Virtual Memory for the Sprite Operating System

Michael N. Nelson

Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

Abstract

Sprite is an operating system being designed for a network of powerful personal workstations. A virtual memory system has been designed for Sprite that currently runs on the Sun architecture. This virtual memory system has several important features. First, it allows processes to share memory. Second, it allows all of the physical pages of memory to be in use at the same time; that is, no pool of free pages is required. Third, it performs remote paging. Finally, it speeds program startup by using free memory as a cache for recently-used programs. †

---

† This work was supported in part by the Defense Advanced Research Projects Agency under contract N00039-85-C-0269, in part by the National Science Foundation under grant ECS-8351961, and in part by General Motors Corporation.

1. Introduction

This paper describes the virtual memory system for the Sprite operating system. Sprite's virtual memory model is similar in many respects to Unix 4.2BSD†, but has been redesigned to eliminate unnecessary complexity and to support three changes in computer technology that have occurred recently or are about to occur: multiprocessing, networks, and large physical memories. The resulting implementation is both simpler than Unix and more powerful. In order to show the relationship between the Sprite and Unix virtual memory systems, this paper not only provides a description of Sprite virtual memory, but also where appropriate describes the Unix virtual memory system and compares it to Sprite.

Although Sprite is being developed on Sun workstations, its eventual target is SPUR, a new multiprocessor RISC workstation being developed at U.C. Berkeley [HILL 85]. In order to support the multiprocessor aspect of SPUR, the user-level view of virtual memory provided by Unix has been extended in Sprite to allow processes to share writable segments. Section 2 compares the Sprite and Unix virtual memory systems from the user's view, and Section 3 describes the internal data structures used for the implementation of writable segments.

Although Sprite provides more functionality than Unix with features such as shared writable segments, an equally important contribution of Sprite is its reduction of complexity. Unix requires that a portion of the page frames in memory be kept available to handle page faults. These page frames are kept on a list called the free list. The problem with the free list is that extra complexity is required to manage it and page faults are required for processes that reference pages that are on the list. Sprite has eliminated the need for a free list which has resulted in less complex algorithms and more efficient use of memory. The Sprite page replacement algorithms are described in section 4.

Another major simplification in Sprite has been accomplished by taking advantage of high-bandwidth networks like the Ethernet. These networks have influenced the design of Sprite's virtual memory by allowing workstations to share high-performance file servers. As a result, most Sprite workstations will be diskless, and paging will be carried out over the network. This use of file servers has allowed Sprite to use much simpler methods than Unix for demand loading of code and managing backing store. Section 5 describes the characteristics of Sprite file servers and how they are used by the virtual memory system, including how ordinary files, accessed over the network, are used for backing store.

Another key aspect of modern engineering workstations that Sprite takes advantage of is the advent of large physical memories. Typical sizes today are 4-16 Mbytes; within a few years workstations will have hundreds of Mbytes of memory. Large physical memories offer the opportunity for speeding program startup by using free memory as a cache for recently-used programs; this mechanism is described in Section 6.

---

† All future references to Unix in this paper will refer to Unix 4.2BSD unless otherwise specified.

The design given here has been implemented and tested on Sun workstations. Section 7 gives some statistics and performance measurements for the Sun implementation.

2. Shared Memory

Processes that are working together in parallel to solve a problem need an interprocess communication (IPC) mechanism to allow them to synchronize and share data. Since Sprite will eventually be ported to a multiprocessor architecture, IPC is needed to allow users to exploit the parallelism of the multiprocessor. The traditional methods of IPC are shared writable memory and messages. Sprite uses shared writable memory for IPC because of considerations of efficiency and the SPUR architecture:

- Shared writable memory is at least as efficient as messages since shared writable memory uses the lowest-level hardware primitives without any additional layers of protocol.
- The SPUR architecture is designed to make sharing memory both simple to implement in the operating system and efficient for processes to use.

The rest of this section describes the user-level view of shared memory in both Unix and Sprite. The user-level view of shared memory in Unix is given because the Sprite view is just an extension of the Unix view.

2.1. Unix Sharing

In Unix the address space for a process consists of three segments: code, heap, and stack (see Figure 1). Processes can share code segments but they cannot share heap or stack segments. Thus writable shared memory cannot be used for IPC in Unix. There are two system calls in Unix that allow the code sharing. One call is *fork* which is used for process creation. The other call is *exec* which is used to overlay the calling process's program image with a new program image. When a process calls *fork*, a new process is

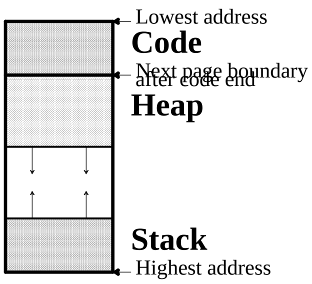

Figure 1. The Unix and Sprite Address Space. The address space for a process is made up of three segments: code, heap, and stack. The code segment begins at address 0, is of fixed size, is read-only and contains the object code for the process. The heap segment begins on the next page boundary after the end of the code segment and grows towards higher addresses. It contains all statically-allocated variables and all dynamically-allocated memory from memory allocators. The stack segment begins at the highest address and grows towards lower addresses. Thus the stack and the heap segment grow towards each other.

created that shares the caller's (parent's) code segment and gets its own copy of the parent's heap and stack segments. When a process calls *exec*, the process will get new stack and heap segments and will share code with any existing process that is using the same code segment.

2.2. Sprite Sharing

In Sprite processes can share both code and heap segments. Sharing the heap segment permits processes to share writable memory. Sprite has system calls equivalent to the Unix *fork* and *exec* that allow this sharing of code and heap. The Sprite *exec* is identical in functionality to the Unix *exec*. The Sprite *fork* is similar to the Unix *fork* except that it will also allow a process to share the heap segment of its parent. When a process is created by *fork* it is given an identical copy of its parent's stack segment, it shares its parent's code segment, and it can either share its parent's heap segment or it can get an identical copy. Whether or not a child shares its parent's heap is the option of the parent when it calls *fork*.

3. Segment Structure

Although to the user the address spaces of Unix and Sprite look identical, the two operating systems have different internal representations of segments. In this section the internal segment structure of each system is described. For a description of how the segment structure is used in each system to implement shared memory on the Sun-2 and VAX architectures see Appendix A.

3.1. Unix Segments

When a Unix process is created or overlaid, its address space is broken up into the three segments as previously described. Associated with each process is a page table for the code and heap segments, a page table for the stack segment, backing store for the heap and stack segments, and a pointer to a structure that contains the backing store for the code segment (see Figure 2). In addition, all processes that are sharing the same code segment are linked together. Thus page tables are kept on a per-process basis rather than

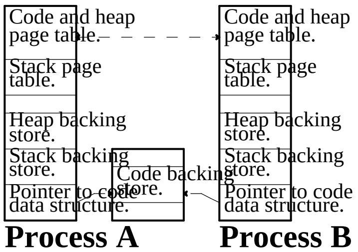

Figure 2. The state of two Unix processes that are sharing code. Both process A and process B have their own page tables for code, heap, and stack segments, and their own backing store for heap and stack segments. Each points to a common data structure that contains the backing store for the code segment. In addition the two processes are linked together.

a per-segment basis and backing store is kept on a per-segment basis. If a code segment is being shared, then the code portion of the code and heap page table for all processes that are sharing the segment is kept consistent. This means that when any process that is sharing code has the code portion of its code and heap page table updated, the page tables in all processes that are sharing the segment are updated.

3.2. Sprite Segments

Sprite has a different notion of segments than Unix. In Sprite, information about segments is kept in a separate data structure, independent of process control blocks. Each process contains pointers to the records describing its three segments. Each segment has its own page table and backing store. Thus segments and backing store are both allocated on a per-segment basis. Since page tables are associated with each segment instead of each process, processes that share code or heap segments automatically share the same page tables. Figure 3 gives an example of two Sprite processes sharing code and heap.

4. Page Replacement Algorithms

Several different page replacement algorithms have been developed. The most commonly used types of algorithms are those that are based on either a least-recently-used (LRU) page replacement policy, a first-in-first-out (FIFO) page replacement policy, or a working set policy [DENN 68]. LRU and FIFO algorithms are used to determine which page frame to replace when handling a page fault. In an LRU algorithm the page that is chosen for replacement is the page that has not been referenced by a process for the longest amount of time. In a FIFO type of algorithm, the page that is replaced is the one that was least recently paged in. Both LRU and FIFO can be applied either to all pages in the system (a global policy) or only those pages that are owned by the faulting process (a local policy). Algorithms that use a global policy allow processes to have a variable-size partition of memory whereas algorithms that use a local policy normally

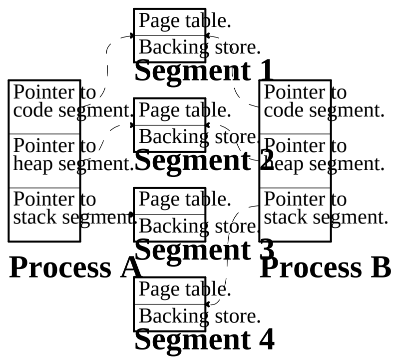

Figure 3. The state of two Sprite processes that are sharing code and heap. All that the process state contains is pointers to the three segments. Since process A and process B are sharing heap and code, they point to the same heap and code segments. However, each process has its own stack segment.

give processes fixed-size memory partitions.

Unlike the LRU or FIFO page replacement policies, the working set policy does not determine which page to replace at page fault time, but rather attempts to determine which pages a process should have in memory to minimize the number of page faults. The working set of a process is those pages that have been referenced in the last τ seconds. Any pages that have not been referenced in the last τ seconds are eligible to be used for the working sets of other processes. A process is not allowed to run unless there is a sufficient amount of free memory to fit all of the pages in its working set in memory.

Studies of algorithms that use LRU, FIFO and working set policies have yielded the following results:

- Algorithms that use an LRU policy have better performance (where better performance is defined to be a lower average page fault rate) than those that use a FIFO policy [KING 71].
- Algorithms that use global LRU have better performance than ones that use local LRU with fixed-size memory partitions [OLIV 74].
- The global LRU and working set policies provide comparable performance [OLIV 74, BABA 81a].

These results indicate that Sprite should use either a global LRU algorithm or an algorithm that uses the working set policy. In practice, these two types of algorithms are approximated rather than being implemented exactly. The first part of this section provides a description of three operating systems that have already implemented an approximation of one of these algorithms. In addition a fourth operating system, VMS, is presented that implements a different type of algorithm that can give an approximation to global LRU. The second part of this section provides a description of the Sprite page replacement algorithms.

4.1. Implementations of Page Replacement Algorithms

4.1.1. Multics

The Multics paging algorithm [CORB 69] is generally referred to as the clock algorithm. The algorithm has been shown to provide a good approximation of global LRU [GRIT 75]. In the algorithm all of the pages are arranged in a circular list. A pointer, called the clock hand, cycles through the list looking for pages that have not been accessed since the last sweep of the hand. When it finds unreferenced pages, it removes them from the process that owns them and puts them onto a list of free pages. If the page is dirty, it is written to disk before being put onto the free list. This algorithm is activated at page fault time whenever the number of free pages falls below a lower bound. Whenever the algorithm is activated several pages are put onto the list of free pages.

The clock algorithm is relatively simple to implement. The only hardware support required is reference and modified bits. In fact the Unix version of the clock algorithm was implemented without any hardware reference bits [BABA 81a, BABA 81b].

4.1.2. Tenex

There are very few examples of operating systems that have implemented algorithms that use the working set policy. An example of an operating system that implements something similar to it is Tenex [BOBR 72]. There are two main differences between the policy used by Tenex and the working set policy. First, instead of defining the working set of a process to be those pages that have been referenced in the last τ seconds, it is defined to be the set of pages that keeps the page fault frequency down to an acceptable level. Second, instead of removing pages from the working set as soon as they have been unreferenced for τ seconds, pages are only removed from the working set at page fault time.

The actual algorithm used in Tenex is the following. Whenever a process experiences a page fault, the average time between page faults is calculated. If it is greater than *PVA*, a system parameter, then the process is considered to be below its working set size. In this case the requested page is brought into memory and added to the working set. If the page fault average is below *PVA*, then before loading the requested page into memory the size of the working set is reduced by using an LRU algorithm. Since removing pages is costly, whenever the working set is reduced all sufficiently old pages are removed.

Chu and Operbeck [CHU 76] performed some measurements of an algorithm that is very similar to the algorithm used in Tenex. They showed that this type of algorithm is able to produce performance comparable to that of algorithms that use Dennings working set policy.

4.1.3. VMS

VMS uses a fixed-size memory partition policy [LEVY 82, KENA 84]. Each process is assigned a fixed-size partition of memory called its resident set. Whenever a process reaches the size of its resident set, the process must release a page for every page that is added to its resident set. VMS uses a simple first-in-first-out (FIFO) policy to select the page to release. There are two lists that contain pages that are not members of any process's resident set: the free list and the dirty list. Both of these lists are managed by an LRU policy. Whenever a page is removed from a process's resident set it is put onto the free list if it is clean or the dirty list if it is dirty. The two lists are used when handling a page fault. If the page that is faulted on is still on one of the lists then it is removed from the list and added to the process's resident set. Otherwise the page on the head of the free list is removed, loaded with data, and added to the process's resident set. Once the dirty list gets too many pages on it, some are cleaned and moved to the free list.

The VMS algorithm is interesting for two reasons. First, it does not require reference bits. Second, it is an example of a hybrid between a FIFO and a LRU algorithm. Each process's local partition is managed FIFO but the global free list is managed LRU. It has been shown that for a given program and a given memory size, the resident set size can be set so as to achieve a fault rate close to that of LRU [BABA 81a]. However, the optimal resident set size for a fixed free list size is very sensitive to changes in memory size and the program. With a fixed sized partition for all programs it would be very difficult to choose one partition that would be optimal for all programs. Another problem with fixed size partitions is that programs cannot take advantage of a large amount of free pages once they have reached their maximum partition size. More recent versions of VMS [KENA 84] have relaxed the restriction on the size of the resident set by allowing the resident set size to increase to a higher limit if the free and dirty lists contain a sufficient number of pages.

4.1.4. Unix

The Unix page replacement policy is based on a variant of the clock algorithm [BABA 81b]. Unlike Multics the clock algorithm is run periodically instead of at page fault time. The data structures used to implement the clock algorithm are the core map and the free list (see Figure 4). The core map is a large array containing one entry for each page frame with the entries stored in order of appearance in physical memory. It forms the circular list of pages required by the clock algorithm. The free list is a list of pages that is used at page fault time. It contains two types of pages: those that are not being used by any process and those that were put onto the list by the clock algorithm. The unused pages are at the front of the list.

The clock algorithm is simulated by a kernel process called the pageout daemon. It is run often enough so that there are a sufficient number of pages on the free list to handle bursts of page faults. The pageout daemon cycles through the core map looking for pages that have not been referenced since the last time that they were looked at by the pageout daemon. Any unreferenced pages that it finds are put onto the end of the free list. There are four constants used to manage the pageout daemon. As long as the number of free pages is greater than *lotsfree* (1/8 of memory on the Sun version of Unix) then the pageout daemon is not run. When the number of free pages is between *desfree* (at most 1/16 of memory) and *lotsfree* then the pageout daemon is run at a rate that attempts to keep the number of pages on the free list acceptably high, while not taking more than ten percent of the CPU. When the number of free pages falls below *desfree* for an extended period of time then a process is swapped out. The process chosen to be swapped out is the oldest of the *nbig* largest processes. *Minfree* is the minimum tolerable size of the free list. If the free list has only *minfree* pages then every time a page is removed from the free list the pageout daemon is awakened to add more pages to the list.

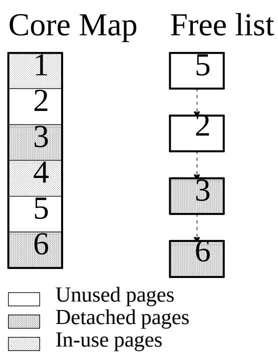

Figure 4. Unix page replacement data structures. In this example there are 6 pages in physical memory. Pages 1 and 4 are currently being used by processes, pages 3 and 6 have been detached from the processes that were using them by the clock algorithm, and pages 2 and 5 are not being used by any process. Thus pages 5, 2, 3, and 6 (in that order) are eligible to be used when a new page is needed and pages 1 and 4 are not eligible. Pages 3 and 6 are also eligible to be reclaimed by the process that originally owned them if the original owner faults on them.

The free list is used when a page fault occurs. Like VMS, there are two ways in which the free list is used to handle a page fault. The first is if the requested virtual page is still on the free list. If this is the case, then the page can be reclaimed by taking it off of the free list and giving it to the faulting process. The second way in which the free list is used is if the requested virtual page is not currently on the free list. In this case a page is removed from the front of the free list, it is loaded with code or data, and it is given to the faulting process.

4.2. Sprite Page Replacement

We decided to base the Sprite page replacement algorithm on the clock algorithm because of its inherent simplicity. However, we decided to do things differently than Unix. The main difference is that Sprite no longer uses a free list. The free list serves an important function by eliminating the need to activate the clock algorithm on every page fault. However, it has the disadvantage of reducing the amount of memory that can be referenced without causing page faults; any reference to a page on the free list requires a page fault to reclaim the page. Sprite does not need a free list because it uses the clock algorithm to maintain an ordered list of all pages in the system. When a page fault occurs this ordered list of pages can be used to quickly find a page to use for the page fault. This section describes the Sprite algorithm that maintains and uses this list.

4.2.1. Data Structures

There are three data structures that the page replacement algorithm uses: the core map, the allocate list, and the dirty list. Diagrams of all three data structures are shown in Figure 5. The core map is identical in form to the Unix core map. It is used by the Sprite version of the clock algorithm to help manage the allocate list. The allocate list contains all pages except those that are on the dirty list. All unused pages are on the front of the list and the rest of the pages follow in approximate LRU order. The list is used when trying to find a new page. The allocate list is managed by both a version of the clock algorithm and the page allocation algorithm. The dirty list contains pages that are being written to backing store. Pages are put onto the dirty list by the page allocation

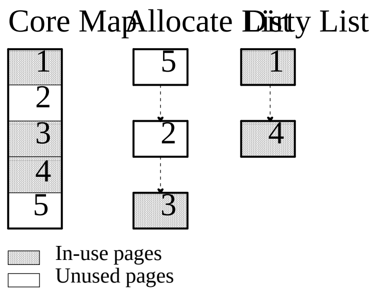

Figure 5. Sprite page replacement data structures. Every page is present in the core map and either the allocate list or the dirty list. In this example there are 5 pages in physical memory. Pages 1, 3, and 4 are currently being used by processes and pages 2 and 5 are unused. Of the pages in use, pages 1 and 4 are on the dirty list waiting to be cleaned.

algorithm.

4.2.2. Clock Algorithm

Sprite uses a variant of the clock algorithm to keep the allocate list in approximate LRU order. A process periodically cycles through the core map moving pages that have their reference bit set to the end of the allocate list. All pages have their reference bit cleared before they are moved to the end of the allocate list. Since unreferenced pages are not moved by the clock algorithm, they will migrate towards the front of the allocate list and referenced pages will stay near the end. The rate at which the core map is cycled through is not known yet.

4.2.3. Page Allocation

Figure 6 summarizes the Sprite page allocation algorithm. The basic algorithm is to remove pages from the front of the allocate list until an unreferenced, unmodified page is found. If the frontmost page has been referenced since the last time that it was examined by the clock algorithm, then it is moved to the end of the allocate list. If the page is modified but not referenced since the last time that it was examined by the clock algorithm, then it is moved to the dirty list. Once an unreferenced, unmodified page is found it is used to handle the page fault. If the page that is selected to handle the page fault is still being used by some process, the page must be detached from the process that owns it before it can be given to the faulting process.

4.2.4. Page Cleaning

A kernel process is used to write dirty pages to backing store. It wakes up when pages are put onto the dirty list. Before a page is written to backing store its modified bit is cleared. After a page is written to backing store it is moved back to the front of the allocate list.

4.2.5. How Much Does a Page Fault Cost

When a page fault occurs on Sprite there are three steps to processing the page fault: find a page frame, detach it from the process that owns it (if any), and fill the page with code or data. The overhead required to fill the page on a Sun-2 is between 3.6 ms for a zero-filled page fault (see Section 8) up to as much as 30 ms to fill the page from disk. The actual time required to the fill a page will vary with the page size. The overhead of filling a page must be paid by any page replacement algorithm. The additional cost that Sprite has to pay at page fault time because it does not keep a free list is searching the allocate list for an unreferenced, unmodified page and then detaching the page from its

	Unreferenced	**Referenced**
Unmodified	Take page	Move to end of page list and clear reference bit
Modified	Move to dirty list	

Figure 6. Summary of Sprite page allocation algorithm.

owner.

Since the allocate list is in approximate LRU order, an unreferenced page should be able to be found quickly. How quickly an unmodified page can be found is dependent on the percentage of memory that is dirty. Since code pages are read-only, there must be clean pages. If a large percentage of memory is dirty then it might take a long search on a page fault to find a clean page. However, since all dirty pages that were found during the search will be cleaned and then put back onto the front of the allocate list, subsequent page faults should be able to find clean pages very quickly. Therefore in the average case the number of pages that have to be searched to find an unreferenced, unmodified page should be small.

The cost of detaching a page from its owner is shown in Section 8 to be 0.3 ms. This cost combined with the cost of searching a small number of pages on the allocate list is small in comparison to the cost of filling the page frame. Therefore the overhead added because Sprite does not keep a free list should be small relative to the large cost of either zero-filling a page or filling it from the file server.

5. Demand Loading of Code and Backing Store Management

A virtual memory system must be able to load pages into memory from the file system or backing store when a process faults on a page and write pages to backing store when removing a dirty page from memory. This can be done by either using the file system both to load pages and to provide backing store or using the file system to load pages and using a separate mechanism for backing store. For Sprite we chose to use the file system for both demand loading and backing store. Examples of other systems that use the file system for backing store are Multics [ORGA 72] and Pilot [REDE 80]. An example of a system that uses a separate mechanism for backing store is Unix. In this section the methods that Sprite and Unix use for demand loading of pages and managing backing store are described in detail and compared.

5.1. Lifetime of a Page

There are three different types of pages in Unix and Sprite: read-only code pages, initialized heap pages, and uninitialized heap or stack pages (i.e. those whose initial contents are all zeroes). There are also three places where these pages can live: in the file system, in backing store, or in main memory. This section describes where the three types of pages spend their lifetimes.

5.1.1. Lifetime of a Unix Page

Figure 7 shows the lifetime of the three types of Unix pages. Code pages can live in three places: main memory, in an object file in the file system, and in backing store. An object file is a file that contains the code and initialized heap pages for a program. When the first page fault occurs for a code page, the page is read from the object file into a page frame in memory. When the page is replaced, its contents is written to backing store even though the page could not have been modified since code pages are read-only. Subsequent reads of the code page come from backing store. Since a code page is read-only, it only has to be written to backing store the first time that it is removed from memory.

The lifetime of an initialized heap page is almost identical to that of a code page. Like a code page an initialized heap page is loaded from the object file when it is first

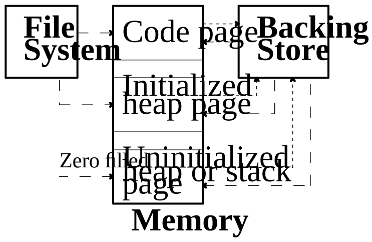

Figure 7. Lifetime of a Unix page. Code pages and initialized heap pages begin their life in the file system. However, once they are thrown out of memory they are written to backing store, and spend the rest of their life in memory and backing store. Uninitialized heap and stack pages are filled with zeroes initially and then spend the rest of their life in memory and backing store.

faulted on, it is written to backing store the first time that it is replaced and all faults on the page after the first one load the page from backing store. Unlike code pages, initialized heap pages can be modified. Because of this, in addition to being written to backing store the first time that they are replaced, initialized heap pages must be written to backing store if when they are replaced they have been modified.

Unlike code and initialized heap pages, uninitialized heap pages and stack pages never exist in the file system. Their entire existence is spent in memory and in backing store. When an uninitialized heap or stack page is first faulted in, it is filled with zeroes. When the page is taken away from the heap or stack segment, it is written to backing store. From this point on whenever a page fault occurs for the page, it is read from backing store. Whenever the page is thrown out of memory, if it has been modified then it is written back to backing store.

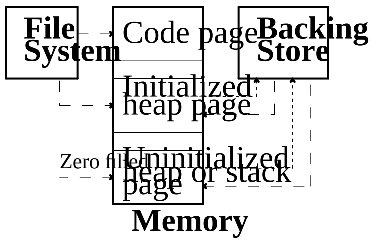

Figure 8. Lifetime of a Sprite page. Code pages live in two places, the file system and memory. Code pages never have to live on backing store because they are never modified. Initialized data pages live in the file system and memory until they get modified, and then spend the rest of their life in memory and backing store. Uninitialized heap and stack pages are filled with zeroes initially and then spend the rest of their life in memory and backing store.

5.1.2. Lifetime of a Sprite Page

The lifetime of a Sprite page (see Figure 8) is very similar to the lifetime of a Unix page. The only difference is the treatment of code pages. Unlike Unix code pages, Sprite code pages can only live in memory and in the object file. When a page fault occurs for a code page, the page is read from the object file into a page frame in memory. When the page is replaced, the contents are discarded since code is read-only. Thus whenever a page fault occurs for a code page it is always read from the object file. The reasons why Unix writes read-only code pages to backing store and Sprite does not are explained in the next section.

5.2. Demand Loading of Code

Unix and Sprite both initially load code and initialized heap pages from object files in the file system. However, the method of reading these pages is different. In Sprite the pages are read using the normal file system operations. In Unix the pages are read directly from the disk without going through the file system. In addition after a code page has been read in once, the object file is bypassed with all subsequent reads coming from backing store. The reasons behind the differences between Unix and Sprite are based on different perceptions of the performance of the file system.

The designers of Unix believed that the file system was too slow for virtual memory. Therefore they tried to use the file system as little as possible. This is why Unix bypasses the file system and uses absolute disk block numbers when demand-loading code or initialized heap pages and why it writes code pages to backing store for subsequent page faults. When a code or initialized heap segment is initially created for a process, the virtual memory system determines the physical block numbers for each page of virtual memory that is on disk. When a code or initialized heap page is first faulted in, the physical address is used to determine where to read the page from. All later reads of the code page are able to use an absolute disk address for the code page in backing store.

There are two reasons why the Unix designers believed that the Unix file system was too slow for virtual memory. The first reason is that when performing a read through the file system there is overhead in translating from a file offset to an absolute disk address. Thus the Unix virtual memory system uses absolute disk addresses to eliminate this overhead. The second reason is that the file system block size was too small. In the original versions of Unix (prior to Unix 4.1BSD), the block size was only 512 bytes [THOM 78]. This severely limited file system throughput [MCKU 84]. The solution was to write code pages to backing store which, as will be explained later, is allocated in contiguous blocks on disk. This allows many pages to be transferred at once, providing higher throughput. Since Unix 4.2BSD uses large block sizes (4096 bytes or larger) there is no longer any advantage for Unix to write code pages to backing store.

We believe that the Sprite file system is fast enough for the virtual memory system to use it for all demand loading; there is no need to use absolute disk block numbers or write code pages to backing store. The reason is that the file system will be implemented using high-performance file servers which will be dedicated machines with local disks and large physical memories. The Sprite file system will have the following characteristics:

- The large memory of the file servers will be used as a block cache. The cache size for the present will be around 8 Mbytes. In a few years the cache size will probably be between 50 and 100 Mbytes. It has been shown that when page caches are large enough (several Mbytes), disk traffic can by reduced by 90% or more [OUST 85].
- Combined with the large cache will be large file block sizes (16 Kbytes or more). This results in the fewest disk accesses [OUST 85].
- In addition to caching pages, the large memory can be used to cache file indexing information to allow faster access to blocks within a file.

Since the file system is used for demand loading of code and initialized heap, the normal file system operations can be used. When a code or heap segment is first created the *open* operation is used to get a token to allow access to the object file for future page faults. The *close* operation is used when the virtual memory system has finished with an object file. The *read* operation is used to load in a page when a page fault occurs. When a read operation is executed the file server is sent the token for the file, an offset into the file where to read from, and a size. The file server then returns the page.

The Sprite method of using a high-performance file system has several advantages over the Unix method of bypassing the file system. First, the virtual memory system is simplified because it does not have to worry about the physical location of pages on disk. This is because when demand loading a page, offsets from the beginning of the object file are used instead of absolute disk block numbers. These offsets can be easily calculated given any virtual address. Second, the file server's cache can be used to increase performance. Page reads may be able to be serviced out of the cache instead of having to go to disk. Third, code pages do not have to be transferred to backing store.

5.3. Backing Store Management

Backing store is used to store dirty pages when they are taken away from a segment. In Sprite each segment has its own file in the file system that it uses for backing store. Unix on the other hand gives each CPU its own special partition of disk space which all segments use together for backing store. The reason for the difference is once again based on different ideas about the performance of the file systems in Unix and Sprite.

As was said before, the Unix designers believed that the Unix file system was too slow for the virtual memory system. Because of this a special area of disk is allocated for the virtual memory system. When each process is created, contiguous chunks of backing store are allocated for each of its code, heap, and stack segments. As segments outgrow the amount of backing store allocated for them, more backing store is allocated in large contiguous chunks on disk. A table of disk addresses for the backing store is kept for each segment. When a page is written to disk the physical address from this table is used to determine where to write the page. If the machine has a local disk then all backing store is allocated on its local disk. Otherwise all backing store is allocated on a partition of the server's disk.

Since Sprite uses a high-performance file system, all writing out of dirty pages is done to files in the file system instead of to a special disk partition dedicated to backing store. The first time that a segment needs to write a dirty page out, a file is opened. We chose not to open the file until a dirty page has to be written out on the assumption that most segments will never have any dirty pages written out. From this point on all dirty pages that need to be saved in backing store are written to the file using the normal file *write* operation. This just involves taking the virtual address of the page to be written and presenting it along with the page to the file server. All reads from backing store can use the normal file *read* operation by using the virtual address of the page to be read. When the segment is destroyed, the file is closed and removed.

There are several advantages of using files for backing store instead of using a separate partition of disk. First, the virtual memory system can deal with virtual addresses instead of absolute disk block numbers. This frees the virtual memory system from having to perform bookkeeping about the location of pages on disk. Second, no preallocated partition of disk space is required for each CPU. This can represent a major savings in disk space. For example, under Unix each Sun workstation requires a disk partition of approximately 16 Mbytes for backing store, most of which is rarely used. Third, process migration - the ability to preempt a process, move its execution state to another machine, and then resume its execution - is simplified. In Unix, backing store is private to a machine, so the actual backing store itself would have to be transferred if a process were migrated. In Sprite, the backing store is part of a shared file system so only a pointer to the file used for backing store would have to be transferred. Finally, Sprite has the potential for higher performance. Since file servers will have large physical memories dedicated to caching pages, the virtual memory system may be able to get pages by hitting on the cache instead of going to disk.

There is one problem with the Sprite scheme of using files. Since disk space is not preallocated it is possible that there may be no disk space available when a dirty page needs to be written out of memory. This will not be a problem as long as the file server's cache has enough room to hold all dirty pages that are written out. However, if the file server cannot buffer all pages that are written out, then this could be a serious problem. A solution is to create a separate partition on the server's disk that is only used for backing store files. This should still represent a savings in disk space over the Unix method because the partition for backing store files can be shared by all workstations.

6. Fast Program Startup

In order for a program to run it must demand load in code and initialized heap pages. These page faults are much more expensive than the initial faults for stack and uninitialized heap pages because code and initialized heap faults require a file server transaction whereas stack and uninitialized heap faults just cause new zero-filled pages to be created. The section under shared memory described one method of eliminating these code and initialized heap page faults by having a newly-created process share code with another process that is already using the desired code segment. However this is of no use if there is no process that is actively using the code segment that is needed. In order to help eliminate code page faults in the case when the normal shared memory mechanism will not work, Sprite uses memory as a cache to hold pages from code segments that were recently in use. The use of memory for this purpose is possible because of the advent of large physical memories and the fact that code is read-only.

Caching of code pages is implemented by saving the state of a code segment when the last process to reference it dies. The saved state includes the page table and all pages that are in memory. Such a code segment is called *inactive*. All of the pages associated with an inactive segment will remain in memory until they are removed by the normal page replacement algorithm. When a process overlays its address space with a new program image, if the code segment of the program corresponds to one of these inactive segments, then the process will reuse the inactive segment instead of allocating a new one.

There are two details to the algorithm for reusing inactive segments that need to be mentioned. First, it may be the case that when a new segment is being created all segments are either inactive or in use. In this case the inactive segment that has been inactive for the longest amount of time is recycled and used to create the new segment. This entails freeing any memory resident pages that the inactive segment may have allocated to it. Second, whenever a process overlays its address space with a new program image, the code segment that is used needs to be from the most recent version of the object file. This is accomplished by opening the object file for the needed code segment to get a token that uniquely identifies the current version of the file. This token is then compared to the tokens for all inactive code segments. A match will occur only if there is a code segment that is from the most recent version of the object file.

The actual performance improvement from the use of inactive segments cannot be measured until the operating system becomes fully operational. However, an indication of the reduction in startup cost can be obtained by determining the ratio of code to initialized heap in the standard Unix programs. I examined over 300 Unix programs† and determined that on the average there was nearly three times as much code as initialized heap. This means the elimination of all code page faults could reduce the startup cost by as much as 75 percent. The actual improvement is dependent on how recently the program has been used, the demand on memory and how many code and initialized heap pages are faulted in by the program.

7. Implementation of Virtual Memory

The virtual memory system that has been described in this paper has been fully implemented on the Sun architecture. In this section some issues related to the implementation and performance of the virtual memory system are discussed.

7.1. Hardware Dependencies

Certain portions of all virtual memory systems are inherently hardware dependent. This includes reading and writing a page table entry and reading and writing hardware virtual memory registers. Even if the code for a virtual memory system is written in a high level language, it must be modified when it is made to run on different machine architectures so that it can handle these hardware dependencies. The Sprite implementation is structured to allow these modifications to be made easily. In contrast, the Unix implementation is not structured to allow it to be easily ported to other architectures.

The Unix virtual memory system was initially written for the VAX architecture. The implementation contains code, algorithms, and data structures that are specifically for the VAX architecture. Instead of being isolated in a single portion of the code, these

---

† The programs that were examined came from Sun's version of /bin, /usr/bin, /usr/local and /usr/ucb. The measurements were performed by using the *size* command on each program and then taking the ratio of initialized code to initialized heap.

hardware dependencies are strewn throughout the code. An example of what is involved in porting the Unix code is the job that the people from Sun Microsystems did when porting Unix from the VAX to the Sun. When they did the port they did three things. First, they used the conditional compilation feature of C (the language that Unix is written in) to separate many of the VAX and Sun hardware dependencies within the code. As a result the code consists of a mixture of VAX and Sun code separated by *ifdef* statements. Second, they wrote a separate file of hardware-dependent routines to manage the Sun virtual memory hardware. Third, they left in some data structures and algorithms that are there because the code was written for a VAX. An example of this is the simulation of reference bits [BABA 81a, BABA 81b]. This was only necessary because the VAX does not have reference bits. Even though the Sun does have reference bits they were not used because it was simpler to stick with the simulation of reference bits.

Since Sprite is going to be ported to the SPUR architecture, one of my goals was to make the virtual memory system easier to port than Unix. This involved defining a separate set of routines that handled all hardware dependent operations. These routines are called by the hardware-independent routines. When we port Sprite to another architecture only the hardware-dependent routines should need to be rewritten.

The result of dividing the code into hardware-dependent and hardware-independent parts is that approximately half of the code is hardware-dependent and half hardware-independent (the actual code size is given in Table 1). Implementation of the clock and page replacement algorithms, handling page faults, creation of new segments, and the management of the segment table have been done with hardware-independent code. Reading and writing page table entries, validating and invalidating pages, mapping pages into and out of the kernel's address space, and managing the Sun memory mapping hardware have all been written as hardware-dependent code. Thus only the low-level operations have to be reimplemented when we port Sprite to another architecture.

	Lines of C		Lines of Assembler		Bytes of Compiled Code
	With Comments	Without Comments†	With Comments	Without Comments	
Machine Dependent	2423	970	359	107	8560
Machine Independent	2825	1095	0	0	8484
Total	5248	2065	359	107	17044

Table 1. Sprite code size. Half of the Sprite code is hardware-dependent and half is hardware-independent. In addition almost two-thirds of the code is comments. Blank lines are treated as comments.

	Lines of C		Lines of Assembler		Bytes of Compiled Code
	With Comments	Without Comments	With Comments	Without Comments	
Total	5855	3833	173	125	32948

Table 2. Unix code size. Unix code is a mixture of hardware-dependent and hardware-independent code. Thus unlike Table 1 there is no differentiation in this table between hardware-dependent and hardware-independent code. Also note that only a little more than one-third of the code is comments. Blank lines are treated as comments.

7.2. Code Size

The virtual memory systems for Sprite and Unix were both written mostly in C with a small amount being written in assembler. Tables 1 and 2 give the sizes of the source and object codes for both systems. There are three observations that can be made from these two tables. First of all, the Sprite source code is much more heavily documented than the Unix sources. This can be seen from the fact that although Unix and Sprite have almost the same amount of C code including comments, Unix has twice as many lines of code not including comments. The second observation is that Sprite is not as complex as Unix. This can be seen from the fact that Unix has twice as many bytes of compiled code as Sprite. Although code size is not an absolute measure of complexity, the fact that Sprite requires half as much code as Unix is an approximate indication of their relative complexities. The final observation from the two tables is the small amount of assembly code in either system. This shows that both virtual memory systems can be written almost entirely in a high-level language.

7.3. Performance

Since the Sprite virtual memory system is currently running on the Sun architecture, I have been able to do some preliminary performance evaluation of the virtual memory system. All of the performance evaluation was done on a Sun-2 workstation (which uses a Motorola 68010 microprocessor). This performance evaluation falls into two categories. The first is the speed of simple zero-filled page faults. The second category is the overhead of the clock algorithm. This measurement consists of determining what percentage of the CPU is required to run the clock algorithm at different rates. Things that were not measured include the time required to read and write pages from/to backing store, the time required to demand load a page from an object file, and the overhead required when memory demand is so tight that pages have to be written to backing store. These were not measured because they are dominated by the speed of the file server and not the speed of virtual memory. Since we are not using the high-performance file server that we will be using in a few months, these measurements are not a good indication of the eventual speed of the virtual memory system.

7.3.1. Simple Page Faults

Table 3 gives the times required to perform zero-filled page faults on Sprite and Unix. These page faults occur when either an uninitialized heap page or a stack page is

	Sprite	Unix
Free Page	3.6 ms	3.5 to 4.0 ms
In Use Page	3.9	----

Table 3. Time to zero-fill a page in milliseconds. The first row is the amount of time required when the page that is being zero-filled does not have to be detached from a process. The second row is the amount of time required when the page must be detached from the process that owns it. Since Unix never has to detach a page, there is no entry in the second row for Unix.

first faulted on. These simple page faults consist of getting a page, filling it with zeroes and giving it to the process that faulted on it. In Sprite if the page is not in use it takes 3.6 ms to zero-fill it and if it is in use it takes 3.9 ms. Thus it only takes 0.3 ms to detach a page from a user process. Unix, which always has free pages available, varies between 3.5 and 4.0 ms for a zero-filled page fault. For both Unix and Sprite the portion of the zero-filled page fault time that is spent writing zeroes into the page is approximately 1 ms.

	All Referenced		All in Use		None in Use	
Pages per Second	Execution Time (sec)	Percent Slowdown	Execution Time (sec)	Percent Slowdown	Execution Time (sec)	Percent Slowdown
0	360	0	360	0	360	0
100	376	4	367	2	362	0.5
500	415	16	390	8	370	3
1000	475	32	415	15	384	7

Table 4. Clock algorithm overhead. There are three different states for pages examined by the clock algorithm: in use and referenced, in use and not referenced, and not in use. An in-use-and-referenced page requires the most overhead because the reference bit has to be examined and cleared, and the page moved to the end of the allocate list. An in-use-and-not-referenced page requires the second largest overhead because the reference bit has to be examined but it does not have to be cleared and the page does not have to be moved. Finally, pages that are not in use require the least amount of overhead because they can be ignored. The above table shows the amount of overhead required to execute the clock algorithm for pages of the three different types at different speeds. The left column shows the overhead when all pages in memory are in use and referenced (the worst case). The middle column shows the overhead when all pages in memory are in use but none are referenced. The right column shows the overhead when no pages are in use (the best case). The overhead was measured by executing a low priority user process that executes a simple four instruction loop 100,000,000 times. While this process was executing the clock algorithm was running at high priority once a second. The overhead was measured by determining how much extra time was required by the user process to complete while the clock algorithm executed.

7.3.2. Overhead of Clock Algorithm

I was able to measure the overhead of the clock algorithm for Sprite. However, I was unable to do similar measurements for the Unix clock algorithm. The reason is that, whereas I was able to easily modify the Sprite kernel to give me good information, it was unclear how to easily modify the Unix kernel to provide me with the necessary information. As a result there is no comparison of the overheads required by the Sprite clock algorithm and the Unix clock algorithm.

Table 4 shows the overhead required by the Sprite clock algorithm. In the worst case, when all pages in memory are in use and being referenced, the algorithm requires between 3 and 4 percent of the CPU for each 100 pages checked per second. In the best case when nothing is being used, it takes around one-half of one percent of the CPU. The significance of these numbers can be seen by determining what is the maximum rate at which the clock must be run to give a good approximation of LRU.

In order for the clock algorithm to keep the allocate list in approximate LRU order, pages must be scanned at a rate several times that of the page fault rate. This will guarantee that as a page rises from the end of the allocate list to the front of the allocate list, it will be examined multiple times by the clock algorithm giving it multiple chances to be put back onto the end of the list. Thus pages that make it to the front of the allocate list will not have been accessed very recently, with the exception of pages that were accessed since the last time that the clock algorithm examined them; that is, the allocate list will be in approximate LRU order.

Since the clock hand must move at several times the page fault rate, the maximum clock rate can only be determined after determining the maximum page fault rate. An upper bound on the page fault rate is the number of zero-filled page faults that can be executed per second. At 3.6 ms per page fault, there can be 277 of these types of page faults per second. However, this is unrealistically pessimistic. When a page is zero-filled it is marked as dirty. Thus if all of memory is being filled by zero-filled pages, then all of memory must be dirty. This means that whenever a zero-filled page fault is handled a page must be written to the file used for backing store. Therefore the actual maximum sustained rate of zero-filled pages faults is the rate that pages can be written to the file server. Since all other types of page faults require a read from the file server, the maximum page fault rate is limited by the speed at which pages can be read or written from/to the file server. Assuming optimistically that it only takes 10 ms to read or write a page from/to the file server, the upper bound on the number of page faults per second is 100. Running the clock algorithm at a rate five times this maximum page fault rate would require 500 pages to be scanned per second or a worst-case overhead of 16 percent.

The page fault rate of 100 pages per second just given is a very pessimistic estimate. There was a study done of the paging activity of several VAX computers running Unix [NELS 84]. It measured the number of page-ins per second averaged over 5, 10, 30 and 60 second intervals when the system had an average of 8 runnable processes. The results showed that the maximum page-in rate over a 5 second interval was 25 pages per second. However, the page-in activity was very bursty and when averaged over longer intervals it dropped dramatically. For example when averaged over 60 second intervals, the maximum page-in rate was 4 per second. The page-in rate averaged over the whole study was less than one per second. Because of the results of this study, we expect that the actual page fault rate that we will experience will be much smaller then the upper bound of 100 pages per second. This should result in a clock rate much lower than 500 pages per second. In addition the overhead of the clock algorithm will not be as bad as the worst case of 3 percent per 100 pages scanned because memory is not usually all in use and all referenced. The combination of a lower clock rate and lower overhead should make the actual total overhead of the clock algorithm negligible (between one and two percent).

8. Conclusion

A virtual memory system has been presented that provides the basic functionality of the Unix 4.2BSD virtual memory system while being simpler, avoiding some of its problems, and providing additional functionality. The simplifications come from using remote servers for paging and a simplified page replacement algorithm. A problem with Unix that is eliminated is the extra page faults required to reclaim pages off of the free list. Sprite eliminates these extra page faults by replacing the free list with a list that contains all of the pages in memory in approximate LRU order. The list is then used to quickly find pages to handle page faults. The extra functionality that Sprite provides is that it allows processes to share writable memory and it reuses old segments to allow fast program startup. The shared writable memory allows high-speed interprocess communication between processes.

The Sprite virtual memory system as described in this paper is fully operational. The rest of the operating system is still under development. We hope that Sprite will be in use by the developers by the end of Summer 1986 and in use in the research community in 1987.

9. References

[BABA 81a]

Babaoglu, O. ‘‘Virtual Storage Management in the Absence of Reference Bits.’’ Ph.D. Thesis, Computer Science Division, University of California, Berkeley, November 1981.

[BABA 81b]

Babaoglu, O., and Joy, W.N. ‘‘Converting a Swap-based System to do Paging in an Architecture Lacking Page-Referenced Bits.’’ *Proceedings of the 8th Symposium on Operating Systems Principles*, 1981, pp. 78-86.

[BOBR 72]

Bobrow, D.G., et al. ‘‘TENEX, a Paged Time Sharing System for the PDP-10.’’ *Communications of the ACM*, Vol. 15, No. 3, March 1972, pp. 135-143.

[CHU 76]

Chu, W., Opderbeck, H. ‘‘Program Behavior and the Page Fault Frequency Replacement Algorithm.’’ *IEEE Computer*, Nov. 1976, pp. 29-38.

[CORB 69]

Corbato, F.J. ‘‘A Paging Experiment with the Multics System.’’ In *In Honor of Philip M. Morse* (edited Feshbach and Ingard), MIT Press, Cambridge, Mass., 1969, pp. 217-228.

[DENN 68]
Denning, P.J. ‘‘The Working Set Model for Program Behavior.’’ *Communications of the ACM*, Vol. 11, No. 5, May 1968, pp. 323-333.

[GRIT 75]
Grit, D.H., and Kain, R.Y. ‘‘An Analysis of the Use Bit Page Replacement Algorithm.’’ *Proceedings of the ACM Annual Conference*, Minneapolis, Minn., 1975, pp.187-192.

[HILL 85]
Hill, M.D., et al. ‘‘SPUR: A VLSI Multiprocessor Workstation.’’ Computer Science Division Technical Report No. UCB/CSD 86/273, University of California, Berkeley, December 1985.

[KENA 84]
Kenah, L.J., Bate, S.F. *VAX/VMS Internals and Data Structures*, Digital Press, Bedford, Mass., 1984.

[KING 71]
King, W.F. III. ‘‘Analysis of Demand Paging Algorithms.’’ *Proceedings of IFIPS Congress*, Ljubljana, Yugoslavia, 1971, Vol. 1, pp. 485-490.

[LEVY 82]
Levy, H., Lipman, P. ‘‘Virtual Memory Management in the VAX/VMS Operating System.’’ *IEEE Computer*, March 1982, pp. 35-41.

[MCKU 84]
McKusick, M.K., Joy, W.N., Leffler, S.J., and Fabry, R.S. ‘‘A Fast File System for UNIX’’, *ACM Transactions on Computer Systems*, Vol. 2, No. 3, August 1984, pp. 181-197.

[NELS 84]
Nelson, M.N., and Duffy, J.A. ‘‘Feasibility of Network Paging and a Page Server Design.’’ Term project, CS 262, Department of EECS, University of California, Berkeley, May, 1984.

[ORGA 72]
Organick, E.I. *The Multics System: An Examination of Its Structure*, MIT Press, Cambridge, Mass., 1972.

[OLIV 74]
Oliver, N.A. ‘‘Experimental data on page replacement algorithm.’’ *Proceedings NCC*, 1974, pp. 179-184.

[OUST 85]
Ousterhout, J.K. et al. ‘‘A Trace-Driven Analysis of the 4.2 BSD UNIX File System.’’ *Proceedings of the 10th Symposium on Operating Systems Principles*, 1985, pp. 15-24.

[REDE 80]
Redell, D.D. et al. ‘‘Pilot: An Operating System for a Personal Computer.’’ *Communications of the ACM*, Vol. 23, No. 2, Feb. 1980, pp. 81-92.

[THOM 78]
Thompson, K. ‘‘The Unix Time-Sharing System: Unix Implementation.’’ *Bell*

System Technical Journal, Vol. 57, No. 6, July-August 1978, pp. 1931-1946.

Appendix A

Implementation of Sharing on VAX and Sun-2 Architectures

Section 3 described the internal representation of segments in Unix and Sprite. This appendix describes how each system either uses or could use its representation of segments to implement sharing on the VAX and Sun-2 architectures. The description of the implementation of sharing includes a description of the memory mapping hardware provided by the VAX and Sun-2 architectures.

A.1 VAX

The VAX divides the address space of each process into the two regions P0 and P1 [KENA 84]. P0 contains the code and heap and P1 contains the stack. Page tables for each of these regions are kept in the kernel's virtual address space. There are two registers P0BR and P1BR that point to the base of the P0 and P1 region page tables respectively. These two registers must be set up correctly by the operating system whenever a user process begins executing in order to allow virtual address translation to be performed.

The Unix organization for the page tables for a process was developed on a VAX. This explains why the page table structure described in Section 3 came about. When a process begins execution Unix just sets the P0BR and P1BR registers to point to the process's page tables. Thus the Unix page table structure works very easily on the VAX

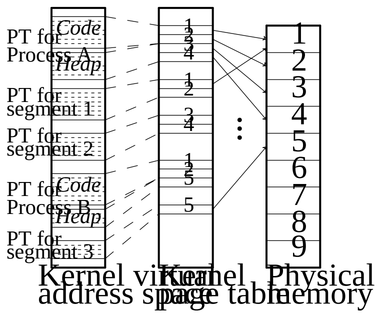

Figure 9. Implementing Sprite on a VAX. In this example Process A and Process B are sharing code but not heap. The page tables for each process and each of the three segments are kept in the kernel's virtual address space. The page tables are allocated such that the code and heap portions each begin on a page boundary. Since Processes A and B are sharing code, the kernel's page table is set up so that the code portion of each process's page table uses the same physical pages as those used by the page table for segment 1, the code segment that they are sharing. In addition the kernel's page table is set up so that the heap portion of each process's page table uses the same physical pages as the page table for the heap segment that they are using.

but it still has the problem that the code page tables of two processes that are sharing code must be kept consistent.

Sprite has not been implemented on a VAX. However, with a small addition to the segment structure described in Section 3 it can be implemented quite easily (see Figure 9). Each segment is allocated a page table that begins on a page boundary. In addition each process is allocated one page table for its code and heap which begins on a page boundary. The heap portion of the page table also begins on a page boundary. The page table entries in the kernel page table that map a process's page table are set up so that the code portion of a process's page table uses the same physical pages as the page table for the code segment that the process is using. Similarly the heap portion of a process's page table uses the same physical pages as the page table for the heap segment that the process is using. When a process begins executing, the P0BR register is set to point to the page table for the process and the P1BR is set to point to the page table that is associated with the stack segment. Thus there can be multiple virtual copies of segment page tables but only one physical copy.

The advantage of the Sprite scheme over the Unix scheme is that page tables for processes that are sharing segments automatically remain consistent. The disadvantage is that since the heap portion of a process's page table begins on a page boundary, heap segments must begin on 64K byte boundaries†. Since the virtual address space for a process is so large this should not be a problem.

A.2 Sun-2

Figure 10 shows a diagram of the memory mapping unit (MMU) provided by the Sun-2 architecture. The memory mapping mechanism is based on the idea of *contexts*. Each context is able to map the virtual address space of one process. Since there are N contexts and one is used to map the kernel, N - 1 processes can be mapped at once.

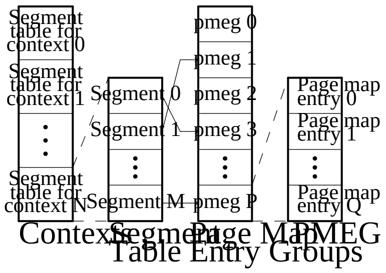

Figure 10. The Sun-2 memory mapping mechanism. The values for N, M, P and Q on a Sun-2 are 8, 512, 256, and 16 respectively. The Sun-3 architecture uses the same memory mapping mechanism as a Sun-2 with the exception that M is equal to 2048.

† Each VAX page is 512 bytes and each page table entry (PTE) is four bytes. This means that there are 128 PTEs per VAX page. Since each PTE maps 512 bytes, a VAX page full of PTEs maps 128 * 512 or 64K bytes.

Before a process begins executing on the hardware a special context register is set to indicate which context to use to translate the process's virtual addresses. A context is broken up into M segments of Q pages each. Each context contains a segment table with M entries each of which points to a page map entry group (PMEG) with Q entries, one for each page. There are a total of P PMEGs for the whole system. The following procedure is performed when translating a virtual address:

1) The context register is examined to determine which context to use.

2) The high order bits of the virtual address are used to index into the segment table for the context to determine which of the PMEGs to use. If there is no PMEG for the segment then a fault is generated.

3) The middle bits in the virtual address index into the PMEG to select one of the page map entries. Each page map entry contains a valid bit, a referenced bit, a modified bit, protection bits, and the physical page frame number. If the valid bit is not set, then a fault is generated.

4) Finally the physical page frame number is concatenated with the remaining low order bits of the virtual address to form the physical address.

Unlike the VAX, the Sun-2 MMU just described does not use page tables in the kernel's virtual address space; all of the memory mapping information is kept in hardware. However, since there are a small number of contexts (eight) and PMEGs (256) there is not sufficient hardware to map all processes at once. Therefore page tables need to be kept in software to store the state of processes or software segments† when they are not mapped in hardware. In addition the limited number of contexts and PMEGs must be multiplexed across all processes. The page table structure used by Sprite and Unix is just as described in Section 3. The remainder of this section describes the methods that Unix and Sprite use to manage the contexts and PMEGS.

In Unix contexts are shared by all processes by using an LRU policy: the processes that have contexts allocated to them are the ones that have run most recently. PMEGs are shared by all contexts using a FIFO policy. As a process faults in pages PMEGs are allocated to the context in order to map the pages. If multiple processes that are sharing code are mapped in contexts at the same time, as they fault in their code pages they will each use different PMEGs. Thus there is no attempt to share PMEGs between processes that are sharing code. Whenever a context is taken away from one process and given to another one any PMEGs that the context has allocated to it are freed.

In Sprite, like Unix, contexts are shared between processes by using an LRU policy and PMEGs are managed using a FIFO policy. However, unlike Unix PMEGs are shared between software segments instead of between contexts. When a page fault causes a PMEG to be allocated, in addition to putting a pointer to the PMEG in the hardware segment table, a pointer to the PMEG is stored in a table that is associated with the software segment that the page fault was in. This table of PMEGs is used to make sure that

---

For the remainder of this section, the segments that are used by the Sun hardware will be referred to as *hardware segments* and the code, heap, and stack segments used by Sprite will be refered to as *software segments*.

processes that are sharing a software segment will use the same PMEGs to map the software segment. When a context is removed from one process and given to another, instead of freeing any PMEGs in that context they remain allocated to the software segment that owns them. When a context is allocated to a process, the tables of PMEGs that are stored with each software segment that the process uses are copied into the hardware segment table.

The Sprite method of managing PMEGs and contexts has two advantages over the Unix method. First, a page fault in a software segment in one process will have the effect of validating the page for all processes that are sharing the software segment. Second, a process that has its context stolen from it may still have PMEGs allocated to it when it begins running again. The disadvantage of the Sprite scheme is that the hardware segment tables of all processes that are sharing software segments have to be kept consistent; if a PMEG is taken away from a software segment of a process in one context it must be removed from all contexts that are sharing the software segment.