SunOS Virtual Memory Implementation

Joseph P. Moran

Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View, CA 94043 USA

ABSTRACT

The implementation of a new virtual memory (VM) system for Sun's implementation of the UNIX† operating system (SunOS‡) is described. The new VM system was designed for extensibility and portability using an object-oriented design carefully constructed to not compromise efficiency. The basic implementation abstractions of the new VM system and how they are managed are described. Some of the more interesting problems encountered with a system based on mapped objects and the resolution taken to these problems are described.

1. Introduction

In December 1985 our group at Sun Microsystems began a project to replace our 4.2BSD-based VM system with a VM system engineered for the future. A companion paper [1] describes the general architecture of our new VM system, its goals, and its design rationale. To summarize, this architecture provides:

- Address spaces that are described by mapped objects.
- Support for shared or private (copy-on-write) mappings.
- Support for large, sparse address spaces.
- Page level mapping control.

We wanted the new VM system's implementation to reflect the clean design of its architecture and felt that basing the implementation itself on the proper set of abstractions would result in a system that would be efficient, extensible to solving future problems, readily portable to other hardware architectures, and understandable. Our group's earlier experience in implementing the *vnode* architecture [2] had shown us the utility of using object-oriented programming techniques as a way of devising useful and efficient implementation abstractions, so we chose to apply these techniques to our VM implementation as well.

The rest of this paper is structured as follows. Section 2 provides an overview of the basic object types that form the foundation of the implementation, and sections 3 through 6 describe these object types in detail. Sections 7 through 9 describe related changes made to the rest of the SunOS kernel. The most extensive changes were those related to the file system object managers. A particular file system type is used to illustrate those changes. Section 10 compares the performance of the old and new VM implementations. Sections 11 and 12 discuss conclusions and plans for future work.

2. Implementation Structure

The initial problem we faced in designing the new VM system's implementation was finding a clean set of implementation abstractions. The system's architecture suggested some candidate abstractions and examining the architecture with an eye toward carving it into a collection of objects suggested others.

---

† UNIX is a trademark of Bell Laboratories.
‡ SunOS is a trademark of Sun Microsystems.

We ultimately chose the following set of basic abstractions.

- The architecture allows for page-level granularity in establishing mappings from file system objects to virtual addresses. Thus the implementation uses the *page* structure to keep track of information about physical memory pages. The object managers and the VM system use this data structure to manage physical memory as a cache.
- The architecture defines the notion of an ‘‘address space’’. In the implementation, an *address space* consists of an ordered linked list of mappings. This level defines the external interface to the VM system and supplies a simple procedural interface to its primary client, the UNIX kernel.
- A *segment* describes a contiguous mapping of virtual addresses onto some underlying entity. The corresponding layer of the implementation treats segments as objects, acting as a class in the C++ [3] sense[1]. Segments can map several different kinds of target entities. The most common mappings are to objects that appear in the file system name space, such as files or frame buffers. Regardless of mapping type, the segment layer supplies a common interface to the rest of the implementation. Since there are several types of segment mappings, the implementation uses different *segment drivers* for each. These drivers behave as subclasses of the segment class.
- The *hardware address translation* (*hat*) layer is the machine dependent code that manages hardware translations to pages in the machine’s memory management unit (MMU).

The VM implementation requires services from the rest of the kernel. In particular, it makes heavy demands of the *vnode* [2] object manager. The implementation expects the *vnode* drivers to mediate access to pages comprising file objects. The part of the *vnode* interface dealing with cache management changed drastically. Finding the right division of responsibility between the segment layer and the *vnode* layer proved to be unexpectedly difficult and accounted for much of the overall implementation effort.

The new VM system proper has no knowledge of UNIX semantics. The SunOS kernel provides UNIX semantics by using the VM abstractions as primitive operations [1]. Figure 1 is a schematic diagram of the VM abstractions and how they interact. The following sections describe in more detail the implementation abstractions summarized above.

3. *page* Structure

The new VM architecture treats physical memory as a cache for the contents of memory objects. The *page* is the data structure that contains the information that the VM system and object managers need to manage this cache. The *page* structure maintains the identity and status of each page of physical memory in the system. There is one *page* structure for every interesting[2] page in the system.

A *page* represents a system page size unit of memory that is a multiple of the hardware page size. The memory page is identified by a <*vnode*, offset> pair kept in the *page* structure. Each page with an identity is initialized to the contents of a page’s worth of the *vnode*’s data starting at the given byte offset. A hashed lookup based on the <*vnode*, offset> pair naming the page is used to find a page with a particular name. The implementation keeps all pages for a given *vnode* on a doubly-linked list rooted at the *vnode.* Maintaining this list speeds operations that need to find all a *vnode*’s cached pages. *page* structures can also be on free lists or on an ‘‘I/O’’ list depending on the setting of page status flags. The *page* structure also contains an opaque pointer that the *hat* layer uses to maintain a list of all the active translations to the page that are loaded in the hardware. In the machine independent VM code above the *hat* layer, the only use for this opaque pointer is to test for NULL to determine if there are any active translations to the page. When the machine-dependent *hat* layer unloads a translation it retrieves the hardware reference and modified bits for that translation to the page, and merges them into machine-independent versions of these bits maintained in the *page* structure.

---

1 Actually, as a class whose public fields are all virtual, so that subclasses are expected to define them.

2 Pages for kernel text and data and for frame buffers are not considered ‘‘interesting’’.

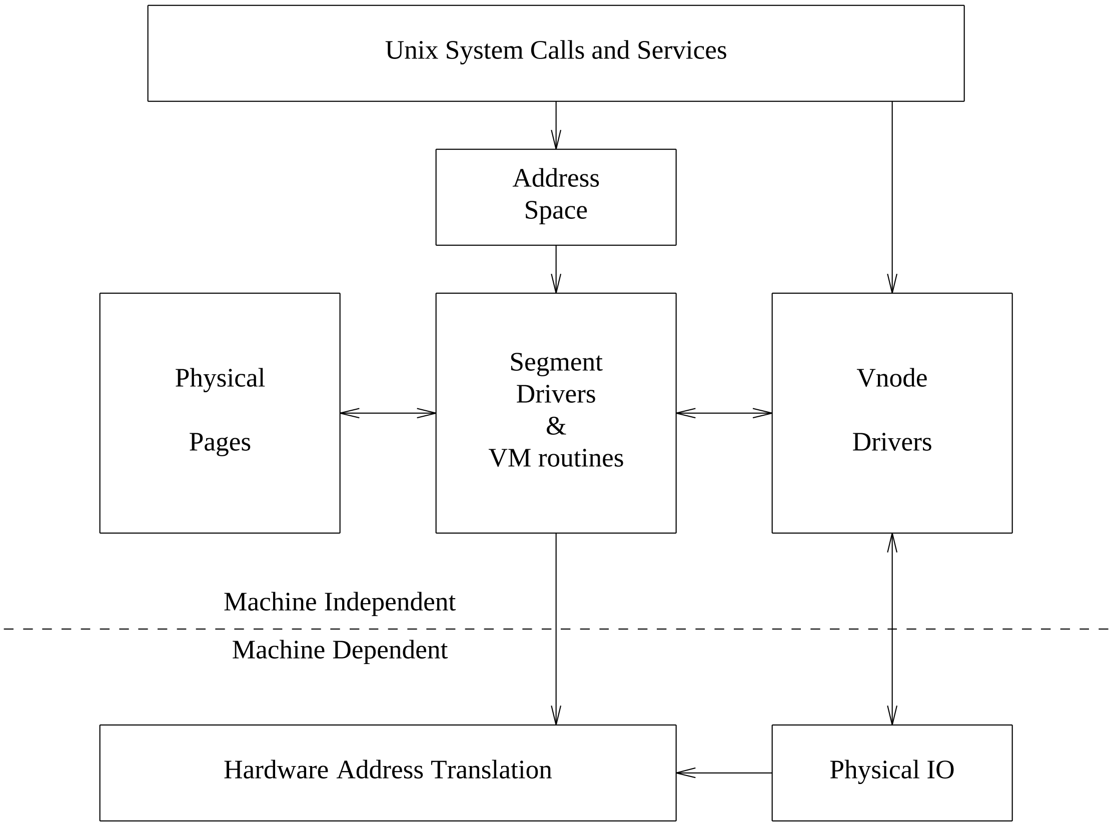

Figure 1

4. Address Space

The highest level abstraction that the VM system implements is called an *address space* (*as*), which consists of a collection of mappings from virtual addresses to underlying objects such as files and display device frame buffers. The *as* layer supports a procedural interface whose operations fall into two basic classes. Procedures in the first class manipulate an entire address space and handle address space allocation, destruction, duplication, and ‘‘swap out’’. Procedures in the second class manipulate a virtual address range within an address space. These functions handle fault processing, setting and verifying protections, resynchronizing the contents of an address space with the underlying objects, obtaining attributes of the mapped objects, and mapping and unmapping objects. Further information on these functions may be found in [1].

The implementation must maintain state information for each address space. The heart of this information is a doubly linked list of contiguous mappings (termed *segments* for lack of a better name) sorted by virtual address. Section 5 describes segments in detail. The *as* layer implements its procedural interface by iterating over the required range of virtual addresses and calling the appropriate segment operations as needed.

In addition, the *as* structure contains a *hardware address translation* (*hat*) structure used to maintain implementation specific memory management information. Positioning the *hat* structure within the *as* structure allows the machine dependent *hat* layer to describe all the physical MMU mappings for an address space, while the machine independent *as* layer manages all the virtual address space mappings. The *hat* structure is opaque to the machine independent parts of the system and only the *hat* layer examines it. Section 6 describes the *hat* layer in detail. The *as* structure also includes machine independent address space statistics that are kept separately from the machine dependent *hat* structure for convenience.

4.1. Address Space Management

The implementation uses several techniques to reduce the overhead of *as* management. To reduce the time to find the segment for a virtual address, it maintains a ‘‘hint’’ naming the last segment found, in a manner similar to the technique used in Mach [4]. Any time the *as* layer translates a virtual address to a segment, this hint is used as the starting point to begin the search.

Another optimization reduces the total number of segments in a given address space by allowing segment drivers to coalesce adjacent segments of similar types. This reduces the average time to find the segment that maps a given virtual address within an address space. By using this technique, the common UNIX *brk*(2) system call normally reduces to a simple segment extension within the process address space.

4.2. Address Space Usage

SunOS uses an *as* to describe the kernel’s own address space, which is shared by all UNIX processes when operating in privileged (supervisor) mode. A UNIX process typically has an *as* to describe the address space it operates in when in non-privileged (user) mode[3]. An *as* is an abstraction that exists independent of any of its uses. Just as several UNIX processes share the same kernel address space when operating in supervisor mode, an *as* can have multiple threads of control active in a user mode address space at the same time. Future implementations of the operating system will take advantage of these facilities [5].

Most UNIX memory management system calls map cleanly to calls on the *as* layer. The *as* layer does not have knowledge of the implementation of the segment drivers below it, thus making it easy to add new segment types to the system. The *as* design provides support for large sparse address spaces without undue penalty for common cases, an important consideration for the future software demands that will be placed on the VM system.

5. Segments

A *segment* is a region of virtual memory mapped to a contiguous region of a memory object[4]. Each segment contains some public and private data and is manipulated in an object-oriented fashion. The public data includes the base and size of the segment in page-aligned bytes, pointers for the next and previous segments in the address space, and a pointer to the *as* structure itself. Each segment also contains a reference to a vector of pointers to operations (an ‘‘ops’’ vector) that implement a set of functions similar to the *as* functions, and a pointer to a private per-segment type data structure. This is similar to the way the SunOS *vnode* and *vfs* abstractions are implemented [2]. Using this style of interface allows multiple segment types to be implemented without affecting the rest of the system.

To most efficiently handle its data structures, a segment driver is free to coalesce adjacent segments of the same type in the virtual address space or even to break a segment down into smaller segments. Individual virtual pages within a segment’s mappings may have varying attributes (e.g. protections). This design allows the segment abstraction control over the attributes and data structures it manages.

Of equal importance to what a segment driver does is what it does not do. In particular, we found that having the segment driver handle the page lookup operation and call the *vnode* object manager only when a needed page cannot be found was a bad idea. After running into some problems that could not be solved as a result of this split, we restructured the VM system so that the segment driver always asks the object manager for the needed page on each fault. Having the *vnode* object manager be responsible for the page lookup operation allows it to take action on each new reference.

5.1. Segment Driver Types

The implementation includes the following segment driver types:

seg_vn Mappings to regular files and anonymous memory.

---

[3] Some processes run entirely in the kernel and have no need for a user mode address space.

[4] Note that the name ‘‘segment’’ is not related to traditional UNIX text, data, and stack segments.

seg_map Kernel only transient <*vnode*, offset> translation cache.

seg_dev Mappings to character special files for devices (e.g. frame buffers).

seg_kmem Kernel only driver used for miscellaneous mappings.

The seg_vn and seg_map segment drivers manage access to *vnode* memory objects and are the primary segment drivers.

5.2. *vnode* Segment

The **seg_vn** *vnode* segment driver provides mappings to regular files. It is the most heavily used segment driver in the system.

The arguments to the segment create function include the *vnode* being mapped, the starting offset, the mapping type, the current page protections, and the maximum page protections. The mapping type can be shared or private (copy-on-write). With a shared mapping, a successful memory write access to the mapped region will cause the underlying file object to be changed. With a private mapping the first write access to a page of the mapped region will cause a copy-on-write operation that creates a private page and initializes it to a copy of the original page.

The UNIX *mmap* (2) system call, which sets up new mappings in the process's user address space, calculates the maximum page protection value for a shared mapping based on the permissions granted on the *open* of the file. Thus, the *vnode* segment driver will not allow a file to be modified through a mapping if the file was originally opened read-only.

5.2.1. Anonymous Memory

An important aspect of the VM system is the management of ''anonymous'' pages that have no permanent backing store. An anonymous page is created for each copy-on-write operation and for each initial fault to the anonymous clone object[5]. For a UNIX executable, the uninitialized data and stack are set up as private mappings to the anonymous clone object.

The mechanism used to manage anonymous pages has been isolated to a set of routines that provide a service to the rest of the VM system. Segment drivers that choose to implement private mappings use this service. The *vnode* segment driver is the primary user of anonymous memory objects.

5.2.1.1. Anonymous Memory Data Structures

The *anon* structure serves as a name for each active anonymous page of memory. This structure introduces a level of indirection for access to anonymous pages. We do not wish to assume that anonymous pages can be named by their position in a storage device, since we would like to be able to have anonymous pages in memory that haven't been allocated swap space. The *anon* data structure is opaque above the anonymous memory service routines and is operated on using a procedural interface in an object-oriented fashion. These objects are reference counted, since there can be more than one reference to an anonymous page[6]. This reference counting allows the *anon* procedures to easily detect when an anonymous page and corresponding resident physical page (if any) are no longer needed.

The other data structure related to anonymous memory management is the *anon_map* structure. This structure describes a cluster of anonymous pages as a unit. The *anon_map* structure consists of an array of *anon* structure pointers with one *anon* pointer per page. Segment drivers that wish to refer to anonymous pages do so by using an *anon_map* structure to keep an array of pointers to *anon* structures for the anonymous pages. These segment drivers lazily allocate an *anon_map* structure with NULL *anon* structure pointers at fault time as needed (i.e., on the first copy-on-write for the segment or on the first fault for an all anonymous mapping).

---

[5] The name of this object in the UNIX file system name space is **/dev/zero**.

[6] Typically from an address space duplication resulting from a UNIX *fork* (2) system call.

5.2.1.2. Anonymous Memory Procedures

There are two *anon* procedures that operate on the arrays of *anon* structure pointers in the *anon_map* structure. *anon_dup* () copies from one *anon* pointer array to another one, incrementing the reference count on every allocated *anon* structure. This operation is used when a private mapping involving anonymous memory is duplicated. The converse of *anon_dup* () is *anon_free* (), which decrements the reference count on every allocated *anon* structure. If a reference count goes to zero, the *anon* structure and associated page are freed. *anon_free* () is used when part of a privately mapped anonymous memory object is unmapped.

There are three *anon* procedures used by the fault handlers for anonymous memory objects. *anon_private* () allocates an anonymous page, initializing it to the contents of the previous page loaded in the MMU. *anon_zero* () is similar to *anon_private* (), but initializes the anonymous page to zeroes. This routine exists as an optimization to avoid having to copy a page of zeroes with *anon_private* (). Finally, *anon_getpage* () retrieves an anonymous page given an *anon* structure pointer.

5.2.2. *vnode* Segment Fault Handling

Page fault handling is a central part of the new VM system. The fault handling code resolves both hardware faults (e.g., hardware translation not valid or protection violation) and software pseudo-faults (e.g., lock down pages). The *as* fault handing routine is called with a virtual address range, the fault type (e.g., invalid translation or protection violation), and the type of attempted access (read, write, execute). It performs a segment lookup operation based on the virtual address and dispatches to the segment driver's fault routine, which is responsible for resolving the fault.

The *vnode* segment driver takes the following steps to handle a fault.

- Verify page protections.
- If needed, allocate an *anon_map* structure.
- If needed, get the page from the object manager.
  Call the *hat* layer to load a translation to the page.
- If needed, obtain a new page by performing copy-on-write.
  Call the *hat* layer to load a writable translation to the new page.

Some specific examples of *vnode* segment fault handling and how anonymous memory is used are given below.

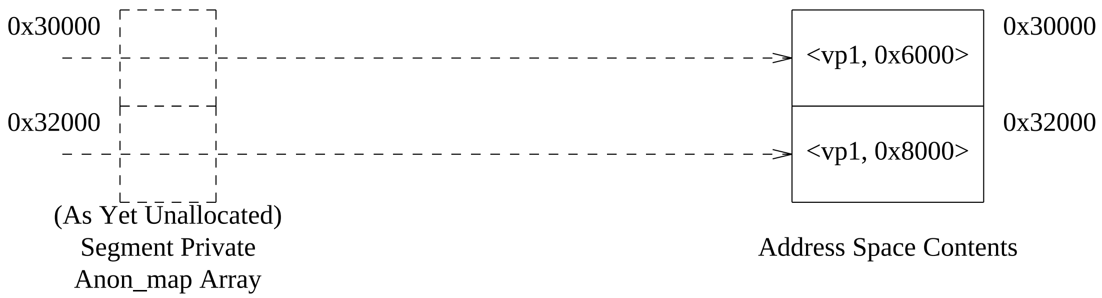

Figure 2
<vp1, 6000> Mapped Private to Address 0x30000 for 0x4000 Bytes

Figure 2 depicts a private mapping from offset 0x6000 in *vnode* vp1 to address 0x30000 for a length of 0x4000 bytes using a system page size of 0x2000. If a non-write fault occurs on an address within the segment, the *vnode* segment driver asks the *vnode* object manager for the page named by <vp1, 0x6000 + (*addr* - 0x30000)>. The *vnode* object manager is responsible for creating and initializing the page when requested to do so by a segment driver. After obtaining the page, the *vnode* segment driver calls the *hat* layer to load a translation to the page. The permissions passed to the *hat* layer from the *vnode* segment driver are for a read-only translation since this is a private mapping for which we want to catch a memory write operation to initiate a copy-on-write operation.

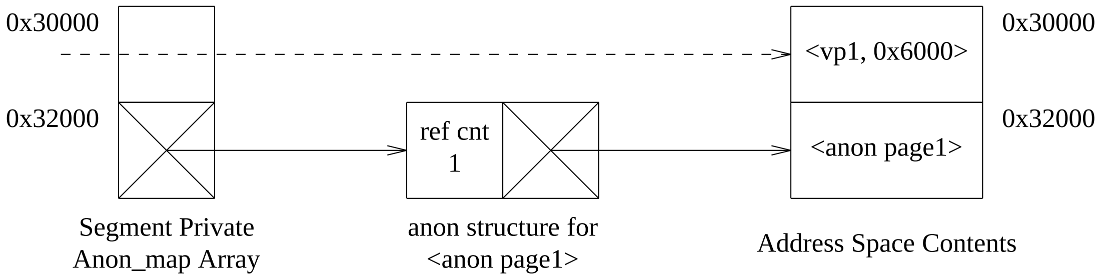

Figure 3
After Copy-On-Write Operation to Address 0x32000

Figure 3 shows the results of a copy-on-write operation on address 0x32000 in Figure 2. The *vnode* segment driver has allocated an *anon_map* structure and initialized the second entry to point to an allocated *anon* structure that initially has a reference count of one. The *anon_private*() routine has allocated the *anon* structure in the array, returned a page named by that *anon* structure, and initialized to the contents of the previous page at 0x32000. After getting the anonymous page from *anon_private*(), the *vnode* segment driver calls the *hat* layer to load a writable translation to the newly allocated and initialized page.

Note that as an optimization, the *vnode* segment driver is able to perform a copy-on-write operation, even if the original translation was invalid, since the fault handler gets a fault type parameter (read, write, execute). If the first fault taken in the *segment* described in Figure 3 is a write fault at address 0x32000 then the first operation is to obtain the page for <vp1, 0x8000> and call the *hat* layer to load a read-only translation. The *vnode* segment driver can then detect that it still needs to perform the copy-on-write operation because the fault type was for a write access. If the copy-on-write operation is needed, the *vnode* segment driver will call *anon_private*() to create a private copy of the page.

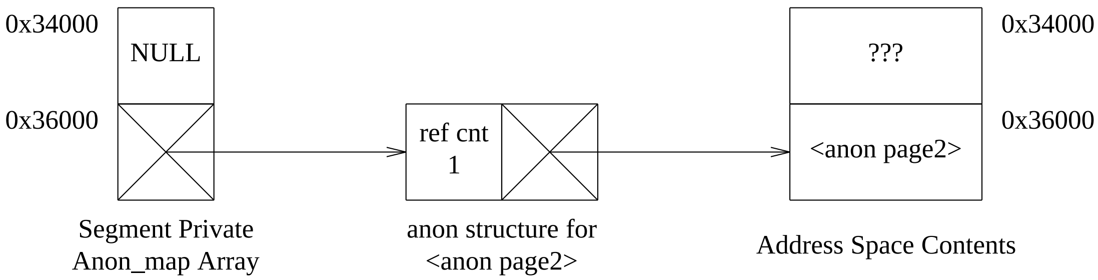

Figure 4
Private Mapping to /dev/zero for 0x4000 bytes at Address 0x34000
After a Page Fault at Address 0x36000

Figure 4 depicts a private mapping from the anonymous clone device **/dev/zero** to address 0x34000 for length 0x4000 after a page fault at address 0x36000. Since there is no primary *vnode* that was mapped, the *vnode* segment driver calls *anon_zero*() to allocate an *anon* structure and corresponding page and initialize the page to zeroes.

Figure 5 shows what happens when the mapped private *vnode* segment shown in Figure 4 is duplicated. Here both segments have a private reference to the same anonymous page. When the segment is duplicated, *anon_dup*() is called to increment the reference count on all the segment's allocated *anon* structures. In this example, there is only one allocated *anon* structure and its reference count has been incremented from one to two. Also, as part of the *vnode* segment duplication process for a privately mapped segment, all the *hat* translations are changed to be read-only so that previously writable anonymous pages are now set up for copy-on-write.

Figure 6 shows the result after the duplicated segment in Figure 5 handles a write fault at address 0x36000. When the segment fault handler calls *anon_getpage*() to return the page for the given *anon*

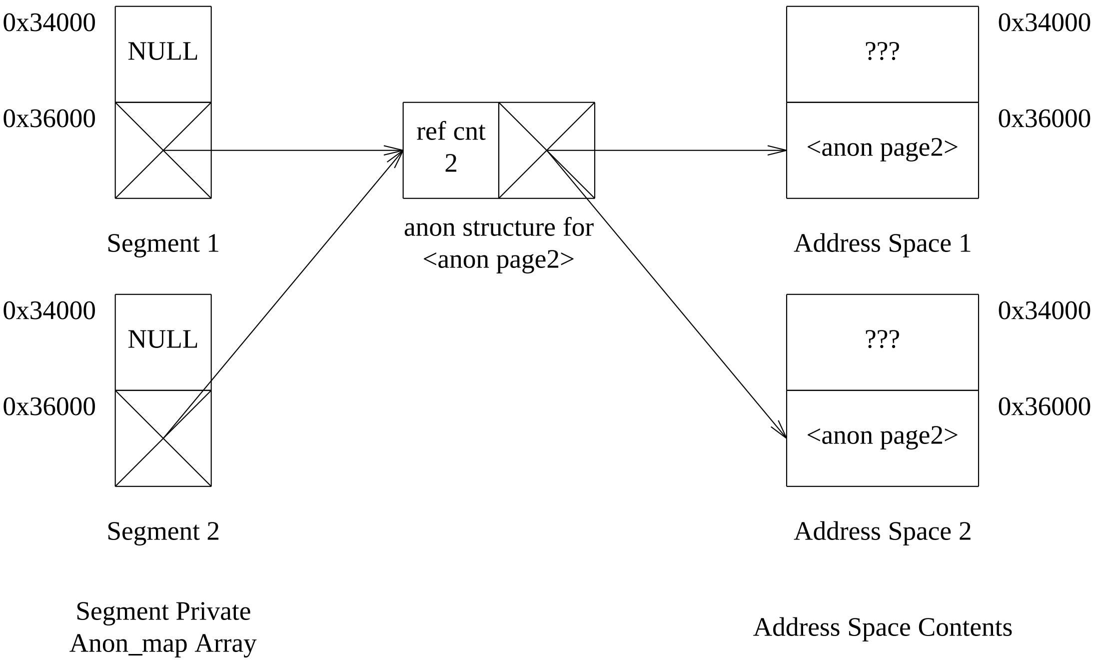

Figure 5
After the Private Mapped Vnode Segment is Duplicated

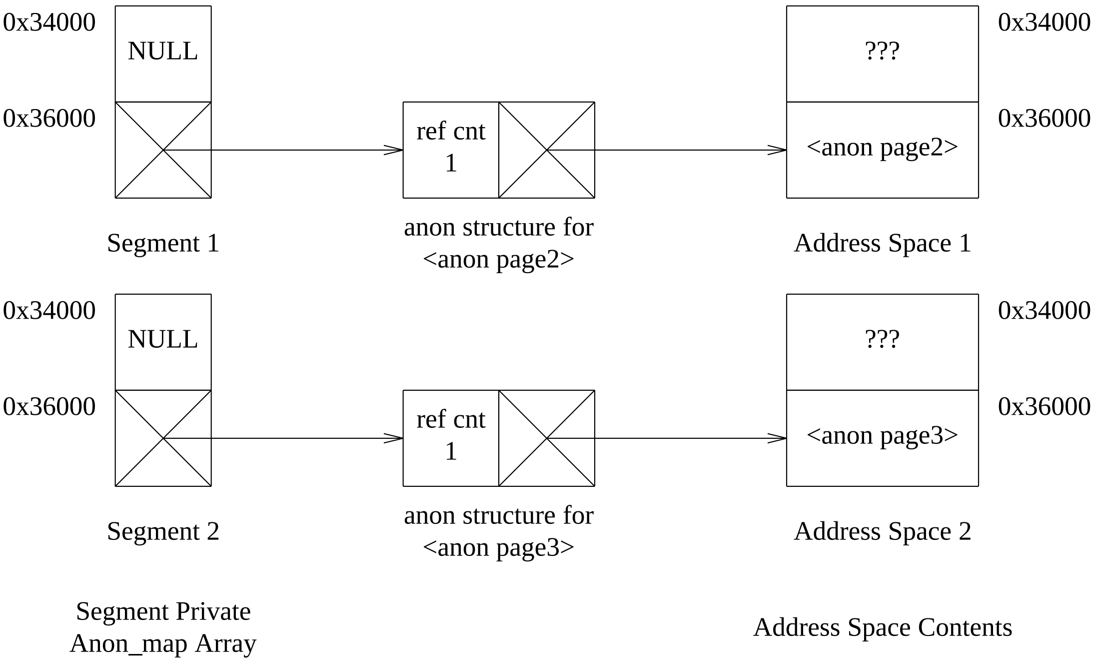

Figure 6
After Write Fault on Address 0x36000 in Address Space 2

structure, it will return protections that force a read-only translation since the reference count on the *anon* structure is greater than one. The segment driver fault handler will then call *anon_private*() to allocate a new *anon* structure and *page* structure and to initialize the page to the contents of the previous page loaded in the MMU. In contrast to the case depicted in Figure 3, *anon_private*() is copying from another anonymous page and will decrement the reference count of the old *anon* structure after the *anon* pointer in the segment's *anon_map* array is changed to point to the newly allocated *anon* structure. Since the reference count on the original *anon* structure reverts to one, this means that the original segment will no longer have to do a copy-on-write operation for a subsequent write fault at address 0x36000. If a fault were to occur at 0x36000 in the original segment, *anon_getpage*() would not enforce a read-only mapping, since the reference count for the *anon* structure is now one.

5.3. Kernel Transient *vnode* Mapping Segment

The **seg_map** segment driver is a driver the kernel uses to get transient <*vnode*, offset> mappings. It supports only shared mappings. The most important service it provides to the *as* layer is fault resolution for kernel page faults. The seg_map driver manages a large window of kernel virtual space and provides a view onto a varying subset of the system's pages. The seg_map driver manages its own virtual space as a cache, so that recently referenced <*vnode*, offset> pairs are likely to be loaded in the MMU and no page fault will be taken when the virtual address within the seg_map segment are referenced.

This segment driver provides fast map and unmap operations using two segment driver-specific subroutines: *segmap_getmap*() and *segmap_release*(). Given a <*vnode*, offset> pair, *segmap_getmap*() returns a virtual address within the seg_map segment that is initialized to map part of the *vnode*. This is similar to the traditional UNIX *bread*() function used in the ''block IO system'' to obtain a buffer that contains some data from a block device. The *segmap_release*() function takes a virtual address returned from *segmap_getmap*() and handles releasing the mapping. *segmap_release*() also handles writing back modified pages. *segmap_release*() performs a similar function to the traditional UNIX *brelse*() / *bdwrite*() / *bwrite*() / *bawrite*() ''block IO system'' procedures depending on the flags given to *segmap_release*().

The seg_map driver is simply used as an optimization in the kernel over the standard *vnode* driver. It is important to be able to do fast map and unmap operations in the kernel to implement *read*(2) and *write*(2) system calls. The basic algorithm for the *vnode* read and write routines is to get a mapping to the file, copy the data from/to the mapping, and then unmap the file. Note that the kernel accesses the file data just as user processes do by using a mapping to the file. The *vnode* routines that implement read and write use *segmap_getmap*() and *segmap_release*() to provide the fast map and unmap operations within the kernel's address space.

5.4. Device Driver Segment

The **seg_dev** segment driver manages objects controlled by character special (''raw'') device drivers that provide an *mmap* interface. The most common use of the seg_dev driver is for mapped frame buffers, though it is also used for mappings to machine-specific memory files such as physical memory, kernel virtual memory, Multibus memory, or VMEbus memory. This driver currently only supports shared mappings and does not deal with anonymous private memory pages. The driver is simple since it doesn't have to worry about a many operations that don't make sense for these types of objects (e.g., swap out). To resolve a fault, it simply calls a function to return an opaque ''cookie'' from the device driver, which is then handed to the machine-specific *hat* layer to load a translation to the physical page denoted by the cookie.

5.5. Kernel Memory Segment

The **seg_kmem** segment driver is an example of the use of a machine independent concept to solve a machine dependent problem. The kernel's address space is described by an *as* structure just like the user's address space. The seg_kmem segment driver is used as a catch-all to map miscellaneous entities into the kernel's address space. These entities includes the kernel's text, data, bss, and dynamically allocated memory space. This driver also manages other machine dependent portions of the kernel's address space (e.g. Sun's Direct Virtual Memory Access space [6]).

The seg_kmem driver currently only supports non-paged memory whose MMU translations are always locked[7]. In the previous 4.2BSD-based VM system, the management of the kernel's address space

[7] This means that the *hat* layer cannot remove any of these translations without explicitly being told to do so by the seg_kmem driver.

for things like device registers was done by calls to a *mapin*() procedure that set up MMU translations using a machine-dependent page table entry. For kernel and driver compatibility reasons, the seg_kmem driver supports a *mapin*-like interface as a set of segment driver-specific procedures.

6. Hardware Address Translation Layer

The *hardware address translation* (*hat*) layer is responsible for managing the machine dependent memory management unit. It provides the interface between the machine dependent and the machine independent parts of the VM system. The machine independent code above the *hat* layer knows nothing about the implementation details below the *hat* layer. The clean separation of machine independent and dependent layers of the VM system allows for better understandability and faster porting to new machines with different MMUs.

The *hat* layer exports a set of procedures for use by the machine independent segment drivers. The higher levels cannot look at the current mappings, they can only determine if any mappings exist for a given page. The machine independent levels call down to the *hat* layer to set up translations as needed. The basic dependency here is the ability to handle and recover from page faults (including copy-on-write). The *hat* layer is free to remove translations as it sees fit if the translation was not set up to be locked. There exists a call back mechanism from the hat layer to the segment driver so that the virtual reference and modified bits can be maintained when a translation is take away by the *hat* layer. This ability is needed for alternate paging techniques in which per address space management of the working set is done.

6.1. *hat* Procedures

Table 1 lists the machine independent *hat* interfaces. All these procedures must be provided, although they may not necessarily do anything if not required by the *hat* implementation for a given machine.

Operation	Function
`hat_init()`	One time *hat* initialization.
`hat_alloc(as)`	Allocate *hat* structure for *as*.
`hat_free(as)`	Release all *hat* resources for *as*.
`hat_pageunload(pp)`	Unload all translations to page *pp*.
`hat_pagesync(pp)`	Sync ref and mod bits to page *pp*.
`hat_unlock(seg, addr)`	Unlock translation at *addr*.
`hat_chgprot(seg, addr, len, prot)`	Change protection values.
`hat_unload(seg, addr, len)`	Unload translations.
`hat_memload(seg, addr, pp, prot, flags)`	Load translation to page *pp*.
`hat_devload(seg, addr, pf, prot, flags)`	Load translation to cookie *pf*.

Table 1
***hat* operations**

6.2. *hat* Implementations

Several *hat* implementations have already been completed. The first implementations were for the Sun MMU [6]. The MMUs in the current Sun-2/3/4 family are quite similar. All use a fixed number of context, segment, and page table registers in special hardware registers to provide mapping control. The Sun-2 MMU has a separate context when running in supervisor mode whereas the Sun-3 and Sun-4 MMUs have the kernel mapped in each user context. The maximum virtual address space for the Sun-2, Sun-3, and Sun-4 MMUs are 16 megabytes, 256 megabytes, and 4 gigabytes respectively.

Some machines in the Sun-3 and Sun-4 families use a virtual address write-back cache. The use of a virtual address cache allows for faster memory access time on cache hits, but can be a cause of great trouble to the kernel in the old VM system [7]. Since the *hat* layer has information about all the translations to a given page, it can manage all the details of the virtual address cache. It can verify the current virtual address alignment for the page and decide to trade translations if an attempt to load a non-cache consistent address occurs. In the old 4.2BSD-based VM system the additional support needed for the virtual address cache permeated many areas of the system. Under the new VM system, support for the virtual address cache is isolated within the *hat* layer.

Other *hat* implementations have been done for more traditional page table-based systems. The Motorola 68030 has a flexible on-chip MMU. The *hat* layer chooses to manage it using a three level page table to support mapping a large sparse virtual address space with minimal overhead. The Intel 80386 also has an on-chip MMU, but it has a fixed two level translation scheme of 4KB pages. The problem with the 80386 MMU is that the kernel can write all pages regardless of the page protections (i.e., the write protection only applies to non-supervisor mode accesses)! This means that explicit checks must be performed for kernel write accesses to read-only pages so that kernel protection faults can be simulated. Another implementation has been done for IBM 370/XA compatible main frames. The biggest problem with this machine's architecture for the new VM system is that an attempted write access to a read-only page causes an protection exception that can leave the machine in an unpredictable state for certain instructions that modify registers as a side effect. These instructions cannot be reliably restarted thus breaking copy-on-write fault handling. The implementation resorts to special work arounds for the few instructions that exhibit this problem[8].

7. File System Changes

The VM system required changes to several other parts of the SunOS kernel. The VM system relies heavily on the *vnode* object managers, and required changes to the *vnode* interface as well as to each *vnode* object type implementation. It took us several attempts to get the new *vnode* interface right.

Our initial attempt gave the core VM code responsibility for all decisions about operations it initiated. We repeatedly encounted problems induced by not having appropriate information available within the VM code at sites where it had to make decisions, and realized that the proper approach was to make decisions at locations possessing the requisite information. The primary effect of this shift in responsibility was to give the *vnode* drivers control on each page reference. This allows the *vnode* drivers to recognize and act on each new reference. These actions include validating the page, handling any needed backing store allocation, starting read-ahead operations, and updating file attributes.

7.1. File Caching

Traditionally, buffers in the UNIX buffer cache have been described by a device number and a physical block number on that device. This use of physical layout information requires all file system types implemented on top of a a block device to translate (*bmap*) each logical block to a physical block on the device before it can be looked up in the buffer cache.

In the new VM system, the physical memory in the system is used as a *logical* cache; each buffer (page) in the cache is described by an object name (*vnode*) and a (page-aligned) offset within that object. Each file is named as a separate *vnode,* so the VM system need not have any knowledge of the way the *vnode* object manager (file system type) stores the *vnode.* A segment driver simply asks the *vnode* object manager for a range of logical pages within the *vnode* being mapped. The file system independent code in the segment drivers only has to deal with offsets into a *vnode* and does not have to maintain any file system-specific mapping information that is already kept in the file system-specific data structures. This provides a much cleaner separation between the segment and *vnode* abstractions and puts few constraints on the implementation of a *vnode* object manager[9].

The smallest mapping unit relevant to the VM system is a system page. However, the system page size is not necessarily related to the block sizes that a file system implementation might use. While we

[8] Such instructions are highly specialized and the standard compilers never generate them.

[9] We have taken advantage of this and have implemented several interesting *vnode* object managers that are nothing like typical file systems.

could have implemented a new file system type that used blocks that were the same size as a system page, and only supported file systems that had this attribute, we did not feel this was an acceptable approach. We needed to support existing file systems with widely varying block size. We also did not feel that it was appropriate to use only one system page size across a large range of machines of varying memory size and performance. We decided it was best to push the handling of block size issues into each file system implementation, since the issues would vary greatly depending on the file system type.

The smallest allocatable file system block is potentially smaller than the system page size, while the largest file system block may be much larger than the system page size. The *vnode* object manager must initialize each page for a file to the proper contents. It may do this by reading a single block, multiple blocks, or possibly part of a block, as necessary. If the size of the file is not a multiple of the system page size, the *vnode* object manager must handle zeroing the remainder of the page past the end of the file.

Using a logical cache doesn't come without some cost. When trying to write a page back to the file system device, the VOP_PUTPAGE routine (discussed below) may need to map the logical page number within the object to a physical block number, or perhaps to a list of physical block numbers. If the file system-specific information needed to perform the mapping function is not present in memory, then a read operation may be required to get it. This complicates the work the page daemon must do when writing back a dirty page. File system implementations need to be careful to prevent the page daemon from deadlocking waiting to allocate a page needed for a *bmap*-like operation while trying to push out a dirty page when there are no free pages available.

7.2. *vnode* Interface Changes

We defined three new *vnode* operations for dealing with the new abstractions of mappings in address spaces and pages. These new *vnode* operations replaced ones that dealt with the old buffer cache and the 4.2BSD-based VM system [2]. The primary responsibility of the *vnode* page operations is to fill and drain physical pages (page-in and page-out). It also provides an opportunity for the managers of particular objects to map the page abstractions to the representation used by the object being mapped.

The VOP_MAP() routine is used by the *mmap* system call and is responsible for handling file system dependent argument checking, as well as setting up the requested mapping. After checking parameters it uses two address space operations to do most of the work. Any mappings in the address range specified in the *mmap* system call are first removed by using the *as_unmap*() routine. Then the *as_map*() routine establishes the new mapping in the given address space by calling the segment driver selected by the *vnode* object manager.

The VOP_GETPAGE() routine is responsible for returning a list of pages from a range of a *vnode.* It typically performs a page lookup operation to see if the pages are in memory. If the desired pages are not present, the routine does everything needed to read them in and initialize them. It has the opportunity to perform operations appropriate to the underlying *vnode* object on each fault, such as updating the reference time or performing validity checks on cached pages.

As an optimization, the VOP_GETPAGE() routine can return extra pages in addition to the ones requested. This is appropriate when a physical read operation is needed to initialize the pages and the *vnode* object manager tries to perform the I/O operation using a size optimal for the particular object. Before this is done the segment driver is consulted, using a ''kluster'' segment function, so that the segment driver has the opportunity to influence the *vnode* object manager's decisions. The VOP_GETPAGE() routine also handles read-ahead if it detects a sequential access pattern on the *vnode.* It uses the same segment kluster function to verify that the segment driver believes that it would be worthwhile to perform the read-ahead operation. The I/O klustering and read-ahead conditions allow both the *vnode* object manager and the segment driver controlling a mapping onto this object to have control over how these conditions are handled. Thus, for these conditions we have set up our object-oriented interfaces to allow distributed control among different abstractions that have different pieces of knowledge about a particular problem. The *vnode* object manager has knowledge about preferred I/O size and reference patterns to the underlying object, whereas the segment driver has the knowledge about the view established to this object and may have advice passed in from above the address space regarding the expected reference pattern to the virtual address space.

The other new *vnode* operation for page management is VOP_PUTPAGE(). This operation is the complement of VOP_GETPAGE() and handles writing back potentially dirty pages. A flags parameter controls whether the write back operation is performed asynchronously and whether the pages should be invalidated after being written back.

The VOP_GETPAGE() and VOP_PUTPAGE() interfaces deal with offsets and pages in the logical file. No information about the physical layout of the file is visible above the *vnode* interface. This means that the work of translating from logical blocks to physical disk blocks (the *bmap* function) is all done within the *vnode* routines that implement the VOP_GETPAGE() and VOP_PUTPAGE() interfaces. This is a clean and logical separation of the file object abstractions from the VM abstractions and contrasts with the old 4.2BSD-based implementation where the physical locations of file system blocks appeared in VM data structures.

8. UFS File System Rework

Another difficult issue pertinent to the conversion to a memory-mapped, page-based system is how to convert existing file systems. The most notable of these in SunOS is the 4.2BSD file system [8], which is known in SunOS as the UNIX File System (UFS). The relevant characteristics of this file system type include support for two different blocking sizes (a large basic block size for speed, and a smaller fragment size to avoid excessive disk waste), the ability to have unallocated blocks (‘‘holes’’) in the middle of a file which read back as zeroes, and the need to *bmap* from logical blocks in the file to physical disk blocks.

8.1. Sparse UFS File Management

ufs_getpage() is the UFS routine that implements the VOP_GETPAGE() interface. When a fault occurs on a UFS file, the segment driver fault routine calls this routine, passing it the type of the attempted access (e.g., read or write access). It uses this access type information to determine what to do if the requested page corresponds to an as yet unallocated section of a sparse file. If a write access to one of these holes in the file is attempted, *ufs_getpage*() will attempt to allocate the needed block(s) of file system storage. If the allocation fails because there is no more space available in the file system, or the user process has exceeded its disk quota limit, *ufs_getpage*() returns the error back to the calling procedure which then propagates back to the caller of address space fault routine.

When *ufs_getpage*() handles a read access to a page that does not have all its disk blocks allocated, it zeroes out the part of the page that is not backed by an allocated disk block and arranges for the segment driver requesting the page to establish a read-only translation to it. Thus no allocation is done when a process tries to read a hole from a UFS file. However, an attempted write access to such a page causes a protection fault and *ufs_getpage*() can perform the needed file system block allocation as previously described.

8.2. UFS File Times

Another set of problems resulted from handling the file access and modified times. The obvious way to handle this problem is to simply update the access time in *ufs_getpage*() any time a page is requested and to update the modification time in *ufs_putpage*() any time a dirty page is written back. However, this approach has some problems.

The first problem is that the UFS implementation has never marked the access time when doing a *write* to the file[10]. The second problem is related to getting the correct modification time when writing a file. When doing a *write*(2) system call, the file is marked with the current time. When dirty pages created by the *write* operation are actually pushed back to backing store in *ufs_putpage*(), we don’t want to override the modification time already stored in the *inode*[11].

To solve these problems, *inode* flags are set in the ‘‘upper layers’’ of the UFS code (e.g., when doing a file read or write operation) and examined in the ‘‘lower layers’’ of the UFS code (*ufs_getpage*() and *ufs_putpage*()). *ufs_getpage*() examines the *inode* flags to determine whether to update the *inode*’s access

[10] The ‘‘read’’ that is sometimes needed to perform a *write* operation never causes the file’s access time to be updated.

[11] The *inode* is the file system private *vnode* information used by the UFS file system [2].

time based on whether a read or write operation is currently in progress. *ufs_putpage*() can use the *inode* flags to determine whether it needs to update the *inode*'s modification time based on whether the modification time has been set since the last time the *inode* was written back to disk.

8.3. UFS Control Information

Another difficult issue related to the UFS file system and the VM system is dealing with the control information that the *vnode* driver uses to manage the logical file. For the UFS implementation, the control information consists of the *inodes,* indirect blocks, cylinder groups, and super blocks. The control information is not part of the logical file and thus the control information still needs to be named by the block device offsets, not the logical file offsets. To provide the greatest flexibility we decided to retain the old buffer cache code with certain modifications for optional use by file system implementations. The biggest driving force behind this is that we did not want to rely on the system page size being smaller than or equal to the size of control information boundaries for all file system implementations. Other reasons for maintaining parts of the old buffer cache code included some compatibility issues for customer written drivers and file systems. In current versions of SunOS, what's left of the old buffer cache is used strictly for UFS control buffers. We did improve the old buffer code so that buffers are allocated and freed dynamically. If no file system types choose to use the old buffer cache code (e.g., a diskless system), then no physical memory will be allocated to this pool. When the buffer cache is being used (e.g., for control information for UFS file systems), memory allocated to the buffer pool will be freed when demand for these system resources decreases.

9. System Changes

With the conversion to the new VM system, many closely related parts of the SunOS kernel required change as well. For the most part time constraints persuaded us to retain the old algorithms and policies.

9.1. Paging

The use of the global clock replacement algorithm implemented in 4.2BSD and extended in 4.3BSD is retained under the new VM system. The ‘‘clock hands’’ now sweep over *page* structures, calling *hat_pagesync*() on each eligible *page* to sync back the reference and modified bits from all the hat translations to that page. If a dirty page needs to be written back, the page daemon uses VOP_PUTPAGE() to write back the dirty page.

9.2. Swapping

We retained the basic notion of ‘‘swapping’’ a process. Under the new VM system there is much more sharing going on than was present in 4.2BSD where the only sharing was done explicitly via the *text* table. Now a process's address space may have several shared mappings, making it more difficult to understand the memory demands for an address space. This fact is made more obvious with the use of shared libraries [9, 10].

The address space provides an address space swap out operation *as_swapout*() which the SunOS kernel uses when swapping out a process. This procedure handles writing back dirty pages that the *as* maps and that no longer have any MMU translations after all the resources for the *as* being swapped are freed. The *as_swapout*() operation returns the number of bytes actually freed by the swap out operation. The swapper saves this value as a working set estimate[12], using it later to determine when enough memory has become available to swap the process back in. Also written back on a process swap out operation is the process's user area, which is set up to look like anonymous memory pages.

The *as* and segment structures used to describe the machine independent mappings of the address space for the process are currently not swapped out with the process since we don't yet have the needed support in the kernel dynamic memory allocator. This differs from the 4.2BSD VM implementation where the page tables used to describe the address space are written back as part of the swap out operation.

---

[12] Unfortunately, a poor one; this is an opportunity for future improvement.

9.3. System Calls

We rewrote many traditional UNIX system calls to manipulate the process's user address space. These calls include *fork, exec, brk,* and *ptrace.* For example, the *fork* system call uses an address space duplication operation. An *exec* system call destroys the old address space. For a demand paged executable it then creates a new address space using mappings to the executable file. For further discussion on how these system calls were implemented as address space operations see [1].

Memory management related system calls based on the original 4.2BSD specification [11] that were implemented include *mmap, munmap, mprotect, madvise,* and *mincore.* In addition, the *msync* system call was defined and implemented. For further discussion on these system calls see [1].

9.4. User Area

The UNIX user area is typically used to hold the process's supervisor state stack and other per-process information that is needed only when the process is in core. Currently the user area for a SunOS UNIX process is still at a fixed virtual address as is done with most traditional UNIX systems. However, the user area is specially managed so context switching can be done as quickly as possible using a fixed virtual address. There are several reasons why we want to convert to a scheme where the user areas are at different virtual addresses in the kernel's address space. Among them are faster context switching[13], better support for multi-threaded address spaces, and a more uniform treatment of kernel memory. In particular, we are moving toward a **seg_u** driver that can be used to manage a chunk of kernel virtual memory for use as u-areas.

10. Performance

A project goal for the new VM work was to provide more functionality without degrading performance. However, we have found that certain benchmarks show substantial performance improvements because of the much larger cache available for I/O operations. There is still much that can be done to the system as a whole by taking advantage of the new facilities.

Table 2 shows some benchmarks that highlight the effects of the new VM system and dynamically linked shared libraries [9, 10] over SunOS Release 3.2. *Dynamic* linking refers to delaying the final link edit process until run time. The traditional UNIX model is based on *static* linking in which executable programs are completely bound to all their libraries routines at program link time using *ld*(1).

Running a new VM kernel with same 3.2 binaries clearly shows that the new VM system and its associated file system changes has a positive performance impact. The effect of the larger system caching effects can be seen in the read times.

One way that the system uses the new VM architecture is a dynamically linked shared library facility. The *fork* and *exec* benchmarks show that the flexibility provided by this facility is not free. However, the benefits of the VM architecture that provides copy-on-write facilities more than compensate for the cost of duplicating mappings to shared libraries in the *fork* benchmark. The *exec* benchmark is the only test that showed performance degradation from dynamically linked executables over statically linked executables run with a SunOS Release 3.2 kernel. These numbers show that the startup cost associated with dynamically linking at run time is currently about 74 milliseconds. These results are preliminary and more work will be undertaken to streamline startup costs for dynamically linked executables. We feel that the added functionality provided by the dynamic binding facilities more than offsets the performance loss for programs dominated by start up costs.

11. Future Work

The largest remaining task is to incorporate better resource control policies so that the system can make more intelligent decisions based on increased system knowledge. We plan to investigate new management policies for page replacement and for better integration of memory and processor scheduling. We believe that the VM abstractions already devised will provide the hooks needed to do this. SunOS

[13] This is especially true with a virtual address cache and a fixed user area virtual address, since the old user area must be flushed before the mapping to the new user area at the same virtual address can be established.

Kernel Tested	SunOS 3.2	Pre-release New VM	Pre-release New VM
Binaries Executed	3.2	3.2	Dynamically Linked
Tests Performed	**Time (secs)**	**Time (secs)**	**Time (secs)**
exec 112k program 100 times	7.3	3.3	10.7
fork 112k program 200 times	8.8	4.4	7.7
Recursive stat of 125 directories	4.9	1.4	1.3
Page out 1 Mb to swap space	2.0	2.0	0.8
Page in 1 Mb from swap space	4.6	3.8	3.5
Demand page in 1 Mb executable	1.7	0.9	0.8
Sequentially read 1 Mb file (1st time)	1.6	1.5	1.5
Sequentially read 1 Mb file (2nd time)	1.6	0.4	0.4
Random read of 1 Mb file	5.7	0.7	0.8
Create and delete 100 tmp files	6.3	4.7	4.7

Table 2
System Benchmark Tests on a Sun-3/160
with 4 Megabytes of Memory and an Eagle Disk

kernel ports to different uniprocessor and multiprocessor machine bases will provide further understanding of the usability of the abstractions and our success in isolating machine dependencies.

Other future work involves taking advantage of the foundation established with the new VM architecture — both at the kernel and user level. Specialized segment drivers can be used at the kernel level to more elegantly support various unique hardware devices and to support new functionality such as mappings to other address spaces. Shared libraries are an example of the usefulness of mapped files at the user level. We expect to find the features of the new VM system used in various new facilities yet to be imagined. As new uses for the VM system are better understood, we can refine and complete the interfaces that have not yet been fully defined.

12. Conclusions

From our experience in implementing the new VM system, we draw the following conclusions.

- **Object oriented programming works.** The design of the new VM system was done using object-oriented techniques. This provided a coherent framework in which we could view the system.
- **The balance of responsibility is important.** When partitioning a problem amongst different abstractions, it is critical that the system be structured so that each abstraction has the right level of responsibility. When an abstraction gets control at the right time it has the opportunity to recognize and act on events that make sense for that abstraction.
- **The layering in the new VM system is effective.** For example, the *hat* layer provides all the machine dependent MMU translation control and has been found to be easily ported to new hardware architectures. The use of segment drivers has proven to make the system more extensible.
- **Performance did not suffer.** Although the new VM system provides considerably more functionality, it did so without any performance loss. Performance often improved because the new VM system better uses memory resources as a cache. By carefully designing the abstractions with optimizations for critical functions, we reduced the cost sometimes associated with object-oriented techniques to provide clean abstractions that are still efficient.

13. Acknowledgements

I would like to thank Rob Gingell, Dave Labuda, Bill Shannon, and especially Glenn Skinner, for reviewing this paper and helping to make it presentable and for their work with the new VM system. And most of all I would like to give a big thank you to my understanding wife Laurel, who continued to tolerate me during the countless extra hours I put into this project.

14. References

[1] Gingell, R. A., J. P. Moran, W. A. Shannon, ''Virtual Memory Architecture in SunOS'', *Summer Conference Proceedings, Phoenix 1987*, USENIX Association, 1987.

[2] Kleiman, S. R., ''Vnodes: An Architecture for Multiple File System Types in Sun UNIX'', *Summer Conference Proceedings, Atlanta 1986*, USENIX Association, 1986.

[3] Stroustrup, B., *The C++ Programming Language*, Addison-Wesley Publishing Company, 1986.

[4] Rashid, R., A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, J. Chew, ''Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures'', *Operating Systems Review*, Volume 21, No. 4, October, 1987.

[5] Kepecs, J. H., ''Lightweight Processes for UNIX Implementation and Applications'', *Summer Conference Proceedings, Portland 1985*, USENIX Association, 1985.

[6] Sun Microsystems Inc., *Sun-3 Architecture: A Sun Technical Report*, 1985.

[7] Cheng, R., ''Virtual Address Cache in UNIX'', *Summer Conference Proceedings, Phoenix 1987*, USENIX Association, 1987.

[8] McKusick, M. K., W. N. Joy, S. J. Leffler, R. S. Fabry, ''A Fast File System for UNIX'', *Transactions on Computer Systems*, Volume 2, No. 3, August, 1984.

[9] Gingell, R. A., M. Lee, X. T. Dang, M. S. Weeks, ''Shared Libraries in SunOS'', *Summer Conference Proceedings, Phoenix 1987*, USENIX Association, 1987.

[10] Gingell, R. A., ''Evolution of the SunOS Programming Environment'', *Spring Conference Proceedings, London 1988*, EUUG, 1988.

[11] Joy, W. N., R. S. Fabry, S. J. Leffler, M. K. McKusick, *4.2BSD System Manual*, Computer Systems Research Group, Computer Science Division, University of California, Berkeley, 1983.