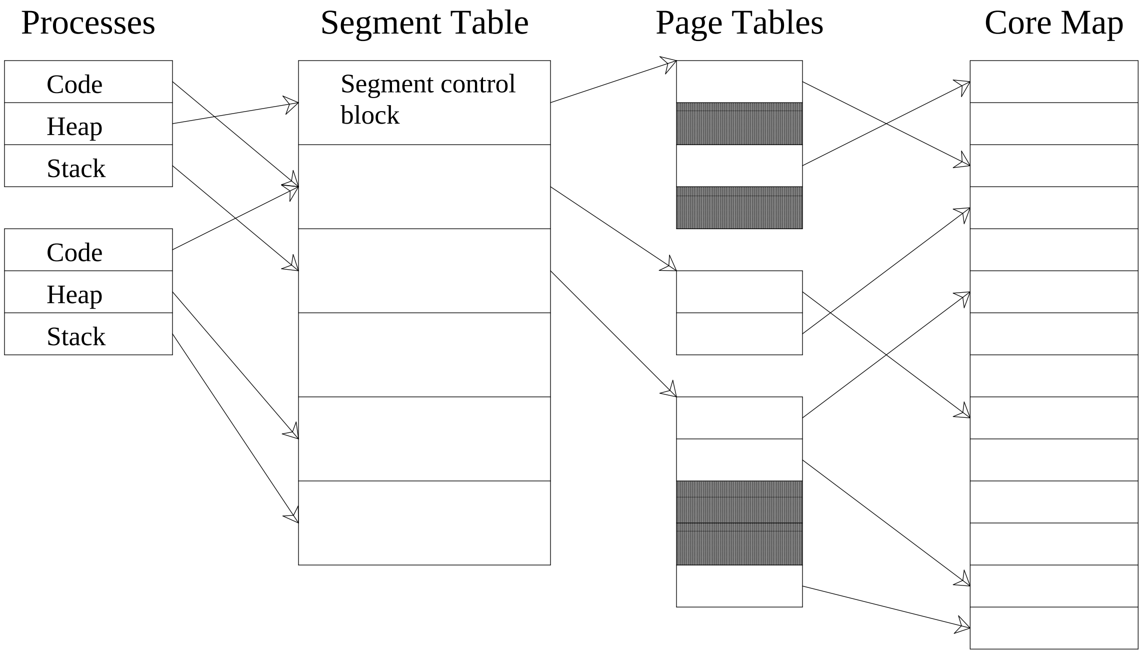

Figure 1: Virtual memory data structures. Each process has pointers to segment control blocks for the code, heap, stack, and system segments. (The system segment is not shown since it is only used by kernel processes.) Each segment control block holds information on the segment and points to the page table for the segment. Each physical page in memory has an entry in the core map. Core map entries are referenced by the page table. Paged-out pages are marked in the page table as being not physically resident (indicated here by gray). The two processes in this figure are sharing a code segment.

Each process has a virtual memory data block as part of the state in its process control block. This data block has four (possibly null) pointers to segment control blocks for the code, heap, stack, and system segments. The data block also contains status flags and any process-specific machine-dependent data structures. Since the data block limits a process to four specific segments, I had to change it to handle additional shared segments.

The segment table consists of a fixed number of segment control blocks. There is one segment control block for each potential segment in the system, with a segment control block holding the state necessary to manage the associated segment. This state includes the virtual address range of the segment, a pointer to the segment's page table, a list of processes using the segment, a reference count, a stream pointer for a swap file, type information, segment size information, copy-on-reference and copy-on-write information, and status flags. Unused segment control blocks are kept in a free list and new segments are allocated from this list.

Each segment has an associated page table with an entry for each page in the segment. Each page table entry has status flags and, for physically resident pages, an index into the core map. There is one core map entry for each physical page of memory. The core map entries are organized into four linked lists, holding the allocated, free, dirty, and reserved pages. Each core map entry also holds the page's status flags, reference time, lock count, and associated virtual address.

These four data structures are used to handle page faults. First, the faulting virtual address is converted into a translated address, consisting of the virtual address and a pointer to the appropriate segment control block. The translated address is obtained by comparing the virtual address to the addresses of the process's four segments to determine in which, if any, segment the address falls. Next, a new physical page is obtained from the core map's free page list. Finally, the page is loaded using the translated address and the address information in the segment table.

7.2. Data structures for shared memory and file mapping

Two data structures were added to the virtual memory system to support file mapping and shared memory. The process segment list holds data on sharing associated with a process and the shared-segment control blocks hold data associated with a shared segment. There are two main functions of these data structures: handling page faults and allowing mapping operations on segments. Figure 2 shows the relationship of these data structures with the original virtual memory data structures.

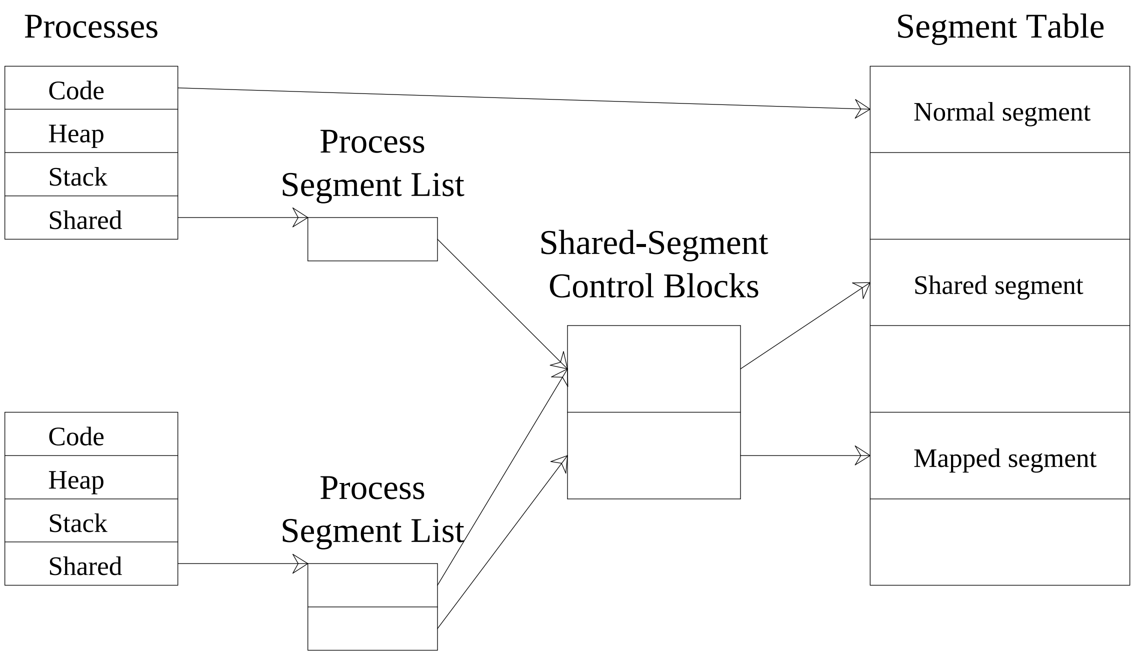

Figure 2: Shared memory data structures. Each process has a process segment list, which lists the process's shared segments and information on the mapping for each segment. A shared-segment control block is associated with each shared segment and holds information on the shared segment and points to the associated segment table entry. The segment table entries reference the page tables and the core map as in Figure 1. (These references are not shown here.) The two processes in this figure are sharing a segment. The second process has an additional shared segment, which is being used for file mapping.

Each process has a pointer to a list of all the shared or mapped segments associated with the process. This list is called the process segment list. Each process segment list entry holds the information associated with one process-segment mapping: the file descriptor of the associated file, the segment table entry, the address range mapped, and the protections on the segment.

The second new data structure is the linked list of shared-segment control blocks. Each shared segment has a control block to hold segment-specific information: the associated file, a reference count of the number of mappings of the segment, and a pointer to the appropriate segment control block. This information could alternatively have been added to the segment table instead of being stored separately.

The main use of the process segment list is handling page-ins of shared or mapped files. When a page is paged-in, the process segment list is used to determine if the faulting virtual address lies in a shared segment, and if so, which segment. The address translation routine checks the address against the system, code, heap, and stack segments, and the process's list of shared segments.

There are several alternatives to the process segment list that could have been used to map addresses to segments. One alternative would be to use a single structure to hold the code, heap, and stack segments and the shared segments, instead of storing shared segments separately. This was not done, in order to minimize changes to the previous Sprite implementation. However, it would probably result in a cleaner implementation since all segment types would be treated uniformly. Another alternative is a tree-structured address map such as is used in Accent [FiR86]. This structure provides compact storage of a complex, sparse address space and allows fast indexing into the appropriate address map. However, it has a high implementation cost [RTY88].

When a segment is mapped into memory by `mmap`, the list of shared-segment control blocks is checked to see if the specified file is already mapped. If so, a pointer to the control block is added to the process's segment list. Otherwise a new segment is created and a new segment control block is added to the segment table. A new shared-segment control block is created for the new segment and a new entry is added to the process's shared segment list, pointing to the shared-segment control block.

When a segment is unmapped by `munmap` or by closing the associated file, the process segment list is updated. If the segment is partially unmapped, the mapped address range in the process segment list entry is updated. If the entire region is unmapped, the process segment list entry is removed and the segment's reference count is checked. If there are no more references to the shared segment, the segment is removed, along with its segment table entry and shared-segment control block.

The file system contains a new data structure, the client list, to handle synchronization of shared data across the network. The file server keeps a list of clients using each shared segment or mapped file, When a client flushes pages to disk, the server notifies all clients using the file that the modified pages are invalid. These pages are then invalidated in the clients' virtual memories. Any accesses to these pages will cause the new data to be paged in from the server, obtaining the new data. The cost of this scheme is one message to each machine sharing a file each time the file is explicitly synchronized with `msync`, and a read of the data from the server every time a client accesses out-of-date data. As discussed in section 5.3, this scheme doesn't provide transparent sharing, but allows the user to implement a desired consistency policy.

7.3. Hardware dependencies

Since the Sprite operating system is designed to run on a variety of machine types, the code is organized to be easily portable to different machines. The operating system source code is organized into modules, with each module having subdirectories to hold the machine-dependent subroutines and definitions for each machine type. In this way, most of the code can be shared across machine types, with only a small amount that must be changed for a new machine type. Table 3 shows the machine-dependent routines used for shared memory and file mapping. The routine `VmMach_SharedProcStart` is called when a process first makes use of shared memory or file mapping. When a process is finished with all its shared segments, `VmMach_SharedProcFinish` is called. In the current implementation, these routines are used to create and destroy data structures for keeping track of what parts of the process's address space are available for shared segments. When a shared segment is mapped, `VmMach_SharedStartAddr` is called to obtain a suitable address range for the shared segment. This routine takes care of any constraints a machine may have on the placement of shared segments. In the current implementation, this allocates the segment from the region of the process's address space reserved for shared memory. When a shared segment is destroyed, `VmMach_SharedSegFinish` is called. In the current implementation, this frees the segment from the reserved address range.

`VmMach_SharedProcStart`	Starts a process's shared memory usage.
`VmMach_SharedProcFinish`	Completes a process's shared memory usage.
`VmMach_SharedStartAddr`	Locates an address and starts a shared segment.
`VmMach_SharedSegFinish`	Finishes usage of a shared segment.

Table 3: Machine dependent functions for the Sprite implementation.

Shared memory is made more complicated by the presence of virtually addressed caches on some machines. Implementation of shared memory was straightforward on the Sun-3 and DECstation 3100, which have physically addressable caches, but was more difficult on the Sun-4, which has a direct-mapped 128-Kbyte virtually addressable cache. Because the Sun-4 cache is virtually addressable, cache aliasing can potentially occur.† One solution to this problem would be to disable caching on shared segments. However, this would decrease performance, since all references to shared memory would have to go to memory. To prevent aliasing without a performance loss, all shared memory regions on the Sun-4 are allocated on an alignment of 128K. Thus, since two different virtual addresses for the same byte of shared memory will differ by 128K, they will map to the same location in the cache, and aliasing won't occur.

---

† Aliasing occurs if one physical memory address has two different virtual addresses which map to different locations in the cache. A write to one of these virtual address will update one of the cache entries, but will leave the other cache entry out-of-date. However, if the two virtual addresses are congruent modulo the cache size, both virtual addresses will map to the same location in the cache. Then, since there is only one cache entry, it will be updated correctly, and aliasing won't occur.

8. Experience with the implementation

The shared memory and file mapping functions are now operational and are being used by the Postgres group. The implementation took about 1200 lines of code, including comments and debugging code. Section 8.1 gives performance measurements of file mapping, comparing it with other I/O techniques. A detailed description of the measurements is given in Appendix 1. Section 8.2 comments on the implementation and gives suggestions for changes to the implementation.

8.1. Performance measurements

This section gives measurements of the performance of mapped files and other I/O techniques. In Section 2 some of the performance tradeoffs of using mapped files instead of other I/O functions were discussed. The measurements in this section clarify the effects of these tradeoffs. To summarize these measurements, in the Sprite implementation mapped files and block I/O provide the same performance for sequentially reading a file, but mapped files are much faster than buffered I/O. For random file I/O, the relative performance of mapped files and block I/O depends on the access patterns.

Table 4 gives performance measurements for sequentially reading a 480-Kbyte file using various methods. The first three measurements give the time to sequentially read the file using buffered I/O, block I/O, and file mapping. In order to explain the performance results, measurements are also given for paging in data pages, copying data, and transmitting data via remote procedure calls.† The measurements show that all the methods which read data in blocks (read, mmap, and data fill) have about the same performance. Buffered I/O (getc) is significantly slower, since each byte must be handled individually. All the I/O functions are an order of magnitude faster when the data is

† For file mapping and paging in data pages, one byte in each page is touched to cause the data to be paged in.

locally cached than when it must be obtained from the server. For locally cached data, most of the time is spent copying the data from the cache.† For remotely cached data, nearly all of the time is to receive the data across the network with remote procedure calls.

Measurement	Locally cached (sec)	Locally cached (Mbytes/sec)	Remotely cached (sec)	Remotely cached (Mbytes/sec)
getc	.54	.91	2.0	.25
read	.13	3.78	1.4	.35
mmap	.13	3.78	1.1	.45
data fill	.13	3.78	1.2	.41
bcopy	.08	6.14		
RPC			1.1	.45

Table 4: Performance measurements for sequential access to a 480-Kbyte file on a DECstation 3100 with a Sun-4 file server. Times are given to read the file sequentially using buffered I/O (getc), block I/O (read), and file mapping (mmap). Times are also given to page in a 480-Kbyte array with initialized data (data fill), to copy a resident array (bcopy), and to transmit 480-Kbytes by remote procedure call (RPC). The accuracy of the remotely cached measurements is not sufficient to indicate any performance difference except betweeen block I/O (getc) and the others.

In [TRY87] it is suggested that mapped files can reduce the cost of file I/O since mapped files don't need to copy data from the kernel to user level. However, in the present Sprite implementation, this benefit of mapped files will only occur for remote file accesses, for reasons described below. When data is read with file I/O functions, the requested data is copied from the local file cache to user level. If the data is not in the file cache, the appropriate page will first be obtained from the file server and stored in the file cache. To page in data, the page is copied from the local cache if possible.‡ If the

† Note that in applications such as databases, mapped file data will normally be kept in user memory once accessed, and will not enter the local file cache. For these applications, it may not be relevant to consider the case of mapping locally cached data. In other cases, mapped file data could be in the local cache initially: for instance, if one program accesses data using file I/O and then another program accesses the same data using file mapping.

‡ This copy operation for paging in local mapped file data could be eliminated by modifying Sprite file caching to permit the cache block to be mapped into the process's address space directly. In the current implementation, the data is explicitly copied from the cache into a memory page because the Sprite file system and virtual memory system manage cache blocks and memory pages separately. Mapping cache blocks into a user's address space would require extensive changes to the interface between the file cache and the virtual memory system. One problem is

data is not locally cached, it is obtained from the file server and is stored into the appropriate memory page, bypassing the local file cache. The required operations are summarized in Table 5. In conclusion, local data accesses require one data copy for reading or mapping. Remote data accesses require one more copy operation for reading than for mapping, and the overhead of a bcopy operation in the total time is about 7%, from the measurements in Table 4.

Source	File I/O read	Mapped file access
Local cache	1 copy	1 copy
Local disk	1 read, 1 copy	1 read, 1 copy
Remote cache	1 rpc, 1 copy	1 rpc
Remote disk	1 read, 1 rpc, 1 copy	1 read, 1 rpc

Table 5: Operations required to read data with block I/O and with file mapping. The "read" operation reads the data from disk. The "rpc" transmits the data across the network via remote procedure call. The "copy" copies the data from the local file cache.

Table 6 gives measurements for randomly accessing individual bytes in a 480-Kbyte file. There are two important differences between these measurements and the measurements in Table 4. First, throughput is dramatically reduced, since each random read obtained only one wanted byte, while each sequential read obtained a 4-Kbyte block. Second, the relative performance of block I/O and of file mapping depends heavily on the data access patterns.

There are three important factors affecting the times for randomly reading and mapping data. First, when data is read from the cache, only the requested data (in this case one byte) is copied from the cache. However, when a mapped file is accessed, the entire 4-Kbyte page is paged in, even if only one byte is used. Second, once a mapped page is loaded, any later accesses to data in that page can be handled immediately. The read

---

that the file system and virtual memory system would have to cooperate in performing operations such as locking, unlocking, flushing, and freeing cache blocks that are mapped into memory. Another problem is that file cache blocks and virtual memory blocks are different sizes on some Sprite machines. I tested an experimental implementation of mapped files that directly mapped cache blocks into memory and found that the performance for locally cached files was about a factor of 10 better than the mmap measurements in Table 4.

Measurement	Locally cached (sec)	(Kbytes/sec)	(ms/probe)	Remotely cached (sec)	(Kbytes/sec)	(ms/probe)
read, 120 probes	.05	2.4	.42	.7	.17	5.8
read, 1000 probes	.40	2.5	.40	1.3	.77	1.3
mmap, 120 probes	.09	1.3	.75	.7	.17	5.8
mmap, 1000 probes	.13	7.7	.13	1.2	.83	1.2

Table 6: Performance measurements for random acceses to a 480-Kbyte file. Times are given to probe the file by reading single random bytes using block I/O (read) and file mapping (mmap). Tests were performed reading 120 bytes in total (from 74 pages) and 1000 bytes (from 120 pages). For block I/O, a system call is made for each probe. For a mapped file, a page-in results for each new page accessed. File mapping is more efficient than block I/O in the 1000 probe case because each page receives many probes.

operation must perform a system call for every read. Finally, for a remotely cached file, the time to receive data pages from the file server dominates the time to access these pages in the local cache later.

These three factors account for the measurements in Table 6. When data is read from the local file cache, the time spent is proportional to the number of reads performed and the amount of data read. On the other hand, when a mapped file is read, the time spent is proportional to the number of pages accessed. Thus, when small amounts of data are accessed from many pages, reading the data with block I/O functions is faster than mapping the file. However, if many references are made to the same page, mapping the file will be faster. For a remote file, these distinctions are obscured by the time required to obtain the data across the network. In this case, the time spent is proportional to the number of remote pages received.

In conclusion, in the Sprite implementation, mapped files and block I/O have similar performance. The data rates for sequential access are nearly identical. However, in some situations either mapped files or block I/O may be better. Mapped files have the advantage of having pages directly available to the user once they've been loaded. However, this must be balanced against the overhead of transferring data in page-sized blocks with mapped files.

8.2. Comments on the implementation

One difficulty in implementing shared memory and file mapping in Sprite was that the file system and the virtual memory system are heavily interrelated. Because files can be cached on both the client and the server, careful changes to the file system code were required to ensure that the right copy of a mapped file would be paged in. The problems arose because Sprite uses separate data paths to access swap files and regular files: swap files are cached only on the server, while regular files are cached both on the client and on the server.† The Sprite implementation treats mapped files and shared memory backing files as swap files.

There are many improvements that can still be made to the Sprite virtual memory system. One improvement would be to provide more support for sharing of data across a network. This could consist of a library of routines to provide transparent sharing and other useful consistency strategies. These routines would use the `mmap`, `mlock`, and `msync` functions to implement the desired sharing semantics.

Another useful change would be support for large, sparse virtual address spaces, in which the validated part of the address space is much larger than the part actually in use. In Sprite, page table data structures will be created for the entire validated region. In contrast, Mach uses an address map [RTY88], which compactly describes the mapping of a contiguous virtual address range. The address map representation is much more compact than the page table representation for sparse address ranges.

Other improvements could be made to the Sprite virtual memory system to make database support more efficient. Some suggestions are given in [Sto81] and [Sel89]. One improvement would be to provide asynchronous I/O for paging in data. This would allow the data manager to request a page and then continue processing while the page is

† Swap files are not locally cached because paging out from virtual memory to a local cache would not be useful since it wouldn't free any physical memory pages.

being read in from disk instead of blocking when a page fault occurs. Another suggestion is to modify system calls so fewer system calls are required. Possible changes are to extend `mmap` to map multiple regions with one system call and to permit `mmap` to lock pages as they are read in, instead of requiring a separate `mlock` call.

The virtual memory interface of Mach is much more flexible than that provided by Unix or Sprite. Mach gives the user more support over the operating system as well as providing additional built-in functionality. For these reasons, there are currently plans to install the virtual memory subsystem of Mach in Sprite.

9. Conclusions

The 4.3 BSD Unix shared memory and file mapping functions have been successfully added to Sprite and are being used by the Postgres database system. These functions provide mapped files, sharing of data, and control over writing data to disk. Sprite is currently running with shared memory and file mapping on the Sun-3, Sun-4, and DECstation 3100.

Measurements show that, in the Sprite implementation, file mapping has comparable performance to block I/O for sequential file access and is much faster than buffered I/O. For random access the relative performance of mapped files versus block I/O depends on the access patterns. For locally cached files, the time for block I/O is proportional to the number of reads and the amount of data read. In contrast, the time for mapped file access is proportional to the number of distinct pages accessed. Thus, if small accesses are made to many data pages, block I/O will generally be faster. If multiple references are made to data pages, mapped files will generally be faster. When the data is not locally cached, the performance of mapped files and block I/O is worse by an order of magnitude. The time spent accessing remote data is proportional to the number of distinct pages accessed, both for mapped files and block I/O. Most of this time is spent transmitting the data pages across the network.

10. Acknowledgements

I am very much indebted to my research advisor, John Ousterhout, for his guidance both in undertaking this research and in writing this paper. I would like to thank Michael Stonebraker, my second reader, for his comments. Finally I would like to thank the members of the Sprite group, with particular thanks to Brent Welch for his patience in explaining the file system.

11. Bibliography

[ATT86] *System V Interface Definition*, AT&T, Indianapolis, IN, 1986.

[BCD72] A. Bensoussan, C. Clingen and R. Daley, The Multics Virtual Memory: Concepts and Design, *Communications of the ACM 15*, 5 (May 1972), 308-318.

[BAL89] B. Bershad, T. Anderson, E. Lazowska and H. Levy, Lightweight Remote Procedure Call, *Proc. Twelfth ACM Symposium on Operating System Principles, Operating Systems Review 23*, 5 (Dec. 1989), 102-113.

[DaD68] R. Daley and J. Dennis, Virtual Memory, Processes, and Sharing in MULTICS, *Communications of the ACM 11*, 5 (May 1968), 306-312.

[FiR86] R. Fitzgerald and R. Rashid, The Integration of Virtual Memory Management and Interprocess Communication in Accent, *ACM Transactions on Computer Systems 4*, 2 (May 1986), 147-177.

[GMS87] R. Gingell, J. Moran and W. Shannon, Virtual Memory Architecture in SunOS, *Proceedings of Summer USENIX*, June 1987, 81-94.

[JFL86] W. Joy, R. Fabry, S. Leffler, M. McKusick and M. Karels, Berkeley Software Architecture Manual 4.3BSD Edition, Computer Systems Research Group, University of California, Berkeley, 1986.

[Li86] K. Li and P. Hudak, Memory Coherence in Shared Virtual Memory Systems, *Proceedings 5th ACM SIGACT News-SIGOPS Symposium of Principles of Distributed Computing*, Canada, Aug. 1986.

[McK86] M. McKusick and M. Karels, A New Virtual Memory Implementation for Berkeley UNIX, *Proceedings of the European UNIX Users Group Meeting*, Manchester, England, Sep. 1986, 451-460.

[McK90] K. McKusick, (Personal communication), Apr. 1990.

[Nel86] M. Nelson, Virtual Memory for the Sprite Operating System, Technical Report UCB/CSD 86/301, University of California, Berkeley, June 1986.

[Nel88] M. Nelson, Physical Memory Management in a Network Operating System, PhD Thesis, Technical Report UCB/CSD 88/471, University of California, Berkeley, Dec. 1988.

[Org72] E. Organick, *The Multics System: An Examination of its Structure*, The MIT Press, Cambridge, MA, 1972.

[OCD88] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson and B. Welch, The Sprite Network Operating System, *Computer,* Feb. 1988, 23-36.

[Ous89] J. Ousterhout, Why Aren't Operating Systems Getting Faster as Fast As Hardware?, WRL Technical Report TN-11, Digital Western Research Laboratory, Palo Alto, California, Oct. 1989.

[RTY88] R. Rashid, A. Tevanian, Jr., M. Young, D. Golub, R. Baron, D. Black, W. Bolosky and J. Chew, Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures, *IEEE Transactions on Computers 37,* 8 (Aug. 1988), 896-908.

[Sel89] M. Seltzer, Transaction Processing Under Unix, CS 286 class project report, University of California, Berkeley, May 1989.

[Seq86] *DYNIX® Programmer's Manual, Volume 1*, Sequent Computer Systems, Inc., 1986.

[Sto81] M. Stonebraker, Operating System Support for Database Management, *Communications of the ACM 24*, 7 (July 1981), 412-418.

[TRY87] A. Tevanian, Jr., R. Rashid, M. Young, D. Golub, M. Thompson, W. Bolosky and R. Sanzi, A Unix Interface for Shared Memory and Memory Mapped Files Under Mach, *Proceedings of Summer USENIX*, June 1987, 53-67.

[YTR87] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black and R. Baron, The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System, *Proc. Eleventh ACM Symposium on Operating Systems, Operating Systems Review 21*, 5 (Nov. 1987), 63-76.

Appendix 1: Details of the performance measurements

This section describes the measurements in Tables 4 and 6 in detail. I performed these measurements on a DECstation 3100 client served by a Sun 4 file server. The page size and file block size are 4-Kbytes, and the file is 480-Kbytes (491520 bytes), so the file is 120 pages long. Table 4 gives the time in seconds to sequentially read, copy, or transmit 480-Kbytes of data. Table 6 gives the time to individually access 120 or 1000 random bytes in a 480-Kbyte file. The resulting throughput is given in megabytes (10^6 bytes) per second in Table 4 and kilobytes (10^3 bytes) per second in Table 6. For the "locally cached" measurements, the file being read is in the client machine's file cache. For the remotely cached" measurements, the file is in the file server's cache, but not the client's file cache. The times given are elapsed time, returned by the `time` function. The measurements were done on a lightly loaded network. The numbers in Table 4 and 6 are averages over 10 measurements. The measurements for locally cached data fluctuated around the average by about 4%. The measurements for remotely cached data fluctuated by about 30% due to variations in network traffic and server load.

The measurements in Table 4 are of sequential accesses to 480-Kbytes of data. The measurement "getc" is the time to read the file a byte at a time, using the `getc` buffered I/O function. For "read", the `read` block I/O function read the file in 4-Kbyte blocks. For "mmap", the file is mapped into memory with `mmap` and then each page is touched, causing the file to be read into memory. These measurements exclude the time to open and close the file (about 6.3 ms) and the time to perform a `mmap` system call (about 5.7 ms). The "data fill" measurement shows the time to handle page faults. Each page of a 480-Kbyte array of initialized data was touched, causing the data to be paged in from the object file's initialized heap. For "bcopy", a 480-Kbyte array in physical memory was copied. Finally, for "RPC", the remote procedure call function sent 480-Kbytes of data from the server to the client in blocks of 4-Kbytes.

The measurements in Table 6 are of random accesses to a 480-Kbyte file. The same pseudorandom sequence was used for all measurements. For "read", `lseek` moved to a random position in the file and then `read` read one byte. These operations were repeated 120 times and 1000 times, for the respective measurements. For "mmap", the file was mapped into memory and then 120 or 1000 bytes at random offsets into the file were read from memory, causing the appropriate pages to be faulted into memory. The 120 byte measurements accessed 74 different pages, while the 1000 byte measurements accessed all 120 pages of the file.