Virtual Memory Architecture in SunOS

Robert A. Gingell
Joseph P. Moran
William A. Shannon

Sun Microsystems, Inc.
2550 Garcia Ave.
Mountain View, CA 94043

ABSTRACT

A new virtual memory architecture for the Sun implementation of the UNIX† operating system is described. Our goals included unifying and simplifying the concepts the system used to manage memory, as well as providing an implementation that fit well with the rest of the system. We discuss an architecture suitable for environments that (potentially) consist of systems of heterogeneous hardware and software architectures. The result is a page-based system in which the fundamental notion is that of mapping process addresses to files.

1. Introduction and Motivation

The UNIX operating system has traditionally provided little support for memory sharing between processes, and no support for facilities such as file mapping. For some communities, the lack of such facilities has been a barrier to the adoption of UNIX, or has hampered the development of applications that might have benefited from their availability. Our own desire to provide a shared libraries capability has provided additional incentive for us to explore providing new memory management facilities in the system.

We have also found ourselves faced with having to support a variety of interfaces. These included the partially implemented interfaces we have had in our 4.2BSD-derived kernel [JOY 83] and those specified by AT&T for System V [AT&T 86]. Aggravating these situations were the variations on those interfaces being developed by a number of vendors that were incompatible with or extended the original proposals. Also, entirely new interfaces have been proposed and implemented, most notably in Carnegie-Mellon's MACH [ACCE 86]. There has been no market movement to suggest which, if any, of these would become dominant, and in some cases a specific interface lacked an important capability (such as System V's lack of file mapping).

Finally, our existing implementation is too constraining a base from which to provide the new functionality we wanted. It is targeted to traditional models of UNIX memory management and specifically towards the hardware model of the VAX.‡ The work required to enhance the current implementation appeared to be adding its own new wart to an increasingly baroque implementation, and we were concerned for its long-term maintainability.

Thus, we decided to create a new Virtual Memory (VM) system for Sun's implementation of UNIX, SunOS. This paper describes the architecture of this new system: the goals we had for its design and the constraints under which we operated, the concepts it embodies, the interfaces it offers the UNIX application programmer and its relationship to the rest of the system. Although our primary intent is to discuss the architectural issues, information relating to the project and its implementation is provided to add context to the presentation.

---

† UNIX is a trademark of Bell Laboratories.
‡ VAX is a trademark of Digital Equipment Corporation

2. Goals/Non-Goals

Beyond the previously mentioned functional issues of memory sharing and file mapping, our goals for the new architecture were:

- **Unify memory handling**. Our primary architectural goal was to find the general concepts underlying all of the functions we wanted to provide or could envision, and then to provide them as the basis for all VM operations. If successful, we should be able to reimplement existing kernel functions (such as *fork* and *exec*) in terms of these new mechanisms. We also hoped to replace many of the existing memory management schemes in the kernel with facilities provided by the new VM system.
- **Non-kernel implementation of many functions**. If we were successful in identifying and providing the right mechanisms as kernel operations, then it seemed likely that many functions that otherwise would have had to be provided in the kernel could in fact be implemented as library routines. In particular, we wanted to be able to provide capabilities such as shared libraries and the System V interfaces as *applications* of these basic mechanisms.
- **Improved portability**. The existing system was targeted towards a specific machine architecture. In many cases, attributes of this architecture had crept cancerously through the code that implements software-defined functionality. We therefore wanted to describe software-defined objects using data structures appropriate to the software, and relegate machine-dependent code to a lower system layer accessed through a well-defined and narrow interface.
- **Consistent with environment**. We wanted our system to fit well with the UNIX concepts we were not changing. It would not be acceptable to build the world's most wonderful memory management system if it was completely incompatible with the rest of the system and its environment. Particularly important to us in this respect was the use of the file system as the name space for the objects supported by the system. Moreover, we sell systems that are intended to operate in highly networked environments, and thus we could not create a system that presented barriers to the networked environment.

In addition to these *architectural* goals, there were other goals we had for the project as a whole. These *project* goals were:

- **Maintain performance**. Although it is always desirable to tag a project with the label ''improves performance'', we chose the apparently more conservative goal of simply providing more functionality for the same cost in terms of overall system performance. While the new functionality might enable increased *application* performance, the performance of the system itself seemed uncertain. Further, when one considers that we replaced a mature implementation with one which has not been subjected to several years of tuning, getting back to current performance levels appeared to be an ambitious goal, something later experience has proven correct.
- **Engineer for the future**. We wanted to build an implementation that would be amenable to anticipated future requirements, such as kernel support for ''lightweight'' processes [KEPE 85] and multiprocessors.

When engaging in a large project, it is often as important to know what one's goals are *not*. In the architectural arena, our principal ''non-goals'' were:

- **New external interfaces**. As previously noted, a large number of groups were already working on the refinement and definition of interfaces. To the extent possible, we wanted to use such interfaces as had already been defined by others, and to provide those that were sufficiently defined to be implementable and that the market was demanding.
- **Compatible internal interfaces**. An unfortunate characteristic of UNIX is the existence of programs that have some understanding of the system's internals and use this information to rummage through the kernel by reading the memory device. The changes to the system we contemplated clearly made it impossible for us to try to support these programs, and thus we decided not to fool ourselves into trying.

Relevant project non-goals included:

- **Pageable kernel**. We did not intend to produce an implementation in which the kernel itself was paged – beyond a general desire in principle for the kernel to use less physical memory, we would have satisfied no specific functional goal by having the kernel pageable. However, it has turned out that a considerable portion of the memory that was previously ''wired down'' for kernel use is in fact now paged, although kernel code remains physically locked.
- **Massive policy changes**. Our interests lay in changing the mechanisms and what they provide, not in the policies by which they were administered. Although we would eventually like to support an integrated view of process and memory scheduling using techniques such as working set page replacement policies and balance set scheduling, we decided to defer these to future efforts.

3. Constraints

Working within the framework of an existing system imposed a number of constraints on what we could do. The constraints were not always limits on our flexibility; in fact, those reflecting specific customer requirements provided data that guided us through a number of design decisions. A major constraint was that of compatibility with previous versions of the system – ultimately, compatibility drove many decisions.

One such decision was that the new system would execute existing *a.out* files. This was necessary to preserve the utility of the programs already in use by customers and third parties. An important implication is that the system must provide a binary-compatible interface for existing programs, which means that existing system calls that perform memory management functions must continue to work. In our case, this meant supporting our partial implementation of the 4.2BSD *mmap*(2) system call, which we used to map devices, such as frame buffers, into a process's address space.

Although the system had to be binary-compatible, we did not feel constrained to leave it source-compatible, nor to use *mmap* as the principal interface to the memory management facilities of the system. Users with programs that used interfaces we changed in this manner would have to change their programs the next time they compiled them, but they would not be forced to recompile just to install and continue operating on the new system.

A wide variety of customer requirements implied that the interfaces we would offer would have to present very few constraints on a process's use of its address space. Some applications wanted to manage their address space completely, including the ability to assign process addresses for objects and to use a large, sparsely populated address space. Our own desire to build a base on which many different interfaces could be easily constructed suggested that we wanted as much flexibility as possible in user level address space management. However, other factors and requirements suggested that the system should also be able to control many details of an address space. One such factor was the introduction of a virtual address cache in the Sun-3/200 family of processors, where system control of address assignment would have a beneficial impact on performance. We also wanted to use copy-on-write techniques to enhance the level of sharing in the system, and to do this efficiently required page-level protection.

4. New Architecture: General Concepts

This section describes in general terms the abstractions and properties of the new VM system, and some reflections on the decisions that led to their creation. In many cases, our decisions were not based on obvious considerations, but rather ''fell out'' of a large number of small issues. Although this makes the decisions more difficult to explain, the process by which they were reached increased our confidence that, given our goals and constraints, we had in fact reached the best conclusion.

4.1. Pages vs. Segments

Our earliest decision was that the basic kernel facilities would operate on pages, rather than segments. The major factors in this decision included:

- compatibility with current systems (the 4.2BSD *mmap* is page-based);

- implementing efficient copy-on-write facilities required maintenance of per-page information anyway;
- pages appeared to offer the greatest opportunity to satisfy customer requirements for flexibility; and
- segments could be built as an abstraction on top of the page-based interface by library routines.

The major advantage to a segment-based mechanism appeared simply to be that it was a ‘‘better’’ programming abstraction. Since we could still build abstraction from the page-based mechanisms, and in fact gained some flexibility in building different forms of the abstraction as libraries, providing segments through the kernel appeared to offer little benefit and possibly even presented barriers to accomplishing some of our goals.

Although we believed we could gain the architectural advantages of segments through library routines built on our page-based system, another potential advantage to a segment-based system was the opportunity to implement a compact representation for a sparsely populated address space. However, since we needed per-page information to implement per-page copy-on-write and perform other physical storage management, at the very least we would end up with a mix of page- and segment-oriented data structures. We recognized that we could keep the major implementation advantage of a segment-based system, i.e., the concise description of the mapping for a range of addresses, by viewing it as an optimization (a sort of run-length encoding) of the per-page data structure (a similar scheme is used in MACH.)

4.2. Virtual Memory, Address Spaces, and Mapping

The system’s *virtual memory* consists of all its available physical memory resources. Examples include file systems (both local and remote), pools of unnamed memory (also known as *private* or *anonymous* storage, and implemented by the processor’s primary memory and *swap space*), and other random access memory devices. Named objects in the virtual memory are referenced through the UNIX file system. This does not imply that all file system objects are in the virtual memory, but simply that all named objects in the virtual memory are named in the file system. One of the strengths of UNIX has been the use of a single name-space for system objects, and we wished to build upon that strength. Some objects in the virtual memory, such as process private memory and our implementation of System V shared memory segments, do not have names. Although the most common form of object is the UNIX ‘‘regular file’’, previous work on SunOS has allowed for many different implementations of objects, which the system manipulates as an abstraction of the original UNIX *inode*, called a *vnode* [KLEI 86].

A process’s *address space* is defined by mappings onto the address spaces of one or more objects in the system’s virtual memory. As previously discussed, the system provides a page-based interface, and thus each mapping is constrained to be sized and aligned with the page boundaries defined by the system on which the process is executing. Each page may be mapped (or not) independently, and thus the programmer may treat an address space as a simple vector of pages. It should be noted that the only valid process address is one which is mapped to some object, and in particular there is no memory associated with the process itself – all memory is represented by virtual memory objects.

Each object in the virtual memory has an *object address space* defined by some physical storage, the specific form being object-specific. A reference to an object address accesses the physical storage that implements the address within the object. The virtual memory’s associated physical storage is thus accessed by transforming process addresses to object addresses, and then to the physical store. The system’s VM management facilities may interpose one or more layers of logical caching on top of the actual physical storage used to implement an object, a fact that has implications for *coherency*, discussed below.

A given process page may map to only one object, although a given object address may be the subject of many process mappings. The amount of the object’s address space covered by a mapping is an integral multiple of the page size as seen by the process performing the mapping. An important characteristic of a mapping is that the object to which the mapping is made is not required to be affected by the mere *existence* of the mapping. The implications of this are that it cannot, in general, be expected than an object has an ‘‘awareness’’ of having been mapped, or of which portions of its address space are accessed by mappings; in particular, the notion of a ‘‘page’’ is not a property of the object. Establishing a mapping to an object simply provides the *potential* for a process to access or change the object's contents.

The establishment of mappings provides an *access method* that renders an object directly addressable by a process. Applications may find it advantageous to access the storage resources they use directly rather than indirectly through *read* and *write*. Potential advantages include efficiency (elimination of unnecessary data copying) and reduced complexity (e.g., updates changed to a single step rather than a *read*, modify buffer, *write* cycle). The ability to access an object and have it retain its identity over the course of the access is unique to this access method, and facilitates the sharing of common code and data.

It is important to note that this *access method* view of the VM system does not directly provide sharing. Thus, although our motivations included providing shared memory, we have actually only provided the mechanisms for applications to *build* such sharing. For the system to provide not only an access method but also the *semantics* for such access is not only difficult or impossible, it is not clear that it is the correct thing to do in a highly heterogeneous environment. However, useful forms of sharing can be built in such environments, as the previous mechanisms for sharing in the kernel (such as the shared program text and file data buffer cache) have been subsumed by kernel programming building on top of these mechanisms.

4.3. Networking, Heterogeneity, and Coherence

Many of the factors that drove our adoption of the access method view of a VM system originated from our goal of providing facilities that ''fit'' with their expected environment. A major characteristic of our environment is the extensive use of networking to access file systems that would be part of the system's virtual memory. These networks are not constrained to consist of similar hardware or a common operating system; in fact, the opposite is encouraged. Making extensive assumptions about the properties of objects or their access creates potentially extensive barriers to accommodating heterogeneity. These properties include such system variables as page sizes and the ability of an object to synchronize its uses. While a given set of processes may *apply* a set of mechanisms to establish and maintain various properties of objects, a given operating system should not *impose* them on the rest of the network.

As it stands, the access method view of a virtual memory maintains the potential for a given object (say a text file) to be mapped by systems running our memory management system but also accessed by systems for which the notion of a virtual memory or storage management techniques such as paging would be totally foreign, such as PC-DOS. Such systems could continue to share access to the object, each using and providing its programs with the access method appropriate to that system. The alternative would be to prohibit access to the object by less capable systems, an alternative we find unacceptable.

A new consideration arises when applications use an object as a communications channel, or otherwise attempt to access it simultaneously. In addition to providing the mapping functions described previously, the VM management facilities also manage a storage hierarchy in which the processor's primary memory is often used as a cache for data from the virtual memory. Since the system cannot assume either that the object will coordinate accesses to it, nor that other systems will in fact cooperate with such coordination, it does not attempt on its own to synchronize the ''virtual memory cache'' it maintains. This is not to say that such objects can not exist, nor that systems will not cooperate; simply that *in general* the system can not make such an assumption. Even within a single system, the sharing that results is a consequence of the system's attempt to use its cache resources efficiently, not part of its defined functionality.

However, the lack of cache synchronization is not the limitation it might first appear. Applications that intend to share an object must employ a synchronization mechanism around their access and this requirement is independent of the access method they use. The scope and nature of the mechanism employed is best left to the application to decide. While today applications sharing a file object must access and update it indirectly using *read* and *write*, they must coordinate their access using semaphores or file locking or some application-specific protocol. In such environments, either caching is totally disabled (resulting in performance limitations) or the applications must employ a function such as *fsync* to ensure that the object is updated. Coherency of shared objects is not a new issue, and the introduction of a new access method simply exposes a new manifestation of an old problem. All that is required in an environment where mapping replaces *read* and *write* as the access method is that an operation comparable to *fsync* be provided.

Thus, the nature and scope of synchronization over shared objects is something that is application-defined from the outset. If the system attempted to impose any automatic semantics for sharing, it might prohibit other useful forms of mapped access that have nothing whatsoever to do with communication or sharing. By providing the mechanism to support coherency, and leaving it to cooperating applications to apply the mechanism, our design meets the needs of applications without providing barriers to heterogeneity. Note that this design does not prohibit the creation of libraries that provide coherent abstractions for common application needs. Not all abstractions on which an application builds need be supplied by the ‘‘operating system’’.

4.4. Historical Acknowledgements

Many of the concepts we have described are not new. MULTICS [ORGA 72] supported the notion of file/process memory integration that is fundamental to our system. TENEX [BOBR 72] [MURP 72] supported a page-based environment together with the notion of a process page map independent of the object being mapped.

5. External Interfaces: System Calls

The applications programmer gains access to the facilities of the new VM system through several sets of system calls. At present, we have defined our principal interface to be a refinement of those provided with 4.2BSD. We also provide interfaces for System V’s shared memory operations. The new system also impacted other system calls and facilities. These are described further below. Although these represent the initial interfaces we intend to support, others may be provided in the future in response to market demand.

5.1. 4.2BSD-based Interfaces

The 4.2BSD UNIX specification [JOY 83] included the definition of a number of system calls for mapping files, although the system did not implement them. Earlier releases of SunOS included partial implementations of these calls to support mapping devices such as frame buffers into a process’s address space. The basic concepts embedded in the interface were very close to our own, namely a page-based system providing mappings from process addresses to objects identified with file descriptors, and thus working from this base was a natural thing to do.

However, we had problems with the 4.2BSD interfaces due to their sketchy definition. Although the intent was well understood, the lack of an implementation left many semantic issues unresolved or ambiguous. We required some facilities that were not part of the specification, and other facilities were part of the specification but seemed superfluous. Thus, although we did manage to avoid creating an entirely new interface, we did find ourselves refining an existing, but unimplemented one. The process of refinement involved many people; in fact most were external to Sun and involved exchanges utilizing a ‘‘VM interest’’ mailing list supported and maintained by the developers at UC Berkeley, CSRG. Table 1 summarizes our refined interface, and the following sections expand on various areas of refinements.

5.1.1. *mmap*

The *mmap*(2) system call is used to establish mappings from a process’s address space to an object. Its definition is:

caddr_t mmap(addr, len, prot, flags, fd, off)

mmap establishes a mapping between the process’s address space at an address *paddr* for *len* bytes to the object specified by *fd* at offset *off* for *len* bytes. The value of *paddr* is an implementation-dependent function of the parameter *addr* and values of *flags*, further described below. A successful *mmap* call returns *paddr* as its result. The address ranges covered by [*paddr*, *paddr* + *len*) and [*off*, *off* + *len*) must be legitimate for the address space of a process and the object in question, respectively. The mapping established by *mmap* replaces any previous mappings for the process’s pages in the range [*paddr*, *paddr* + *len*).

The parameter *prot* determines whether *read*, *execute*, *write* or some combination of accesses are permitted to the pages being mapped. The values desired are expressed by or’ing the flags values PROT_READ, PROT_EXECUTE, and PROT_WRITE. It is not expected that all implementations

Table 1 – Refined 4.2BSD Interfaces

Call	Function
`madvise(addr, len, behav)` `  caddr_t addr; int len, behav;`	Gives advice about the handling of memory over a range of addresses.
`mincore(addr, len, vec)` `  caddr_t addr; int len; result char *vec;`	Determines residency of memory pages. (Will be replaced by more general map reading function.)

`caddr_t`
`mmap(addr, len, prot, flags, fd, off)`
`  caddr_t addr; int len, prot, flags, fd;`
`  off_t off;`
	Establish mapping from address space to object named by `fd`.

`mprotect(addr, len, prot)` `  caddr_t addr; int len, prot;`	Change protection on mapped pages.
`msync(addr, len, flags)` `  caddr_t addr; int len, flags;`	Synchronizes and/or invalidates cache of mapped data.
`munmap(addr, len)` `  caddr_t addr; int len;`	Removes mapping of address range.

literally provide all possible combinations. PROT_WRITE is often implemented as PROT_READ|PROT_WRITE, and PROT_EXECUTE as PROT_READ|PROT_EXECUTE. However, no implementation will permit a write to succeed where PROT_WRITE has not been set. The behavior of PROT_WRITE can be influenced by setting MAP_PRIVATE in the *flags* parameter.

The parameter *flags* provides other information about the handling of the pages being mapped. The options are defined by a field describing an enumeration of the ‘‘type’’ of the mapping, and a bit-field specifying other options. The enumeration currently defines two values, MAP_SHARED and MAP_PRIVATE. The bit-field values are MAP_FIXED and MAP_RENAME. The ‘‘type’’ value chosen determines whether stores to the mapped addresses are actually propagated to the object being mapped (MAP_SHARED) or directed to a copy of the object (MAP_PRIVATE). If the latter is specified, the initial write reference to a page will create a private copy of the page of the object and redirect the mapping to the copy. The mapping type is retained across a *fork*(2). The mapping ‘‘type’’ only affects the disposition of stores by *this* process – there is no insulation from changes made by other processes. If an application desires such insulation, it should use the *read* system call to make a copy of the data it wishes to keep protected.

MAP_FIXED informs the system that the value of *paddr* must be *addr*, exactly. The use of MAP_FIXED is discouraged, as it may prevent an implementation from making the most effective use of system resources.

When MAP_FIXED is not set, the system uses *addr* as a hint in an implementation-defined manner to arrive at *paddr*. The *paddr* so chosen will be an area of the address space that the system deems suitable for a mapping of *len* bytes to the specified object. All implementations interpret an *addr* value of zero as granting the system complete freedom in selecting *paddr*, subject to constraints described below. A non-zero value of *addr* is taken to be a suggestion of a process address near which the mapping should be placed. When the system selects a value for *paddr*, it will never place a mapping at address 0, nor will it replace any extant mapping, nor map into areas considered part of the potential data or stack ‘‘segments’’. In the current SunOS implementation, the system strives to choose alignments for mappings that maximize the performance of systems with a virtual address cache.

MAP_RENAME causes the pages currently mapped in the range [*paddr*, *paddr* + *len*) to be effectively renamed to be the object addresses in the range [*off*, *off* + *len*). The currently mapped pages must be mapped as MAP_PRIVATE. MAP_RENAME implies a MAP_FIXED interpretation of *addr*. *fd* must be open for write. MAP_RENAME affects the size of the memory object referenced by *fd*: the size is max(*off + len - 1*, *flen*) (where *flen* was the previous length of the object). After the pages are renamed, a mapping to them is reestablished with the parameters as specified in the renaming *mmap*.

The addition of MAP_FIXED and corresponding changes in the default interpretation of *addr* and *mmap*'s return value represent the principal change made to the original 4.2BSD specification. The change was made to remove the burden of managing a process's address space from applications that did not wish it.

5.1.2. Additions

We added one new system call, *msync*. *msync* has the interface

msync(addr, len, flags)

msync causes all modified copies of pages over the range [*addr, addr + len*) in system caches to be flushed to the objects mapped by those addresses. *msync* optionally invalidates such cache entries so that further references to the pages will cause the system to obtain them from their permanent storage locations. The *flags* argument provides a bit-field of values which influences *msync*'s behavior. The bit names and their interpretations are:

`MS_ASYNC`	Return immediately
`MS_INVALIDATE`	Invalidate caches

MS_ASYNC causes *msync* to return immediately once all I/O operations are scheduled; normally, *msync* will not return until all I/O operations are complete. MS_INVALIDATE causes all cached copies of data from mapped objects to be invalidated, requiring them to be re-obtained from the object's storage upon the next reference.

5.1.3. Unchanged Interfaces

Two 4.2BSD calls were implemented without change. They were *mprotect* for changing the protection values of mapped pages, and *munmap* for removing a mapping.

5.1.4. Removed: *mremap*

We deleted one system call, *mremap*. Upon reading the 4.2BSD specification, we had the impression that *mremap* was the mapping equivalent of the UNIX *mv* command. However, discussions with those involved in its original specification created confusion as to whether it was in fact supposed to be the equivalent of *mv*, *cp*, or *ln*. In the presence of the uncertainty and lacking any other motivation to include it, *mremap* was dropped from the system.

5.1.5. Open Issues

Two 4.2BSD system calls, *madvise* and *mincore*, remain unspecified. *madvise* is intended to provide information to the system to influence its management policies. Since a major rework of such policies was deferred to a future release, we decided to defer full specification and implementation of *madvise* until that time.

mincore was specified to return the residency status of a group of pages. Although the intent was clear, we felt that a more comprehensive interface for obtaining the status of a mapping was required. However, at present, this revised interface has not been defined.

Also unspecified is an interface for locking pages in memory. We envision either a new *mlock* system call, or a variation on *madvise*.

5.2. System V Shared Memory

The ''System V Interface Definition'' [AT&T 86] defines a number of operations on entities called ''shared memory segments''. Early in our project, we had hoped to implement these operations not as system calls but rather as library routines which built the System V abstractions out of the basic mechanisms supplied by the kernel. Unfortunately, System V shared memory is almost, but not completely the same as, a UNIX file. The primary differences are:

- **name space**: a shared memory segment exists in a name space different from that of the traditional UNIX file system; and
- **ownership and access**: a shared memory segment separates the notion of ''creator'' from ''owner''.

Together, these differences motivated a kernel-based implementation to allocate and manage the different name space (which shared implementation with other System V-specific objects such as semaphores), and to administer the different ownership and access control operations.

Although the databases peculiar to these differences are maintained inside the kernel, the implementation of the objects and access are built from the standard notions. Specifically, the memory object representing the shared memory segment exists as an unnamed object in the system's virtual memory, and the operation which attaches processes to it performs the internal equivalent of an *mmap*.

Implementation plans call for the object used to represent the shared memory segment to be supported by an anonymous memory-based file system. `/tmp` could be implemented as a file system of this type, potentially eliminating all I/O operations for temporary files and simply supporting them out of the processor's memory resources.

5.3. Other System Calls and Facilities

The new VM system has had an impact on other areas of the system as well, either extending or slightly altering the semantics of existing operations.

5.3.1. ''Segments''

Traditionally, the address space of a UNIX process has consisted of three segments: one each for write-protected program code (text), a heap of dynamically allocated storage (data), and the process's stack. Under the new system, a process's address space is simply a vector of pages and there exists no real structure to the address space. However, for compatibility purposes, the system maintains address ranges that ''should'' belong to such segments to support operations such as extending or contracting the data segment's ''break''. These are initialized when a program is initiated with *exec*.

5.3.2. *exec*

exec overlays a process's address space with a new program to be executed. Under the new system, *exec* performs this operation by performing the internal equivalent of an *mmap* to the file containing the program. The text and initialized data segments are mapped to the file, and the program's uninitialized data and stack areas are mapped to unnamed objects in the system's virtual memory. The boundaries of the mappings it establishes are recorded as representing the traditional ''segments'' of a UNIX process's address space.

exec establishes MAP_PRIVATE mappings, which has implications for the operation of *fork* and *ptrace*, as discussed below. The text segment is mapped with only PROT_READ and PROT_EXECUTE protections, so that write references to the text produce segmentation violations. The data segment is mapped as writable; however any page of initialized data that does not get written may be shared among all the processes running the program.

5.3.3. *fork*

Previously, a process created by *fork* had an address space made from a copy of its parent's address space. Under the new system, the address space is not copied, but the mappings defining it are. Since *exec* specifies MAP_PRIVATE on all the mappings it performs, parent and child thus effectively have copy-on-write access to a single set of objects. Further, since the mapping is generally far smaller than the data it describes, *fork* should be considerably more efficient. Any MAP_SHARED mappings in the parent are also MAP_SHARED in the child, providing the opportunity for both parent and child to operate on a common object.

5.3.4. *vfork*

Berkeley-based systems include a ‘‘VM-efficient’’ form of the *fork* system call to avoid the overhead of copying massive processes that simply threw away the copy operation with a subsequent *exec* call. At one point we hoped that the efficiencies gained through a reimplemented *fork* would obviate the need for *vfork*. Unfortunately, *vfork* is defined to suspend the parent process until the child performs either an *exec* or an *exit* and to allow the child full access to the parent’s address space (*not* a copy) in the interim. A number of programs take advantage of this quirk, allowing the child to record data in the address space for later examination by the parent. Eliminating *vfork* would break these programs, a fact we discovered in numerous ways when early versions of the system simply treated a *vfork* as *fork*. Further, *vfork* remains fundamentally more efficient than even a *fork* that only copies an address space map, since *vfork* copies nothing.

However, to encourage programmers at Sun to avoid the use of *vfork*, we took our time restoring it to the system and as a result got many programs ‘‘fixed’’.

5.3.5. *ptrace*

In previous versions of the system, the *ptrace* system call (used for process debugging) would refuse to deposit a breakpoint in a program that was being run by more than one process. This restriction was imposed by the nature of the old system’s facility for sharing program code, which was to share the entire text portion of an executable file.

In the new system, the system simply shares file pages among all those who have mappings to them. When a mapping is made MAP_PRIVATE, writes by a process to a page to which writes are permitted are diverted to a copy of the page − leaving the original object unaffected. *ptrace* takes advantage of the fact that an *exec* establishes the mapping to the file containing the program and its initialized data as MAP_PRIVATE, as it inserts a breakpoint by making a read-only page writable, depositing the breakpoint, and restoring the protection. The page on which the breakpoint is deposited, and only that page, is no longer shared with other users of the program − and their view of that page is unaffected.

5.3.6. *truncate*

The *truncate* system call has been changed so that it sets the length of a file. If the newly specified length is shorter than the file’s current length, *truncate* behaves as before. However, if the new length is longer, the file’s size is increased to the desired length. When writing a file exclusively through mapping, extending through *truncate* is the only alternative to MAP_RENAME operations for growing a file.

5.3.7. Resource Limits

Berkeley-based systems include functions for limiting the consumption of certain system resources. We have introduced a new resource limit: RLIMIT_PRIVATE. This limit controls the amount of ‘‘private memory’’ that a process may dynamically allocate from the system’s source of unnamed backing store. In many respects, RLIMIT_PRIVATE really describes the limit that RLIMIT_DATA and RLIMIT_STACK attempt to capture, namely the amount of swap space a given process may consume.

6. Internal Interfaces

The new VM system provides a set of abstractions and operations to the rest of the kernel. In many cases, these are used directly as the basis for the system call interfaces described above. In other areas they support internal forms of those system call interfaces, allowing the kernel to perform mappings for the address space in which it operates. The VM system also relies on services from other areas of the kernel.

6.1. Internal Role of VM

In general, the kernel uses the VM system as the manager of a logical cache of memory pages and as the object manager for ‘‘address space objects’’. In its role as cache manager, the VM system also manages the physical storage resources of the processor, as it uses these resources to implement the cache it maintains. The VM system is a particularly effective cache manager, and maintains a high degree of sharing over multiple uses of a given page of an object. As such, it has subsumed the functions of older data structures, in particular the text table and disk block data buffer cache (the ‘‘buffer cache’’). The VM system has replaced the old fixed-size buffer cache with a logical cache that uses all of the system’s pageable physical memory. Thus its use as a ‘‘buffer cache’’ in the old sense dynamically adapts to the pattern of the system’s use − in particular if the system is performing a high percentage of file references, all of the system’s pageable physical memory is devoted to a function that previously only had approximately 10% of the same resources. The VM system is also responsible for the management of the system’s memory management hardware, although these operations are invisible to the machine-independent portions of the kernel.

Kernel algorithms that operate on logical quantities of memory, such as the contents of file pages, do so by establishing mappings from the kernel’s address space to the object they wish to access. Those algorithms that implement the *read* and *write* system calls on such memory objects are particularly interesting: they operate by creating a mapping to the object and then copying the data to or from user buffers as appropriate. When mapping is used in this manner, users of the object are provided with a consistent view of the object, even if they mix references through mapped accesses or the *read* and *write* system calls. Note that the decision to use mapping operations in this way is left to the manager of the object being accessed.

The VM system does not know the semantics of the UNIX operating system. Instead, those properties of an address space that are the province of UNIX, such as the notions of ‘‘segments’’ and stack-growth, are implemented by a layer of UNIX semantics over the basic VM system. By providing only the basic abstractions from the VM system itself, we believe we have made it easier to provide future system interfaces that may not have UNIX-like characteristics.

The VM system relies on the rest of the system to provide managers for the objects to which it establishes mappings. These managers are expected to provide advice and assistance to the VM system to ensure efficient system management, and to perform physical I/O operations on the objects they manage. These responsibilities are detailed further below.

6.2. *as* layer

The primary object managed by the VM system is a (process) *address space* (*as*). The interfaces through which the system requests operations on an *as* object are summarized in Table 2, and are collectively referred to as the *as*-layer of the system. An *as* contains the memory of the mappings that comprise an address space. In addition, it contains a *hardware address translation* (*hat*) structure that holds the state of the memory management hardware associated with this address space. This structure is opaque to much of the VM system, and is interpreted only by a machine-dependent layer of the system, described further below.

An *as* exists independent of any of its uses, and may be shared by multiple processes, thus setting the stage for future integration of a multi-threaded address space capability as described in [KEPE 85]. The ‘‘address space’’ in which the kernel operates is also described by an *as* structure, and is the handle by which the kernel effects internal mapping operations using *as_map*.

The operations permitted on an *as* generally correspond to the functions provided by the system call interface. An implication of this is that just about any operation that the kernel could perform on an address space could also be implemented by an application directly. More work is necessary to define an interface for obtaining information about an *as*, to support the generation of *core* files, and the as-yet unspecified interfaces for reading mappings. An additional interface is also needed to support any advice operations we might choose to define in the future.

Internally to an address space, each individual mapping is treated as an object with a ‘‘mapping object manager’’. Such mappings are run-length compact encodings describing the mapping being performed, and may or may not have per-page information recorded depending on the nature of the mapping or subsequent references to the object being mapped. Due to a regrettable lack of imagination at a critical junction in our design, these ‘‘mapping objects’’ are termed *segments*, and their managers are called ‘‘segment drivers’’.

Table 2 – *as* operations	
Operation	**Function**
`struct as *as_alloc()`	*as* allocation.
`struct as *as_dup(as)` `  struct as *as;`	Duplicates `as` – used in *fork*.
`void as_free(as)` `  struct as *as;`	*as* deallocation.

`enum as_res`
`as_map(as, addr, size, crfp, crargsp)`
`  struct as *as; addr_t addr; u_int size;`
`  int (*crfp)(); caddr_t crargsp;`
	Internal *mmap*. Establish a mapping to an object using the mapping manager routine identified in `crfp`, providing object specific arguments in the opaque structure `crargsp`.

`enum as_res`
`as_unmap(as, addr, size)`
`  struct as *as; addr_t addr; u_int size;`
	Remove a mapping in `as`.

`enum as_res`
`as_setprot(as, addr, size, prot)`
`  struct as *as; addr_t addr;`
`  u_int size, prot;`
	Alter protection of mappings in `as`.

`enum as_res`
`as_checkprot(as, addr, size, prot)`
`  struct as *as; addr_t addr;`
`  u_int size, prot;`
	Determine whether mappings satisfy protection required by `prot`.

`enum as_res`
`as_fault(as, addr, size, type, rw)`
`  struct as *as; addr_t addr; u_int size;`
`  enum fault_type type; enum seg_rw rw;`
	Resolves a fault.

`enum as_res`
`as_faulta(as, addr, size)`
`  struct as *as; addr_t addr; u_int size;`
	Asynchronous fault – used for ‘‘fault-ahead’’.

6.3. *hat* layer

As previously noted, a *hat* is an object representing an allocation of memory management hardware resources. The set of operations on a *hat* are not visible outside of the VM system, but represent a machine-dependent/independent boundary called the *hat*-layer. Although it provides no services to the rest of the system, the *hat*-layer is of import to those faced with porting the system to various hardware architectures. It provides the mapping from the software data structures of an *as* and its internals to those required by the hardware of the system on which it resides.

We believe that the *hat*-layer has successfully isolated the hardware-specific requirements of Sun’s systems from the machine-independent portions of the VM system and the rest of the kernel. In particular, under the old system the addition of support for a virtual address cache permeated many areas of the system. Under the new system, support for the virtual address cache is isolated within the *hat* layer.

6.4. I/O Layer

The primary services the VM system requires of the rest of the kernel are physical I/O operations on the objects it maps. These operations occur across an interface called the ‘‘I/O Layer’’. Although used mainly to cause physical page frames to be filled (page-in) or drained (page-out) operations, the I/O layer also provides an opportunity for the managers of particular objects to map the system-specific page abstraction used by the VM system to the representation used by the object being mapped.

For instance, although the system operates on page-sized allocations, the 4.2BSD UNIX file system [MCKU 84] operates on collections of disk blocks that are often not page-sized. Efficient file system performance may also require non-page-sized I/O operations, in order to amortize the overhead of starting operations and to maximize the throughput of the particular subsystem involved. Thus, the VM system will pass several operations (such as the resolution of a fault on an object address, even one for which the VM system has a cached copy) through the object manager to provide it the opportunity to intercede. The object manager for NFS files uses these intercessions to prevent cached pages from becoming stale. Managers for network-coherent objects enforce coherence through this technique.

The I/O layer is to some extent bi-directional, as a given operation requested by the VM system may cause the object manager to request several VM-based operations. I/O clustering is an example of this, where a request by the VM system to obtain a page's worth of data may cause the object manager to actually schedule an I/O operation for logical pages surrounding the one requested in the hopes of avoiding future I/O requests. The old notion of ''read-ahead'' is implemented in this manner, and each object manager has the opportunity to recognize and act on patterns of access to a given object in a manner that maximizes its performance.

7. Project Status & Future Work

The architecture described in this paper has been implemented and ported to the Sun-2 and Sun-3 families of workstations. At present, all our major functional goals have been met. The work has consumed approximately four man-years of effort over a year and a half of real time. A surprisingly large amount of effort has been drained by efforts to interpose the VM system as the logical cache manager for the file systems, in particular with respect to the 4.2BSD UNIX file system.

With respect to our performance goals, more tuning work is required before we can claim to meet them. However, in some areas dealing with file access, early benchmarks reveal substantial performance improvements resulting from the much larger cache available for I/O operations. We expect further performance improvements when more of the system uses the new mechanisms. In particular, we expect an implementation of shared libraries to have a substantial impact upon the use of system resources. Future uses of mapping include a rewritten standard I/O library to use *mmap* rather than *read* and perhaps *write*, thus eliminating the dual copying of data and providing a transparent performance improvement to many applications. As sharing increases in the system, we expect the requirements for swap resources to decrease.

Other future work involves refining and completing the interfaces that have not yet been fully defined. We plan an investigation of new management policies, especially with respect to different page-replacement policies and the better integration of memory and processor scheduling. We would also like to port the system to different hardware bases, in particular to the VAX, to test the success of the *hat* layer in isolating machine dependencies from the rest of the system.

8. Conclusions

We believe the new VM architecture successfully meets our goals. Reviewing these reveals:

- **Unify memory handling**. All VM operations have been unified around the single notion of file mapping. Extant operations such as *fork* and *exec* have been reconstructed and their performance, and in some cases function, has been improved through their use of the new mechanisms.
- **Non-kernel implementation of many functions**. Although we were disappointed that kernel support was required to implement System V shared memory segments, we believe that this goal has been largely satisfied. In particular, our implementation of shared libraries [GING 87] requires no specific kernel support. We believe the basic operations the interfaces provide will permit the construction of other useful abstractions with user-level programming.
- **Improved portability**. Although more experience is required, we were pleased with the degree to which the Sun-3 virtual address cache was easily incorporated into the new system, in comparison with the difficulty experienced in integrating it into the previous system.

- **Consistent with environment**. The new system builds on the abstractions already in UNIX, in particular with respect to our use of the UNIX file system as the name space for named virtual memory objects. The integrated use of the new facilities in the system has helped to extend the previous abstractions in a natural manner. The semantics offered by the basic system mechanisms also do not impede the heterogeneous use of objects accessed through the system, an important consideration for the networked environments in which we expect the system to operate.

Finally, we have provided the functionality that motivated the work in the first place.

9. Acknowledgements

The system was designed by the authors, with Joe Moran providing the bulk of the implementation. Bill Joy offered commentary and advice on the architecture, as well as insights into the intents of the 4.2BSD interface, and an initial sketch of an implementation of the internal VM interfaces. Kirk McKusick and Mike Karels of UC Berkeley, CSRG, spent several days discussing the issues with us. The other members of Sun's System Software group gave considerable assistance and advice during the design and implementation of the system.

10. References

[ACCE 86] Accetta, M., R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, M. Young, ''Mach: A New Kernel Foundation for UNIX Development'', *Summer Conference Proceedings, Atlanta 1986*, USENIX Association, 1986.

[AT&T 86] AT&T, *System V Interface Definition*, Volume I, 1986

[BOBR 72] Bobrow, D. G., J. D. Burchfiel, D. L. Murphy, and R. S. Tomlinson, ''TENEX, a Paged Time Sharing System for the PDP-10'', *Communications of the ACM*, Volume 15, No. 3, March 1972.

[GING 87] Gingell, R. A., M. Lee, X. T. Dang, M. S. Weeks, ''Shared Libraries in SunOS'', *Summer Conference Proceedings, Phoenix 1987*, USENIX Association, 1987.

[JOY 83] 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.

[KEPE 85] Kepecs, J. H., ''Lightweight Processes for UNIX Implementation and Applications'', *Summer Conference Proceedings, Portland 1985*, USENIX Association, 1985.

[KLEI 86] Kleiman, S. R., ''Vnodes: An Architecture for Multiple File System Types in Sun UNIX'', *Summer Conference Proceedings, Atlanta 1986*, USENIX Association, 1986.

[MKCU 84] 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.

[MURP 72] Murphy, D. L., ''Storage organization and management in TENEX'', *Proceedings of the Fall Joint Computer Conference*, AFIPS, 1972.

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