Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures

Richard Rashid, Avadis Tevanian, Michael Young, David Golub,
Robert Baron, David Black, William Bolosky, and Jonathan Chew

Department of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

This paper describes the design and implementation of virtual memory management within the CMU Mach Operating System and the experiences gained by the Mach kernel group in porting that system to a variety of architectures. As of this writing, Mach runs on more than half a dozen uniprocessors and multiprocessors including the VAX family of uniprocessors and multiprocessors, the IBM RT PC, the SUN 3, the Encore MultiMax, the Sequent Balance 21000 and several experimental computers. Although these systems vary considerably in the kind of hardware support for memory management they provide, the machine-dependent portion of Mach virtual memory consists of a single code module and its related header file. This separation of software memory management from hardware support has been accomplished without sacrificing system performance. In addition to improving portability, it makes possible a relatively unbiased examination of the pros and cons of various hardware memory management schemes, especially as they apply to the support of multiprocessors.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4864, monitored by the Space and Naval Warfare Systems Command under contract N00039-85-C-1034.

This paper was presented at the 2nd Symposium on Architectural Support for Programming Languages and Operating Systems, ACM October, 1987.

1. Introduction

While software designers are increasingly able to cope with variations in instruction set architectures, operating system portability continues to suffer from a proliferation of memory structures. UNIX systems have traditionally addressed the problem of VM portability by restricting the facilities provided and basing implementations for new memory management architectures on versions already done for previous systems. As a result, existing versions of UNIX, such as Berkeley 4.3bsd, offer little in the way of virtual memory management other than simple paging support. Versions of Berkeley UNIX on non-VAX hardware, such as SunOS on the SUN 3 and ACIS 4.2 on the IBM RT PC, actually simulate internally the VAX memory mapping architecture -- in effect treating it as a machine-independent memory management specification.

Over the last two years CMU has been engaged in the development of a portable, multiprocessor operating system called Mach. One of the goals of Mach has been to explore the relationship between hardware and software memory architectures and to design a memory management system that would be readily portable to multiprocessor computing engines as well as traditional uniprocessors.

Mach provides complete UNIX 4.3bsd compatibility while significantly extending UNIX notions of virtual memory management and interprocess communication [1]. Mach supports:

- *large, sparse virtual address spaces,*
- *copy-on-write virtual copy operations,*
- *copy-on-write and read-write memory sharing between tasks,*
- *memory mapped files and*
- *user-provided backing store objects and pagers.*

This has been accomplished without patterning Mach's internal memory representation after any specific architecture. In fact, Mach makes relatively few assumptions about available memory management hardware. The primary requirement is an ability to handle and recover from page faults (for some arbitrary page size).

As of this writing, Mach runs on more than half a dozen uniprocessors and multiprocessors including the entire VAX family of uniprocessors and multiprocessors, the IBM RT PC, the SUN 3, the Encore MultiMax and the Sequent Balance 21000. Implementations are in progress for several experimental computers. Despite differences between supported architectures, the machine-dependent portion of Mach's virtual memory subsystem consists of a single code module and its related header file. All information important to the management of Mach virtual memory is maintained in machine-independent data structures and machine-dependent data structures contain only those mappings necessary to running the current mix of programs.

Mach's separation of software memory management from hardware support has been accomplished without sacrificing system performance. In several cases overall system performance has measurably improved over existing UNIX implementations. Moreover, this approach makes possible a relatively unbiased examination of the pros and cons of various hardware memory management schemes, especially as they apply to the support of multiprocessors. This paper describes the design and implementation of virtual memory management within the CMU Mach Operating System and the experiences gained by the Mach kernel group in porting that system to a variety of architectures.

2. Mach Design

There are five basic Mach abstractions:

1. A *task* is an execution environment in which threads may run. It is the basic unit of resource allocation. A task includes a paged virtual address space and protected access to system resources (such as processors, port capabilities and virtual memory). A task address space consists of an ordered collection of mappings to memory objects (see below). The UNIX notion of a *process* is, in Mach, represented by a task with a single thread of control.

2. A *thread* is the basic unit of CPU utilization. It is roughly equivalent to an independent program counter operating within a task. All threads within a task share access to all task resources.

3. A *port* is a communication channel -- logically a queue for messages protected by the kernel. Ports are the reference objects of the Mach design. They are used in much the same way that object references could be used in an object oriented system. *Send* and *Receive* are the fundamental primitive operations on ports.

4. A *message* is a typed collection of data objects used in communication between threads. Messages may be of any size and may contain pointers and typed capabilities for ports.

5. A *memory object* is collection of data provided and managed by a server which can be mapped into the address space of a task.

Operations on objects other than messages are performed by sending messages to ports. In this way, Mach permits system services and resources to be managed by user-state tasks. For example, the Mach kernel itself can be considered a task with multiple threads of control. The kernel task acts as a server which in turn implements tasks, threads and memory objects. The act of creating a task, a thread or a memory object, returns access rights to a port which represents the new object and can be used to manipulate it. Incoming messages on such a port results in an operation performed on the object it represents.

The indirection provided by message passing allows objects to be arbitrarily placed in the network (either within a multiprocessor or a workstation) without regard to programming details. For example, a thread can suspend another thread by sending a suspend message to that thread’s *thread port* even if the requesting thread is on another node in a network. It is thus possible to run varying system configurations on different classes of machines while providing a consistent interface to all resources. The actual system running on any particular machine is thus more a function of its servers than its kernel.

Traditionally, message based systems of this sort have operated at a distinct performance disadvantage to conventionally implemented operating systems. The key to efficiency in Mach is the notion that virtual memory management can be integrated with a message-oriented communication facility. This integration allows large amounts of data including whole files and even whole address spaces to be sent in a single message with the efficiency of simple memory remapping.

2.1. Basic VM Operations

Each Mach task possesses a large address space that consists of a series of mappings between ranges of memory addressible to the task and memory objects. The size of a Mach address space is limited only by the addressing restrictions of the underlying hardware. An RT PC task, for example, can address a full 4 gigabytes of memory under Mach[1] while the VAX architecture allows at most 2 gigabytes of user address space. A task can modify its address space in several ways, including:

- *allocate a region of virtual memory on a page boundary,*

---

[1]This feature is actively used at CMU by the CMU RT implementation of CommonLisp.

- *deallocate a region of virtual memory,*
- *set the protection status of a region of virtual memory,*
- *specify the inheritance of a region of virtual memory and*
- *create and manage a memory object that can then be mapped into the address space of another task.*

The only restriction imposed by Mach on the nature of the regions that may be specified for virtual memory operations is that they must be aligned on system page boundaries. The definition of page size is a boot time system parameter and can be any power of two multiple of the hardware page size. Table 2-1 lists the set of virtual memory operations that can be performed on a task.

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

Table 2-1:

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

Both copy-on-write and read/write sharing of memory are permitted between Mach tasks. Copy-on-write sharing between unrelated tasks is typically the result of large message transfers. An entire address space may be sent in a single message with no actual data copy operations performed. Read/write shared memory can be created by allocating a memory region and setting its inheritance attribute. Subsequently created child tasks share the memory of their parent according to its inheritance value. Inheritance may be specified as *shared*, *copy* or *none*, and may be specified on a per-page basis. Pages specified as *shared*, are shared for read and write. Pages marked as *copy* are logically copied by value, although for efficiency copy-on-write techniques are employed. An inheritance specification of *none* signifies that a page is not to be passed to a child. In this case, the child's corresponding address is left unallocated.

Like inheritance, protection is specified on a per-page basis. For each group of pages there exist two protection values: the current and the maximum protection. The current protection controls actual hardware permissions. The maximum protection specifies the maximum value that the current protection may take. While the maximum protection can never be raised, it may be lowered. If the maximum protection is lowered to a level below the current protection, the current protection is also lowered to that level. Each protection is implemented as a combination of read, write and execute permissions. Enforcement of access permissions depends on hardware support. For example, many machines do not allow for explicit execute permissions, but those that do will have that protection properly enforced.

Mach’s implementation of UNIX **fork** is an example of how its virtual memory operations can be used. When a fork operation is invoked, the newly created child task address map is created based on the parent’s inheritance values. By default, all inheritance values for an address space are set to copy. Thus the child’s address space is, by default, a copy-on-write copy of the parent’s and UNIX address space copy semantics are preserved.

One of the more unusual features of Mach is that fact that virtual memory related functions, such as pagein and pageout, can be performed directly by user-state tasks for memory objects they create. Section 3.3 describes this aspect of the system.

3. The Implementation of Mach Virtual Memory

Four basic memory management data structures are used in Mach:

1. the *resident page table* --

   a table used to keep track of information about machine independent pages,

2. the *address map* --

   a doubly linked list of map entries, each of which describes a mapping from a range of addresses to a region of a memory object,

3. the *memory object* --

   a unit of backing storage managed by the kernel or a user task and

4. the *pmap* --

   a machine dependent memory mapping data structure (i.e., a hardware defined physical address map).

The implementation is split between machine *independent* and machine *dependent* sections. Machine dependent code implements only those operations necessary to create, update and manage the hardware required mapping data structures. All important virtual memory information is maintained by machine independent code. In general, the machine dependent part of Mach maintains only those mappings which are crucial to system execution (e.g., the kernel map and the mappings for frequently referenced task addresses) and may garbage collect non-important mapping information to save space or time. It has no knowledge of machine independent data structures and is not required to maintain full knowledge of valid mappings from virtual addresses to hardware pages.

3.1. Managing Resident Memory

Physical memory in Mach is treated primarily as a cache for the contents of virtual memory objects. Information about physical pages (e.g., modified and reference bits) is maintained in page entries in a table indexed by physical page number. Each page entry may simultaneously be linked into several lists:

- a *memory object list*,
- a *memory allocation queue* and
- a *object/offset hash bucket*.

All the page entries associated with a given object are linked together in a *memory object list* to speed-up object deallocation and virtual copy operations. Memory object semantics permit each page to belong to at most one memory object. *Allocation queues* are maintained for free, reclaimable and allocated pages and are used by the Mach paging daemon. Fast lookup of a physical page associated with an object/offset at the time of a page fault is performed using a bucket hash table keyed by memory object and byte offset.

Byte offsets in memory objects are used throughout the system to avoid linking the implementation to a particular notion of physical page size. A Mach physical page does not, in fact, correspond to a page as defined by the memory mapping hardware of a particular computer. The size of a Mach page is a boot time system parameter. It relates to the physical page size only in that it must be a power of two multiple of the machine dependent size. For example, Mach page sizes for a VAX can be 512 bytes, 1K bytes, 2K bytes, 4K bytes, etc. Mach page sizes for a SUN 3, however, are limited to 8K bytes, 16K bytes, etc. The physical page size used in Mach is also independent of the page size used by memory object handlers (see section below).

3.2. Address Maps

Just as the kernel keeps track of its own physical address space, it must also manage its virtual address space and that of each task. Addresses within a task address space are mapped to byte offsets in memory objects by a data structure called an *address map*.

An address map is a doubly linked list of *address map entries* each of which maps a contiguous range of virtual addresses onto a contiguous area of a memory object. This linked list is sorted in order of ascending virtual address and different entries may not map overlapping regions of memory.

Each address map entry carries with it information about the inheritance and protection attributes of the region of memory it defines. For that reason, all addresses within a range mapped by an entry must have the same attributes. This can force the system to allocate two address map entries that map adjacent memory regions to the same memory object simply because the properties of the two regions are different.

This address map data structure was chosen over many alternatives because it was the simplest that could efficiently implement the most frequent operations performed on a task address space, namely:

- *page fault lookups,*
- *copy/protection operations on address ranges and*
- *allocation/deallocation of address ranges.*

A sorted linked list allows operations on ranges of addresses (e.g., copy-on-write copy operations) to be done simply and quickly and does not penalize large, sparse address spaces. Moreover, fast lookup on faults can be achieved by keeping last fault "hints". These hints allow the address map list to be searched from the last entry found for a fault of a particular type. Because each entry may map a large region of virtual addresses, an address map is typically small. A typical VAX UNIX process has five mapping entries upon creation - one for its UNIX u-area and one each for code, stack, initialized and uninitialized data.

3.3. Memory Objects

A Mach address map need not keep track of backing storage because all backing store is implemented by Mach *memory objects*. Logically, a virtual memory object is a repository for data, indexed by byte, upon which various operations (e.g., read and write) can be performed. In many respects it resembles a UNIX file.

A reference counter is maintained for each memory object. This counter allows the object to be garbage collected when all mapped references to it are removed. In some cases, for example UNIX text segments or other frequently used files, it is desirable for the kernel to retain information about an object even after the last mapping reference disappears. By retaining the physical page mappings for such objects subsequent reuse can be made very inexpensive. Mach maintains an cache of such frequently used memory objects. A pager may use domain specific knowledge to request that an object be kept in this cache after it is no longer referenced.

An important feature of Mach's virtual memory is the ability to handle page faults and page-out requests outside of the kernel. This is accomplished by associating with each memory object a managing task (called a *pager*). For example, to implement a memory mapped file, virtual memory is created with its pager specified as the file system. When a page fault occurs, the kernel will translate the fault into a request for data from the file system.

Access to a pager is represented by a port (called the *paging_object* port) to which the kernel can send messages requesting data or notifying the pager about a change in the object's primary memory cache. In addition to this pager port, the kernel maintains for each memory object a unique identifier called the *paging_name* which is also represented by a port. The kernel also maintains some status information and a list of physical pages currently cached in primary memory. Pages currently in primary memory are managed by the kernel through the operation of the kernel paging daemon. Pages not in primary memory are stored and fetched by the pager. A third port, the *paging_object_request* port is used by the pager to send messages to the kernel to manage the object or its physical page cache.

Tables 3-1 and 3-2 list the calls (messages) made by the kernel on an external pager and by an external pager on the kernel. Using this interface an external pager task can manage virtually all aspects of a memory object including physical memory caching and permanent or temporary secondary storage. Simple pagers can be implemented by largely ignoring the more sophisticated interface calls and implementing a trivial read/write object mechanism.

A pager may be either internal to the Mach kernel or an external user-state task. Mach currently provides some basic paging services inside the kernel. Memory with no pager is automatically zero filled, and page-out is done to a default inode pager. The current inode pager utilizes 4.3bsd UNIX file systems and eliminates the traditional Berkeley UNIX need for separate paging partitions.

Kernel to External Pager Interface

pager_server(message)	
	Routine called by task to process a message from the kernel.
pager_init(paging_object, pager_request_port,pager_name)	
	Initialize a paging object (i.e. memory object).
pager_create(old_paging_object, new_paging_object, new_request_port,new_name)	
	Accept ownership of a memory object.
pager_data_request(paging_object,pager_request_port,offset,length,desired_access)	
	Requests data from an external pager.
pager_data_unlock(paging_object,pager_request_port,offset,length,desired_access)	
	Requests an unlock of an object.
pager_data_write(paging_object, offset, data, data_count)	
	Writes data back to a memory object.

Table 3-1:

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

3.4. Sharing Memory: Sharing Maps and Shadow Objects

When a copy-on-write copy is performed, the two address maps which contain copies point to the same memory object. Should both tasks only read the data, no other mapping is necessary.

If one of the two tasks writes data "copied" in this way, a new page accessible only to the writing task must be allocated into which the modifications are placed. Such copy-on-write memory management requires that the kernel maintain information about which pages of a memory object have been modified and which have not. Mach manages this information by creating memory objects specifically for the purpose of holding modified pages which originally belonged to another object. Memory objects created for this purpose are referred to as *shadow objects*.

External Pager to Kernel Interface

vm_allocate_with_pager(target_task, address, size, anywhere, paging_object, offset)
Allocate a region of memory at specified address backed by a memory object.

pager_data_provided(paging_object_request, offset, data, data_count, lock_value)
Supplies the kernel with the data contents of a region of a a memory object.

pager_data_unavailable(paging_object_request, offset, size)
Notifies kernel that no data is available for that region of a memory object.

pager_data_lock(paging_object_request, offset, length, lock_value)
Prevents further access to the specified data until an unlock or it specifies an unlock event.

pager_clean_request(paging_object_request, offset, length)
Forces modified physically cached data to be written back to a memory object.

pager_flush_request(paging_object_request, offset, length)
Forces physically cached data to be destroyed.

pager_readonly(paging_object_request)
Forces the kernel to allocate a new memory object should a write attempt to this paging object be made.

pager_cache(paging_object_request, should_cache_object)
Notifies the kernel that it should retain knowledge about the memory object even after all references to it have been removed.

Table 3-2:

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

A shadow object collects and "remembers" modified pages which result from copy-on-write faults. A shadow object is created as the result of a copy-on-write fault taken by a task. It is initially an empty object without a pager but with a pointer to the shadowed object. A shadow object need not (and typically does not) contain all the pages within the region it defines. Instead, it relies on the original object that it shadows for all unmodified data. A shadow object may itself be shadowed as the result of a subsequent copy-on-write copy. When the system tries to find a page in a shadow object, and fails to find it, it proceeds to follow this list of objects. Eventually, the system will find the page in some object in the list and make a copy, if necessary.

While memory objects can be used in this way to implementing copy-on-write, the memory object data structure is not appropriate for managing read/write sharing. Operations on shared regions of memory may involve mapping or remapping many existing memory objects. In addition, several tasks may share a region of memory read/write and yet simultaneously share the same data copy-on-write with another task. This implies the need to provide a level of indirection when accessing a shared object. Because operations of shared memory regions are logically address map operations, read/write memory sharing requires a map-like data structure which can be referenced by other address maps. To solve these problems, address map entries are allowed to point to a *sharing map* as well as a memory object. The sharing map, which is identical to an address map, then points to shared memory objects. Map operations that should apply to all maps sharing the data are simply applied to the sharing map. Because sharing maps can be split and merged, sharing maps do not need to reference other sharing maps for the full range of task-to-task address space sharing to be permitted. This simplifies map operations and obviates the need for sharing map garbage collection.

3.5. Managing the Object Tree

Most of the complexity of Mach memory management arises from a need to prevent the potentially large chains of shadow objects which can arise from repeated copy-on-write remapping of a memory object from one address space to another. Remapping causes shadow chains to be created when mapped data is repeatedly modified -- causing a shadow object to be created -- and then recopied. A trivial example of this kind of shadow chaining can be caused by a simple UNIX process which repeatedly forks its address space causing shadow objects to be built in a long chain which ultimately points to the memory object which backs the UNIX stack.

As in the fork example, most cases of excessive object shadow chaining can be prevented by recognizing that new shadows often completely overlap the objects they are shadowing. Mach automatically garbage collects shadow objects when it recognizes that an intermediate shadow is no longer needed. While this code is, in principle, straightforward, it is made complex by the fact that unnecessary chains sometimes occur during periods of heavy paging and cannot always be detected on the basis of in memory data structures alone. Moreover, the need to allow the paging daemon to access the memory object structures, perform garbage collection and still allow virtual memory operations to operate in parallel on multiple CPUs has resulted in complex object locking rules.

3.6. The Machine-Independent/Machine-Dependent Interface

The purpose of Mach's machine dependent code is the management of physical address maps (called *pmaps*). For a VAX, a pmap corresponds to a VAX page table. For the IBM RT PC, a pmap is a set of allocated segment registers. The machine dependent part of Mach is also responsible for implementing page level operations on pmaps and for ensuring that the appropriate hardware map is operational whenever the state of the machine needs to change from kernel to user state or user to kernel state. All machine dependent mapping is performed in a single module of the system called *pmap.c*.

One of the more unusual characteristics of the Mach dependent/independent interface is that the pmap module need not keep track of all currently valid mappings. Virtual-to-physical mappings may be thrown away at almost any time to improve either space or speed efficiency and new mappings need not always be made immediately but can often be lazy-evaluated. In order to cope with hardware architectures which make virtual-to-physical map invalidates expensive, pmap may delay operations which invalidate or reduce protection on ranges of addresses until such time as they are actually necessary.

All of this can be accomplished because all virtual memory information can be reconstructed at fault time from Mach's machine independent data structures. The only major exceptions to the rule that pmap maintains only a cache of available mappings are the kernel mappings themselves. These must always be kept complete and accurate. Full information as to to which processors are currently using which maps and when physical maps must be made correct is provided to pmap from machine-independent code.

In all cases, machine-independent memory management is the driving force behind all Mach VM operations. The interface between machine-independent and machine-dependent modules has been kept relatively small and the implementor of pmap needs to know very little about the way Mach functions. Tables 3-3 and 3-4 list the pmap routines which currently make up the Mach independent/dependent interface.

4. Porting Mach VM

The Mach virtual memory code described here was originally implemented on VAX architecture machines including the MicroVAX II, 11/780 and a four processor VAX system called the VAX 11/784. The first relatively stable VAX version was available within CMU in February, 1986. At the end of that same month the first port of Mach -- to the IBM RT PC -- was initiated by a newly hired programmer who had not previously either worked on

Exported and Required PMAP Routines	
pmap_init(start, end)	*initialize using the specified range of physical addresses.*
pmap_t pmap_create()	*create a new physical map.*
pmap_reference(pmap)	*add a reference to a physical map.*
pmap_destroy(pmap)	*deference physical map, destroy if no references remain*
pmap_remove(pmap, start, end)	*remove the specified range of virtual address from map. [Used in memory deallocation]*
pmap_remove_all(phys)	*remove physical page from all maps. [pageout]*
pmap_copy_on_write(phys)	*remove write access for page from all maps. [virtual copy of shared pages]*
pmap_enter(pmap, v, p, prot, wired)	*enter mapping. [page fault]*
pmap_protect(map, start, end, prot)	*set the protection on the specified range of addresses.*
vm_offset_t pmap_extract(pmap, va)	*convert virtual to physical.*
boolean_t pmap_access(pmap, va)	*report if virtual address is mapped.*
pmap_update()	*sync pmap system.*
pmap_activate(pmap, thread, cpu)	*setup map/thread to run on cpu.*
pmap_deactivate(pmap, th, cpu)	*map/thread are done on cpu.*
pmap_zero_page(phys)	*zero fill physical page.*
pmap_copy_page(src, dest)	*copy physical page. [modify/reference bit maintenance]*

Table 3-3:

These routines must be implemented, although they may not necessarily perform any operation on a pmap data structure if not required by the hardware for a given machine.

Exported but Optional PMAP Routines	
pmap_copy(dest_pmap, src_pmap, dst_addr, len, src_addr)	*copy specified virtual mapping.*
pmap_pageable(pmap, start, end, pageable)	*specify pageability of region.*

Table 3-4:

These routines need not perform any hardware function.

an operating system or programmed in C. By early May the RT PC version was self hosting and available to a small group of users. There are currently approximately 75 RT PC's running Mach within the CMU Department of Computer Science.

The majority of time required for the RT PC port was spent debugging compilers and device drivers. The estimate of time spent in implementing the pmap module is approximately 3 weeks -- much of that time spent understanding the code and its requirements. By far the most difficult part of the pmap module to "get right" was the precise points in the code where validation/invalidation of hardware address translation buffers were required.

Implementations of Mach on the SUN 3, Sequent Balance and Encore MultiMAX have each contributed similar experiences. The Sequent port was the only one done by an expert systems programmer. The result was a bootable system only five weeks after the start of programming. In each case Mach has been ported to systems which possessed either a 4.2bsd or System V UNIX. This has aided the porting effort significantly by reducing the effort required to build device drivers.

5. Assessing Various Memory Management Architectures

Mach's virtual memory system is portable, makes few assumptions about the underlying hardware base and has been implemented on a variety of architectures. This has made possible a relatively unbiased examination of the pros and cons of various hardware memory management schemes.

In principle, Mach needs no in-memory hardware-defined data structure to manage virtual memory. Machines which provide only an easily manipulated TLB could be accommodated by Mach and would need little code to be written for the pmap module[2]. In practice, though, the primary purpose of the pmap module is to manipulate hardware defined in-memory structures which in turn control the state of an internal MMU TLB. To date, each hardware architecture has had demonstrated shortcomings, both for uniprocessor use and even more so when bundled in a multiprocessor.

5.1. Uniprocessor Issues

Mach was initially implemented on the VAX architecture. Although, in theory, a full two gigabyte address space can be allocated in user state to a VAX process, it is not always practical to do so because of the large amount of linear page table space required (8 megabytes). UNIX systems have traditionally kept page tables in physical memory and simply limited the total process addressiblity to a manageable 8, 16 or 64 megabytes. VAX VMS handles the problem by making page tables pageable within the kernel's virtual address space. The solution chosen for Mach was to keep page tables in physical memory, but only to construct those parts of the table which were needed to actually map virtual to real addresses for pages currently in use. VAX page tables in Mach may be created and destroyed as necessary to conserve space or improve runtime. The necessity to manage page tables in this fashion and the large size of a VAX page table (partially the result of the small VAX page size of 512 bytes) has made the machine dependent portion of that system more complex than that for other architectures.

The IBM RT PC does not use per-task page tables. Instead it uses a single inverted page table which describes which virtual address is mapped to each physical address. To perform virtual address translation, a hashing function is used to query the inverted page table. This allows a full 4 gigabyte address space to be used with no additional overhead due to address space size. Mach has benefited from the RT PC inverted page table in significantly reduced memory requirements for large programs (due to reduced map size) and simplified page table management.

One drawback of the RT, however, is that it allows only one valid mapping for each physical page, making it impossible to share pages without triggering faults. The rationale for this restriction lies in the fact that the designers of the RT targeted an operating system which did not allow virtual address aliasing. The result, in Mach, is that physical pages shared by multiple tasks can cause extra page faults, with each page being mapped and then remapped for the last task which referenced it. The effect is that Mach treats the inverted page table as a kind of large, in memory cache for the RT's translation lookaside buffer (TLB). The surprising result has been that, to date, these extra faults are rare enough in normal application programs that Mach is able to outperform a version of UNIX (IBM ACIS 4.2a) on the RT which avoids such aliasing altogether by using shared segments instead of shared pages.

In the case of the SUN 3 a combination of segments and page tables are used to create and manage per-task address maps up to 256 megabytes each. The use of segments and page tables make it possible to reasonably implement sparse addressing, but only 8 such *contexts* may exist at any one time. If there are more than 8 active tasks, they compete for contexts, introducing additional page faults as on the RT.

---

[2] In fact, a version of Mach has already run on a simulator for the IBM RP3 which assumed only TLB hardware support.

The main problem introduced by the SUN 3 was the fact that the physical address space of that machine has potentially large "holes" in it due to the presence of display memory addressible as "high" physical memory. This can complicate the management of the resident page table which becomes a "sparse" data structure. In the SUN version of Mach it was possible to deal with this problem completely within machine dependent code.

Both the Encore Multimax and the Sequent Balance 21000 use the National 32082 MMU. This MMU has posed several problems unrelated to multiprocessing:

- Only 16 megabytes of virtual memory may be addressed per page table. This requirement is very restrictive in large systems, especially for the kernel's address space.
- Only 32 megabytes of physical memory may be addressed[3]. Again, this requirement is very restrictive in large systems.
- A chip bug apparently causes read-modify-write faults to always be reported as read faults. Mach depends on the ability to detect write faults for proper copy-on-write fault handling.

It is unsurprising that these problems have been addressed in the successor to the NS32082, the NS32382.

5.2. Multiprocessor Issues

When building a shared memory multiprocessor, care is usually taken to guarantee automatic cache consistency or at least to provide mechanisms for controlling cache consistency. However, hardware manufacturers do not typically treat the translation lookaside buffer of a memory management unit as another type of cache which also must be kept consistent. None of the multiprocessors running Mach support TLB consistency. In order to guarantee such consistency when changing virtual mappings, the kernel must determine which processors have an old mapping in a TLB and cause it to be flushed. Unfortunately, it is impossible to reference or modify a TLB on a remote CPU on any of the multiprocessors which run Mach.

There are several possible solutions to this problem, each of which are employed by Mach in different settings:

1. *forcibly interrupt all CPUs which may be using a shared portion of an address map so that their address translation buffers may be flushed,*
2. *postpone use of a changed mapping until all CPUs have taken a timer interrupt (and had a chance to flush), or*
3. *allow temporary inconsistency.*

Case (1) applies whenever a change is time critical and must be propogated at all costs. Case (2) can be used by the paging system when the system needs to remove mappings from the hardware address maps in preparation for pageout. The system first removes the mapping from any primary memory mapping data structures and then initiates pageout only after all referencing TLBs have been flushed. Often case (3) is acceptable because the semantics of the operation being performed do not require or even allow simultaneity. For example, it is acceptable for a page to have its protection changed first for one task and then for another.

6. Integrating Loosely-coupled and Tightly-coupled Systems

The introduction of multiprocessor systems adds to the difficulty of building a "universal" model of virtual memory. In addition to differences in address translation hardware, existing multiprocessors differ in the kinds of shared memory access they make available to individual CPUs. Examples strategies are:

- *fully shared memory with uniform access times as in the Encore MultiMax and Sequent Balance,*

[3]The Multimax has however added special hardware to allow a full 4 gigabytes to be addressed

- *shared memory with non-uniform access as in the BBN Butterfly and IBM RP3 and*
- *message-based, non-shared memory systems as in the Intel Hypercube.*

As yet, Mach, like UNIX, has been ported only to multiprocessors with uniform shared memory. Mach does, however, possess mechanisms unavailable in UNIX for integrating more loosely coupled computing systems. An important way in which Mach differs from previous systems is that it has integrated memory management and communication. In a tightly coupled multiprocessor, Mach implements efficient message passing through the use of memory management "tricks" which allow lazy-evaluation of by-value data transmission. It is likewise possible to implement shared copy-on-reference [13] or read/write data in a network or loosely coupled multiprocessor. Tasks may map into their address spaces references to memory objects which can be implemented by pagers anywhere on the network or within a multiprocessor. Experimentation with this approach, which offers the possibility of integrating loosely and tightly coupled multiprocessor computing, is underway. A complete description of this work is currently being written up in [12]. Implementations of Mach on more loosely coupled multiprocessors are in progress.

7. Measuring VM Performance

Tables 7-1 and 7-2 demonstrate that the logical advantages of the Mach approach to machine independent memory management have been achieved with little or no effect on performance as compared with a traditional UNIX system. In fact, most performance measures favor Mach over 4.3bsd.

Performance of Mach VM Operations

Operation		**Mach**	**UNIX**
zero fill 1K (RT PC)		**.45ms**	**.58ms**
zero fill 1K(uVAX II)		**.58ms**	**1.2ms**
zero fill 1K(SUN 3/160)	**.23ms**	**.27ms**	
fork 256K (RT PC)		**41ms**	**145ms**
fork 256K (uVAX II)		**59ms**	**220ms**
fork 256K (SUN 3/160)	**68ms**	**89ms**	
read 2.5M file(VAX 8200)	**(system/elapsed sec)**		
first time		**5.2/11sec**	**5.0/11sec**
second time		**1.2/1.4sec**	**5.0/11sec**
read 50K file (VAX 8200)	**(system/elapsed sec)**		
first time		**.2/.3sec**	**.2/.5sec**
second time		**.1/.1sec**	**.2/.2sec**

Table 7-1:

The cost of various measures of virtual memory performance for Mach, ACIS 4.2a, SunOS 3.2, and 4.3bsd UNIX.

8. Relation to Previous Work

Mach provides a relatively rich set of virtual memory management functions compared to system such as 4.3bsd UNIX or System V, but most of its features derive from earlier operating systems. Accent [8] and Multics [7], for example, provided the ability to create segments within a virtual address space that corresponded to files or other permanent data. Accent also provided the ability to efficiently transfer large regions of virtual memory in memory between protected address spaces.

Obvious parallels can also be made between Mach and systems such as Apollo's Aegis [6], IBM's System/38 [5] and CMU's Hydra [11] -- all of which deal primarily in memory mapped objects. Sequent's Dynix [4] and Encore's Umax [10] are multiprocessor UNIX systems which have both provided some form of shared virtual memory.

Overall Compilation Peformance: Mach vs. 4.3bsd

VAX 8650: 400 buffers		
Operation	**Mach**	**4.3bsd**
13 programs	**23sec**	**28sec**
Mach kernel	**19:58min**	**23:38min**
VAX 8650: Generic configuration		
Operation	**Mach**	**4.3bsd**
13 programs	**19sec**	**1:16sec**
Mach kernel	**15:50min**	**34:10min**
SUN 3/160:		
Operation	**Mach**	**SunOS 3.2**
Compile fork test program 3sec	**6sec**	

Table 7-2:

Cost of compiling the entire Mach kernel and a set of 13 C programs on a VAX 8650 with 36 megabytes of memory under both Mach and 4.3bsd UNIX. Generic configuration reflects the normal allocation of 4.3bsd buffers. The 400 buffer times reflect specific limits set on the use of disk buffers by both systems. Also included is the cost of compiling the fork test program (used above) on a SUN 3/160 under Mach and under SunOS 3.2.

Mach differs from these previous systems in that it provides sophisticated virtual memory features without being tied to a specific hardware base. Moreover, Mach's virtual memory mechanisms can be used either within a multiprocessor or extended transparently into a distributed environment.

9. Conclusion

An intimate relationship between memory architecture and software made sense when each hardware box was expected to run its own manufacturer's proprietary operating system. As the computer science community moves toward UNIX-style portable software environments and more sophisticated use of virtual memory mechanisms[4] this one-to-one mapping appears less and less appropriate.

To date Mach has demonstrated that it is possible to implement sophisticated virtual memory management making only minimal assumptions about the underlying hardware support. In addition, Mach has shown that separation of machine independent and dependent memory management code need not result in increased runtime costs and can in fact improve overall performance of UNIX-style systems. Mach currently runs on virtually all VAX architecture machines, the IBM RT PC, the SUN 3 (including the virtual-address-cached SUN 3 260 and 280), the Encore MultiMAX and the Sequent Balance. All implementations are built from the same set of kernel sources. Machine dependent code has yet to be modified as the result of support for a new architecture. The kernel binary image for the VAX version runs on both uniprocessor and multiprocessor VAXes. The size of the machine dependent mapping module is approximately 6K bytes on a VAX -- about the size of a device driver.

10. Acknowledgements

The implementors and designers of Mach are (in alphabetical order): Mike Accetta, Bob Baron, Bob Beck (Sequent), David Black, Bill Bolosky, Jonathan Chew, David Golub, Glenn Marcy, Fred Olivera (Encore), Rick Rashid, Avie Tevanian, Jim Van Schiver (Encore) and Mike Young.

[4] e.g. for transaction processing, database management [9] and AI knowledge representation [2, 3]

References

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

[2] Bisiani, R., Alleva, F., Forin, A. and R. Lerner.
Agora: A Distributed System Architecture for Speech Recognition.
In *International Conference on Acoustics, Speech and Signal Processing*. IEEE, April, 1986.

[3] Bisiani,R. and Forin,A.
Architectural Support for Multilanguage Parallel Programming on Heterogeneous Systems.
In *2nd International Conference on Architectural Support for Programming Languages and Operating Systems*. IEEE, Palo Alto , October, 1987.

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

[5] French, R.E., R.W. Collins and L.W. Loen.
System/38 Machine Storage Management.
IBM System/38 Technical Developments, IBM General Systems Division :63-66, 1978.

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

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

[8] Rashid, R. F. and Robertson, G.
Accent: A Communication Oriented Network Operating System Kernel.
In *Proc. 8th Symposium on Operating Systems Principles*, pages 64-75. December, 1981.

[9] Alfred Z. Spector, Jacob Butcher, Dean S. Daniels, Daniel J. Duchamp, Jeffrey L. Eppinger, Charles E. Fineman, Abdelsalam Heddaya, Peter M. Schwarz.
Support for Distributed Transactions in the TABS Prototype.
In *Proceedings of the 4th Symposium on Reliability In Distributed Software and Database Systems*. October, 1984.
Also available as Carnegie-Mellon Report CMU-CS-84-132, July 1984.

[10] Encore Computing Corporation.
UMAX 4.2 Programmer's Reference Manual
Encore Computing Corporation, 1986.

[11] Wulf, W.A., R. Levin and S.P. Harbison.
Hydra/C.mmp: An Experimental Computer System.
McGraw-Hill, 1981.

[12] Young, M. W. et. al.
The Duality of Memory and Communication in Mach.
In *Proc. 11th Symposium on Operating Systems Principles*, pages . ACM, November, 1987.

[13] Zayas, Edward.
Process Migration.
PhD thesis, Department of Computer Science, Carnegie-Mellon University, January, 1987.