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

RICHARD RASHID, AVADIS TEVANIAN, JR., MICHAEL YOUNG, DAVID GOLUB, ROBERT BARON, DAVID BLACK, WILLIAM J. BOLOSKY, AND JONATHAN CHEW

Abstract—**Recent technological advances in memory management architectures, multiprocessor systems, and software architectures dictate a reevaluation of the virtual memory management support provided by an operating system. The problems posed by multiprocessor systems and the portability issues raised by the large variety of memory management units available have not been satisfactorily addressed by past virtual memory systems. In addition, increases in virtual memory functionality that can be provided by memory managed architectures have gone largely unnoticed by system designers.**

This paper describes the design, implementation, and evaluation of the Mach virtual memory management system. The Mach virtual memory system exhibits architecture indepedence, multiprocessor and distributed system support, and advanced functionality. The performance of this virtual memory system is shown to often exceed that of commercially developed memory management systems targeted at specific hardware architectures.

Index Terms—**Architecture independence, Mach, parallel operating systems, UNIX, virtual memory.**

I. 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 architectures. UNIX [18] systems have traditionally addressed the problem of virtual memory (VM) portability by restricting the facilities they 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 [10], 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.

Since the fall of 1984, CMU has been engaged in the development of a portable, multiprocessor operating system called Mach. One of the goals of the Mach project 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 and read-write memory sharing between
- 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.

Mach runs on a number of uniprocessors and multiprocessors including the VAX family of uniprocessors and multiprocessors [12], [3], the IBM RT PC family [24], the SUN 3 family, the Encore Multimax, the Sequent Balance 21000 [6], MIPS [13], the IBM 370 family [9], the BBN Butterfly Plus [19], and several experimental computers such as the IBM RP3 [16]. Despite differences between supported architectures, the machine-dependent portion of Mach's virtual memory subsystem has been kept relatively small. All information important to the management of Mach's virtual memory is maintained in machine-independent data structures and machine-dependent data structures which 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.

Manuscript received October 15, 1987; revised February 15, 1988.

R. Rashid, M. Young, D. Golub, R. Baron, D. Black, and J. Chew are with the Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213.

A. Tevanian, Jr., is with NeXT, Inc., Palo Alto, CA 94304.

W. J. Bolosky is with the University of Rochester, Rochester, NY.

IEEE Log Number 8821805.

II. 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 a collection of data provided and managed by a server which can be mapped into the address space of a task.

Mach is fundamentally a message passing communication kernel. 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 result 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 therefore 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.

Mach message passing and virtual memory management are intimately linked. Large message transfers are implemented using copy-on-write memory mapping techniques. This allows large amounts of data, including whole files and even whole address spaces, to be sent in a single message without the cost of data copying. In addition, the interface that Mach provides to memory management is implemented in terms of messages sent to the ports which represent tasks and memory objects. The use of port references provides both location transparency and protection. One consequence of the decision to link message passing and virtual memory is the fact that faults on nonmemory-resident memory object data result in messages sent to the backing-store port for that object. This permits memory objects to be implemented either by the kernel directly or by user-state programs since either could hold receive access rights to a memory object port.

III. The Programmer's View of Mach Virtual Memory

Each Mach task possesses a large address space that consists of a series of mappings between ranges of memory addressable 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 Gbytes of memory under Mach[1] while the VAX architecture allows at most 2 Gbytes 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,
- 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 programming interface that Mach provides to manage tasks and memory objects can be divided into four functional groups:

- *address space manipulation,* including the allocation and deallocation of virtual memory,
- *memory protection,* allowing flexible use of memory protection hardware,
- *memory inheritance,* which defines the sharing relationship between the address spaces of related tasks, and
- *miscellaneous primitives,* that formalize access to statistics maintained by the Mach kernel, access other task's virtual memory, and describe a task's address space.

Table I summarizes the interface based on these functional groups.

A. Address Space Manipulation Primitives

Mach treats an address space as a sequence of pages. The size of a logical page in Mach is not necessarily the same as the underlying hardware page size. Instead, it is a boot time parameter which can be any multiple of the hardware page size that is a power of two. This allows a system administrator to select a page size appropriate both to his machine type and mix of applications.

Each page in a task's address space is considered to be either allocated or unallocated. A reference to an unallocated page results in a memory protection violation. An allocated page may be directly addressed and the data it contains are derived from a memory object.

Virtual memory is allocated using the *vm_allocate* primitive and is deallocated using the *vm_deallocate* primitive.

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

TABLE I
MACH VIRTUAL MEMORY INTERFACE FUNCTIONAL GROUPS

Functional Group	Primitives
Address space manipulation	vm_allocate, vm_deallocate and vm_allocate_with_pager
Protection	vm_protect
Inheritance	vm_inherit
Miscellaneous	vm_read, vm_write, vm_copy, vm_region and vm_statistics

Allocated virtual memory does not necessarily consume system resources. By default, memory allocated by *vm_allocate* contains no data. Physical pages are not allocated until the virtual addresses in the range specified by *vm_allocate* are referenced at which time pages are allocated and filled with zeros. The *vm_allocate_with_pager* primitive is used to allocate a range of task addressable pages that are backed by a specific memory object. This feature of Mach allows user-state programs to provide their own implementation of backing storage for regions of virtual memory. Tasks which implement one or more memory objects are usually referred to as *external pagers*. A complete description of external pagers and their use in Mach can be found in [26].

B. Memory Protection Primitives

Just as memory allocation is performed at the page level, Mach's memory protection primitives are also defined to operate on ranges of pages. Each allocated page in an address space has two protection codes associated with it:

- its *current protection,* which corresponds to the protection code associated with a page for memory references, and
- its *maximum protection,* which limits the value of the current protection.

A page's protection consists of a combination of *read, write,* and *execute* permissions. Each type of permission is mutually exclusive. Proper enforcement of the protection codes depends on hardware support. Lack of hardware support may allow extra access to occur. For example, most memory management units (MMU's) do not distinguish between read access and execute access. Therefore, on such MMU's, it is possible to execute any readable instructions regardless of execute permissions.

Protection codes are set using the *vm_protect* primitive, which can be used to set either the current protection or the maximum protection. The current protection cannot be set to include a permission not set in the maximum protection. Further, the maximum protection may only be lowered. Once a permission is removed from the maximum protection code, it may never be added.

C. Memory Inheritance Primitives

A new address space is created in Mach as a side-effect of task creation. Task creation, and thus address space creation, is performed by the *task_create* primitive. *task_create* constructs a new address space which is either empty or based on an existing address space. When creating a new address space (child) based on an existing space (parent), each page in the new address space is based on the inheritance value of the corresponding page of the existing address space. Inheritance values can be specified on a per page basis and can take on one of three possible values:

VM_INHERIT_NONE: The page is not transferred to the child. The page is left unallocated in the child.

VM_INHERIT_COPY: The page is copied into the child. Subsequent modifications to the page affect the task making the modification only. For efficiency, pages are usually only copied when a write operation occurs (copy-on-write).

VM_INHERIT_SHARE: The page is shared between the parent and the child. Changes made to the page by either the parent or child will be seen by both.

By default, pages have an inheritance value of VM_INHERIT_COPY. The inheritance value of pages is set with the *vm_inherit* primitive. A task may freely set the inheritance value of any allocated pages it can access.

An example of the way in which inheritance might be used is the Mach implementation of the UNIX *fork* operation. *Fork* creates a child process which is a copy of its parent. In Mach, this is implemented by setting the inheritance value of the parent task to VM_INHERIT_COPY, creating the child address space, and then creating a thread of control for the child which begins executing at the location of the parent's program control counter at the time of *fork*. In addition to pure copy-on-write sharing of the parent's address space, however, Mach's inheritance primitives also allow read/write or no sharing of page ranges between a UNIX parent and its child processes.

D. Miscellaneous Primitives

While usually not necessary for normal memory manipulations, Mach provides a number of miscellaneous primitives which formalize memory management functions often needed for advanced applications.

For example, it it typically the case that a thread can only reference the memory corresponding to the task in which it executes. However, it is sometimes necessary to read or write the address space of another task. For example, a debugger needs to examine and modify the address space of the task being debugged. These operations may be performed using the *vm_read* and *vm_write* primitives. The *vm_copy* primitive may be used to efficiently copy a range of virtual memory within an address space using copy-on-write techniques.

Information specific to an address space may be determined with the *vm_region* call. This primitive returns a description of a region of an address space including protection values, inheritance values, and other pertinent information. Finally, the *vm_statistics* primitive provides a formal interface allowing tasks to query current virtual memory statistics maintained by the kernel.

These operations are secured against malicious use by unauthorized users due to the fact that the caller must have send access rights to the task ports of the tasks they manipulate. By default, only the creator of a task has such an access right, although that access right may be passed to other tasks in messages.

IV. The Implementation of Mach Virtual Memory

The design and implementation of Mach's virtual memory subsystem was dictated by several concerns:

- portability (i.e., architecture independence),
- flexible address space manipulation,
- multiprocessor support, and
- performance.

The need for portability led to a system structure in which machine-independent and dependent modules were clearly defined with strict interfaces between them. All key functions were implemented in a machine-independent fashion.

The desire for flexible address space handling directed the choice of address space management data structures. Simple, small data structures were used to represent the contents of an address space to reduce both the implementation complexity of address space operations and the amount of storage required.

Multiprocessor concerns resulted both in a user-visible design which exported the notion of read/write shared memory and in an object-oriented internal implementation style which allowed fine-granularity locking.

Performance was achieved through aggressive lazy-evaluation. The key to this approach is the fact that virtual memory operations are often ephemeral in their effect. For example, UNIX programs frequently allocate more data and stack space than they usually use. Message passing logically copies data, but it is seldom true that a receiving program changes a memory value it has received in a message. By postponing operations until their results are needed, a virtual memory system can often avoid performing them altogether.

A. Virtual Memory Data Structures

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 *memory object:* a unit of backing storage managed by the kernel or a user task,

3) 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, and

4) the *pmap:* a machine-dependent memory mapping data structure (i.e., a hardware-defined physical address map).

Not surprisingly, these data structures correspond roughly to various hardware or software concepts. The resident page table corresponds to a machine's physical memory. An address map corresponds to a task (in Mach) or a process (in UNIX). A memory object corresponds to a source for paging (a file, for example). Finally, a pmap corresponds to a hardware's representation of an address space (page tables, for example).

B. 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 the resident page table. These entries are indexed by physical page number. Each entry may simultaneously be linked into several lists:

- a *memory object list,*
- a *memory allocation queue,* and
- an *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,* a kernel thread that attempts to maintain a minimum number of free, clean pages available to user-state tasks. Fast lookup of the physical page associated with an object/offset 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 always 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.

C. Address Maps

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 (see Fig. 1).

Each address map entry contains 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. Operations that operate on a subset of a region corresponding to an entry usually result in that entry being split into two separate entries pointing to the same object.

The address map entry also contains pointers for the doubly linked list management, as well as the byte offsets of the start and end of the region that the entry represents. Fig. 2 summarizes the address map entry data structure.

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

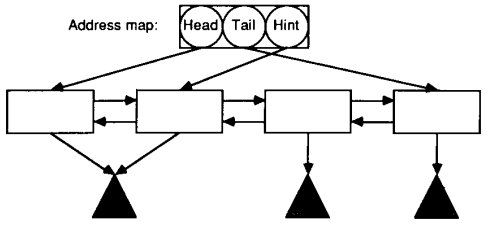


Fig. 1. A simple address map.

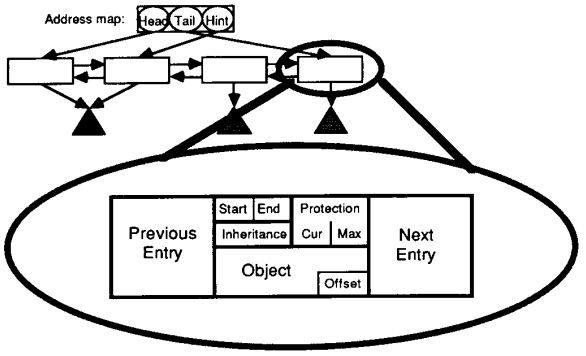


Fig. 2. An address map entry.

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 four mapping entries upon creation—one each for code, stack, initialized data, and uninitialized data.

Several alternative data structures were considered and rejected. The simplest alternative was an array of page table entries, similar to the VAX architecture. This solution makes both lookup and ordered traversal fast, but is not sufficiently compact, especially for sparse address maps. A multilevel page table can be made compact; however, it is potentially expensive to use when performing operations on large address ranges. Tree-structured address maps were also considered, but were rejected due to implementation costs (see [7]).

D. Memory Objects

A Mach address map need not keep track of backing storage. Instead, backing storage is implemented by Mach *memory objects*. Logically, a memory object is a contiguous repository for data, indexed by byte, upon which various operations (e.g., read and write) can be performed.

Data contained in a memory object can be mapped into a task address space by any of the following mechanisms:

- as the result of an explicit mapping, using, for example, the *vm_allocate_with_pager* primitive,
- as the result of a virtual copy operation, such as that which results from receiving a large message, in which the data in the object are accessible copy-on-write, or
- as the result of a Mach *task_create* operation or a UNIX *fork* operation, in which case the data can be either fully shared or accessible copy-on-write, depending on inheritance values.

Conceptually, a memory object represents some form of secondary storage. More specifically, the contents of a memory object are determined by the pager for that object. This could be anything from a UNIX file (in the case of a UNIX file system pager—see [22]) to a large database (in the case of a database disk manager acting as a pager—see [21]). The act of mapping part of an object into a virtual address map makes that data available for direct access within that address space.

The kernel acts as a cache manager for object data, using physical memory as a cache of the object's contents. References to data in an object that are not in the physical memory cache are translated into paging requests on the paper. As a result, the virtual memory object maintains information about those pages cached for each object (the resident page structures) as well as information on how to communicate with the pager (see Fig. 3).

As was pointed out in Section IV-B, resident pages are contained in only a single object. This is because each object defines a separate area of virtual memory. Sharing of physical pages occurs as the result of objects being mapped by several different address map entries.

E. Shadow Objects and Sharing Maps

When a copy-on-write copy is performed, the two address maps which contain the 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. This new page is the page 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*.

A shadow object collects and "remembers" modified pages which result from copy-to-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, creating what is termed a *shadow chain* (see Fig. 4). When

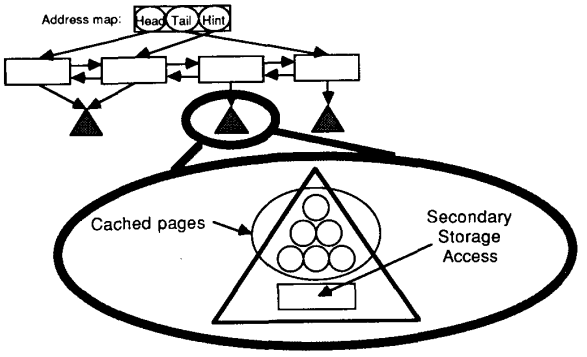


Fig. 3. A virtual memory object.

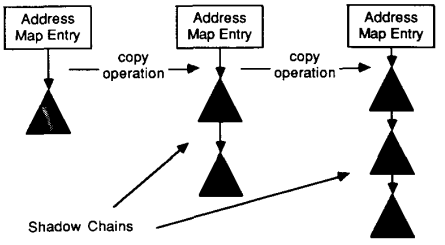


Fig. 4. Shadow chains as a result of virtual copies.

the system attempts to find a page in a shadow object, and fails to locate it, it follows this list of objects. The system will eventually 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 implement 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 on 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.

Sharing maps are created as part of the *task_create* operation whenever an inheritance value of SHARED is detected (see Fig. 5). Since sharing maps can be split and merged (using multiple entries within a map), sharing maps do not need to reference other sharing maps. For example, if a shared region is split into two new regions, one specifying that sharing takes place with a new task, the previous share map is simply split. Therefore, a sharing map is created only if absolutely necessary. This simplifies map operations and obviates the need for garbage collection of sharing maps.

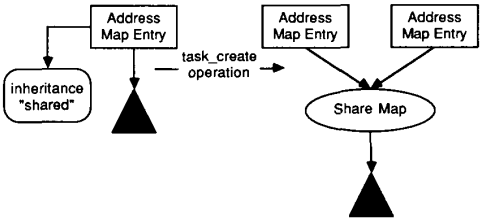


Fig. 5. Share map creation on task_create operation.

F. Managing the Object Tree

Much of the complexity of Mach memory management arises from a need to prevent the potentially large chains of shadow objects arising 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 are 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 that repeatedly forks its address space. This causes shadow objects to be built in a long chain ultimately pointing to the memory object that backs the UNIX stack.

As in the *fork* example, most cases of excessive shadow object chaining can be prevented by recognizing that new shadows often completely overlap (i.e., each page in the original has a copy in the shadow) 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 sometime 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 permit virtual memory operations to operate in parallel on multiple CPU's has resulted in complex object locking rules.

G. The Object Cache

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. Retaining the physical page mappings for such objects can make subsequent reuse very inexpensive. Mach maintains a cache of such frequently used memory objects.

The *object cache* is a least recently used (LRU) list of those objects that are not currently mapped but for which the kernel is retaining cached pages. The size of the LRU list is a fixed, boot time parameter, allowing those objects that are not frequently referenced to be uncached automatically. Whenever a cached object is mapped into an address space with the *vm_allocate_with_pager* primitive, it is removed from the LRU list, allowing other objects to be cached. When an object's reference count goes to zero, indicating no remaining mappings, and if it is flagged as a cacheable object, it is placed at

the end of the LRU list. If the list is full, objects are removed from the front of the list and their physical resources are uncached.

Most of the object cache control is performed automatically by the kernel. All externally managed objects are assumed to be cacheable, and cache control occurs transparently using the LRU list. However, pagers may also exercise all cache control functions. In particular, a pager may indicate whether or not an object should be cached on a zero reference count. In addition, a pager can flush an object from the cache if it is in the cache. This operation is typically used to indicate to the kernel that the cached data in the object are no longer valid. For example, the file system removes the object corresponding to a file whenever it is rewritten in order to flush stale cached pages.

H. The PMAP Interface

The *pmap* module is the sole machine-dependent module in the virtual memory implementation. The interface to this module has been designed to support a wide variety of hardware MMU's, and could even be considered for direct implementation in hardware or firmware. The interface assumes only a simple, paged MMU architecture, and has been shown to be effective for designs ranging from complex page tables (such as those found in the VAX or IBM RT PC architectures) to designs that employ a simple translation look-aside buffer (TLB) and no in-memory table (notably that found in the MIPS architecture). See Table II.

The implementor of a pmap module need not know any details of the machine independent implementation and data structures. A pmap structure is the handle with which the machine-independent code communicates with the machine-dependent code. All routines that operate on pmaps explicitly specify which pmap is to be operated on.

The pmap module is informed of address space creation and destruction with the *pmap_create, pmap_reference,* and *pmap_destroy* primitives. *Pmap_create* is called when the system begins using a new address space. The pmap module creates a new pmap structure and returns its handle to the machine-independent code. This handle is then used as an argument to other pmap routines in order to specify which pmap to affect. *Pmap_reference* and *pmap_destroy* increment and decrement reference counts on pmaps. When the reference count of a pmap goes to zero, the pmap module can free up all resources used by the pmap. For the most part, reference counts on pmaps never to go higher than once since each pmap corresponds to a single address map. However, there are some exceptions, most notably some uses of address maps for the kernel's address space management.

Address space specific calls are used by the machine-independent part of the system to notify the pmap module of various types of changes in virtual address mappings. *Pmap_enter* is the basic routine that is used to validate addresses. Address invalidation is performed by the *pmap_remove* routine. The *pmap_protect* routine is used to change the protection. Both *pmap_remove* and *pmap_protect* operate on a range of pages within a pmap. *Pmap_enter* operates on a single page of machine-independent size.

Some kernel functions cause multiple address spaces to be simultaneously affected. These functions are typified by the need to affect all current mappings to a physical page. There are two important examples of this type of function.

1) Page out operations require that all references to a physical page become invalidated.

2) Copy-on-write operations require that all writeable references to a physical page be changed to read-only.

To handle these cases, two primitives must be provided by the pmap module. The page out case is handled by the *pmap_remove_all* primitive. Given a physical page, this primitive invalidates all virtual mappings to that page. The copy-on-write case is handled by the *pmap_copy_on_write* primitive, which like the *pmap_remove_all* primitive affects all mappings of a page. However, in this case, rather than remove all mappings, the access for all mappings of the physical page is lowered to read-only.

The need for the global *pmap_remove_all* and *pmap_copy_on_write* primitives has important implications for the pmap module. In particular, if the pmap module is to allow multiple mappings of a physical page, then the pmap module must maintain some type of physical-to-virtual mapping list. Without such a list, the pmap module cannot properly implement the global operations.

While most of the pmap interface is used by the machine-independent part of Mach to inform the machine-dependent part of changes in mappings, there is also a primitive which allows the machine-independent code to query the status of various mappings. This primitive *pmap_extract* performs a virtual address translation on a specified virtual address and returns the corresponding physical address.

Other primitives supplied by the pmap module exist for the purposes of manipulating a machine's physical memory. Two primitives provide such support: *pmap_zero_page* zeros the specified physical page, *pmap_copy_page* copies one physical page to another. Again, the size of the page is the machine-independent page size, so the implementation must be prepared to copy multiple hardware page size pages.

The next part of the pmap interface consists of primitives that notify the pmap module of which address spaces are being used on which processors. These primitives allow the pmap module to set up hardware page table registers, or other machine specific hardware registers related to memory management. This information is also especially important for implementations that must perform some type of TLB consistency.

Finally, the pmap interface has a small number of optional primitives. Depending on the pmap implementation, these may perform no function. The first is *pmap_update.* This primitive informs the pmap module that all virtual mappings should now be in sync. This primitive allows other operations to be delayed until the sync time if there is some potential performance gain to be achieved. All primitives that affect mappings, except for *pmap_enter,* may be delayed until *pmap_update* is called. The other two miscellaneous primitives are *pmap_copy* and *pmap_pageable.* Both operate on a range of virtual addresses; *pmap_copy* indicates that a range of virtual memory has been copied using copy-on-write, and

pmap_pageable indicates that a range of virtual memory can (or cannot) be made pageable. These primitives are optional in that this information is given to the pmap module by using the other primitives (using *pmap_enter* for example). These miscellaneous primitives represent redundant information that may be used for performance optimizations.

I. The Page Fault Handler

The Mach page fault handler is the hub of the Mach virtual memory system. The kernel fault handler is invoked when the hardware tries to reference a page for which there is an invalid mapping or a protection violation. The fault handler has several responsibilities:

- *validity and protection*—The kernel determines if the faulting thread has the desired access to the address by performing a lookup in its task's address map. This lookup also provides a memory object and offset into that object.
- *page lookup*—The kernel attempts to find an entry for a cached page in the *virtual-to-physical* hash table. If the page is not present, the kernel must request the data from the pager.
- *copy-on-write*—Once the page has been located, the kernel determines if a copy-on-write operation is needed. If the task desires write permission and the page has not yet been copied, then a new page is created as a copy of the original. If necessary, the kernel also creates a new shadow object.
- *hardware validation*—Finally, the kernel informs the hardware physical map module of the new virtual-to-physical mapping.

With the exception of the hardware validation, all of these steps are implemented in a machine-independent fashion.

V. Performance

A. Uniprocessor Performance

Tables III and IV compare the performance of Mach's zero fill and *fork* operations with those of native UNIX implementations. In each table, the version of UNIX depends on the hardware base as follows: 4.3 BSD on the MicroVAX-II, SunOS 3.2 on the 3/160, and ACIS 4.2a on the RT/PC.

Mach outperforms each UNIX counterpart for zero fill data. On the MicroVAX, the large disparity in time is due primarily to 4K page size compared to 1K for 4.3 BSD. However, the minimum time required to zero 1K of data on the MicroVAX (using the bzero routine, for example) is 0.25 ms. This leaves 0.3 ms and 0.95 ms of overhead for the Mach and BSD fault handlers, respectively (per K). Since the Mach page size is 4K, the actual overhead is approximately 1.2 ms. This compares favorably to the 0.95 ms value for BSD, which uses the VAX page tables as its "machine-independent" data structure representation. Both SunOS and ACIS suffer additional overhead due to the need to translate from VAX page tables to their native counterparts.

Mach also outperforms each UNIX counterpart in implementing the UNIX *fork* primitive (see Table IV). Since Mach implements *fork* using copy-on-write, it avoids the copy costs that inflate the time to *fork* on UNIX systems that do not support copy-on-write. BSD-based systems try to avoid these extra costs with the *vfork* primitive, which eliminates the copy at the expense of different semantics. However, the time for a basic *fork* under Mach (24 ms on MicroVAX-II) is comparable to a BSD *vfork* (17 ms on MicroVAX-II).

TABLE II
PMAP INTERFACE PRIMITIVES

Functional Group	Primitives
Pmap Handles	pmap_create, pmap_reference and pmap_destroy
Range Operations	pmap_enter, pmap_remove, and pmap_protect
Global Operations	pmap_remove_all, pmap_copy_on_write
Query Operations	pmap_extract
Physical Memory	pmap_zero_page and pmap_copy_page
Use Information	pmap_activate and pmap_deactivate
Miscellaneous	pmap_update, pmap_copy and pmap_pageable

TABLE III
PERFORMANCE OF ZERO FILL PER K OF DATA

Machine	Mach	UNIX
DEC MicroVAX-II	.55ms	1.2ms
Sun 3/160	.25ms	.27ms
IBM RT/PC	.45ms	.58ms

TABLE IV
COST OF FORK/EXIT FOR A 256K PROCESS

Machine	Mach	UNIX
DEC MicroVAX-II	24ms	220ms
Sun 3/160	26ms	87ms
IBM RT/PC	19ms	145ms

B. Parallel Performance

Measurement of the parallel performance of the virtual memory system was performed on the Encore Multimax. The Multimax contained 16 National 32032 processors with a total of 32 Mbytes of primary memory.

Fig. 6 shows the parallel performance of manipulating data that are faulted. The data are manipulated in three significant ways:

- *bcopy (no fault):* Data are copied using the UNIX C library bcopy routine. This routine is hand-coded in Assembly language to copy bytes as fast as possible. In this test, data are simply copied, no page faults are generated.
- *bcopy (with fault):* Data are copied using the UNIX C library bcopy routine, but in this test accessing the source data generates page faults.
- *register copy:* This test moves each byte to a register. This has the effect of simulating the copy without generating excessive bus write operations. As in the previous test, references to data generate page faults.

The major contribution to the dropoff in performance with increasing numbers of CPU's is cache-thrashing on the Multimax. Each pair of CPU's share a data cache and, as the number of competing processors goes beyond eight, processors are competing for cache slots and interfering with each other's performance. This can be seen clearly in the case of the *bcopy (no fault)* curve which drops off even more dramatically in performance than the *bcopy (with fault)* curve. In any event, the amount of fault parallelism available is not a visible bottleneck on the Multimax with 16 processors. Moreover, assuming that faulted data are processed in some way, the cost of this processing is likely to dominate fault handling costs.

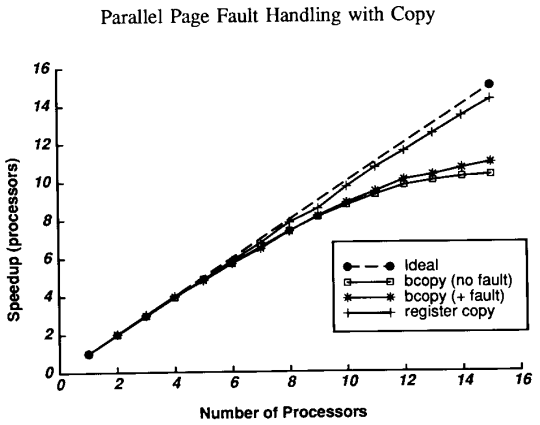


Fig. 6. Parallel performance of page fault operations (with copy) on Multimax.

C. Overall Performance

While the previous section showed that many low-level virtual memory functions perform better under Mach than other UNIX systems, this is of little importance if the user does not see an improvement in overall performance of higher level operations. Table V compares the performance of Mach to 4.3 BSD and SunOS. In each case, the time to compile a subset of the programs in/bin is reported (the subset corresponds to those programs that are compilable on both 4.3 and SunOS).

The VAX 11/780 used was running 4.3 BSD as released by Berkeley. Both the 4.3 kernel and Mach kernel were used on this system. The Sun 3/260 tests were performed on separate but identical Suns with the same compilation environment.

The most important information contained in Table V pertains to elapsed time and I/O operations. In both categories, Mach outperforms its UNIX counterparts. Mach reduces I/O operations by making more effective use of primary memory as a cache. For example, the object cache allows Mach to reuse programs cached in memory, even after those programs have terminated. The reduction in I/O operations has the direct effect of lowering the elapsed time.

D. Implementation Code Size

The size of the virtual memory implementation is summarized in Fig. 7. The total size, about 16K, does not include UNIX specific support, including support for paging to and from UNIX file systems. Also, implementation of the pmap module increases this total by an amount that depends on the complexity of the hardware.

The pmap module is about the size of a disk device driver with much of its implementation optional. The current VAX version is 8K bytes (compiled uniprocessor) and 12K bytes (compiled multiprocessor) of object code. The SUN implementation is 5K bytes. The IBM RT version is approximately 8.5K bytes.

Taken altogether, the virtual memory component of Mach is negligible when compared to the total size of operating systems such as 4.3 BSD or SunOS which can easily exceed 300–500K of code. In fact, it represents a small portion of the total amount of code required to emulate 4.3 BSD on a VAX (see Fig. 8).

TABLE V
OVERALL PERFORMANCE: COMPILING A SUBSET OF PROGRAMS IN/BIN

Machine	System	User Time (sec)	System Time (sec)	Elapsed Time (mm:ss)	I/O Reads/Writes
VAX 11/780	4.3 BSD	349.9	79.6	8:25	1686/1885
VAX 11/780	Mach	353.0	81.5	7:56	655/1571
Sun 3/260	SunOS 3.3	128.3	35.5	4:21	4231/1752
Sun 3/260	Mach	127.5	40.7	3:41	725/1506

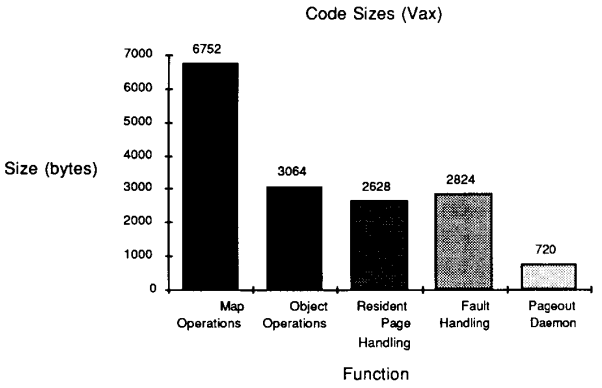


Fig. 7. The object size of the machine-independent virtual memory implementation broken down into modules (VAX architecture).

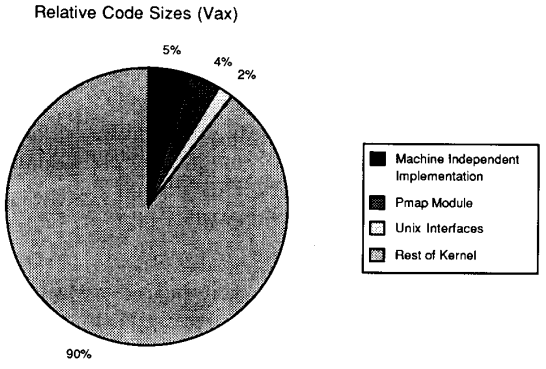


Fig. 8. Relative code sizes on a VAX.

VI. Hardware Experience

Mach runs on a large number of uniprocessors and multiprocessors. In the course of porting Mach to these machines, we gained considerable experience both with porting Mach's virtual memory code and with the pros and cons of various hardware memory management schemes as they apply to a system like Mach.

A. Porting Mach

Mach 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 implementation began in the fall of 1985. The first relatively stable VAX version was available in February 1986. At the end of that month the first port of Mach, to the IBM RT/PC, was initiated by a newly hired programmer who had not previously

worked on an operating system nor programmed in C. By early May of 1986, the RT/PC version was self-hosting and available to a small group of users.

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 was approximately three 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 and 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 4.2 BSD or System V UNIX. This has aided the porting effort significantly by reducing the effort required to build device drivers. Fig. 9 graphically illustrates the time frame in which several Mach ports have taken place.

Even after all of these porting efforts, the machine-independent implementation has never been modified in order to support a new hardware architecture. There were, however, two changes that were made as a result of porting efforts. The first was a two-line change to eliminate an autoincrement of a bit field which exercised a compiler bug in the RT/PC compiler. The only other change was to eliminate an informational print-out of some automatically computed paging parameters since it interfered with other diagnostic output generated by the Multimax.

B. Assessing Various Memory Hardware 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 translation look-aside buffer can be accommodated by Mach and would need little code to be written for the pmap module. In fact, a version of Mach has already run on a simulator for the IBM RP3 which assumed only TLB hardware support. In practice, though, the primary purpose of the pmap module is to manipulate hardware defined in-memory structures that in turn control the state of an internal MMU TLB. Each hardware architecture has shortcomings, both for uniprocessor use and even more so when bundled in a multiprocessor.

1) Uniprocessor Issues: Mach was initially implemented on the VAX architecture. Although, in theory, a full 2 Gbyte 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 Mbytes). UNIX systems have traditionally kept page tables in physical memory and simply limited the total process addressability to a manageable

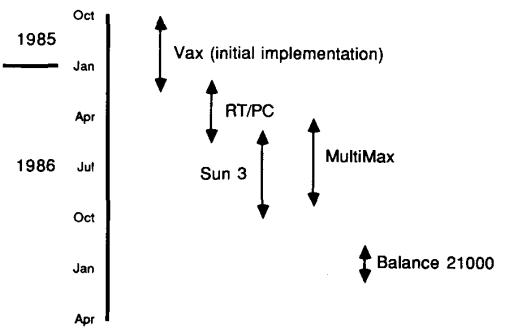


Fig. 9. The timing of the Mach implementation and several ports.

8, 16 or 64 Mbytes. 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 that were actually needed to 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 run time. 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.

As specified by the VAX architecture, each page table is broken down into two regions. Splitting the address space into two regions is attractive, especially for UNIX, since it is easy to cause the address space to grow in two separate directions. For UNIX, one direction corresponds to stack growth, the other corresponds to data segment growth. Unfortunately, these advantages do not apply to multithreaded Mach applications that need a stack for each thread.

The IBM RT/PC does not use per-task page tables [24]. Instead it uses a single inverted page table that 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 Gbyte 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 is that physical pages shared by multiple tasks, in Mach, 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 look-aside buffer. 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 Mbytes each. The use of segments and page tables makes it possible to implement sparse addressing reasonably, but only eight such *contexts* may exist at any one time. If there are more than eight active tasks, they compete for contexts, introducing additional page faults as on the RT.

Both the Encore Multimax and the Sequent Balance 21000 use the National 32082 MMU [14]. This MMU has posed several problems unrelated to multiprocessing.

- Only 16 Mbytes 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 Mbytes of physical memory may be addressed. Once again, this requirement is very restrictive in large systems.[2]
- 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 not surprising that these problems have been addressed in the successor to the NS32082, the NS32382.

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 look-aside buffer of a memory management unit as another type of cache which also must be kept consistent. None of the multiprocessors running Mach supports 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 has been employed by Mach implementations in different settings:

1) forcibly interrupt all CPU's 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 CPU's have taken a timer interrupt (and have had a chance to flush), or

3) allow temporary inconsistency.

Case 1) applies whenever a change is time critical and must be propagated 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 TLB's 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 if that protection is increasing. TLB's (or page tables) containing stale data will cause a protection fault that will be handled properly by the fault handler.

[2] The Multimax has, however, added special hardware to allow a full 4 Gbytes to be addressed.

VII. Relation to Previous Work

Mach provides a relatively rich set of virtual memory management functions compared to systems such as 4.3BSD UNIX or System V, but most of its features derive from earlier operating systems. Accent [17] and Multics [15], 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 [11], IBM's System/38 [8], and CMU's Hydra [25]—all of which deal primarily in memory mapped objects. Sequent's Dynix [4] and Encore's Umax [5] are multiprocessor UNIX systems which have both provided some form of shared virtual memory. 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.

VIII. 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[3] this one-to-one mapping appears less and less appropriate.

To date, Mach has demonstrated that it is possible to implement a sophisticated virtual memory management which

- is easily portable to paged architectures,
- provides support for parallel architectures,
- provides mechanisms for advanced use of virtual memory, and
- performs better than other designs targeted at specific hardware architectures.

The system has been ported to a variety of hardware architectures. At CMU alone, Mach has been ported to the VAX family of uniprocessors and multiprocessors, the IBM RT PC family, the SUN 3 family, the Encore Multimax, and the Sequent Balance 21000. In each case, the portability of the system has been demonstrated by the ease with which the pmap module has been implemented. Further, no porting effort has required any substantive change to the machine-independent implementation.

The needs of parallel processing are handled at two levels. First, flexible sharing and protection primitives provide support for high-performance parallel applications. Second,

[3] e.g., transaction processing, database management [20], and AI knowledge representation [2]

the virtual memory system implementation is fully parallel, allowing for efficient execution on parallel processors.

Mach provides for advanced use of virtual memory by providing powerful internal primitives that allow for virtually arbitrary mapping relations to be created. Shared memory and support for file mapping (with UNIX emulation) have all been provided.

Although one is often willing to trade performance in return for added functionality and/or portability, this performance tradeoff has been unnecessary in the Mach virtual memory system. In fact, the net effect of the advanced Mach design is a system that typically performs other designs.

Acknowledgment

The implementors and designers of Mach are (in alphabetical order): M. Accetta, B. Baron, B. Beck (Sequent), D. Black, B. Bolosky, J. Chew, D. Golub, G. Marcy, F. Olivera (Encore), R. Rashid, A. Tevanian, J. Van Schiver (Encore), and M. Young. For more detailed information on Mach and its memory management implementation and interface see [23].

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A new kernel foundation for UNIX development," in *Proc. Summer Usenix,* July 1986.

[2] R. Bisiani and A. Forin, "Architectural support for multilanguage parallel programming on heterogenous systems," in *Proc. 2nd Symp. Architectural Support Programming Languages Oper. Sys.,* Oct. 1987.

[3] *VAX Architecture Reference Manual,* Digital Equipment Corp., 1983.

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

[5] *UMAX 4.2 Programmer's Reference Manual,* Encore Computing Corp., 1986.

[6] G. Fiell and D. Rodgers, "32-bit computer system shares load equally among up to 12 processors," *Electron. Des.,* Sept. 1984.

[7] R. Fitzgerald and R. F. Rashid, "The integration of virtual memory management and interprocess communication in Accent," *ACM Trans. Comput. Syst.,* vol. 4, May 1986.

[8] R. E. French, R. W. Collins, and L. W. Loen, "System/38 machine storage management," *IBM Syst./38 Tech. Develop., IBM General Systems Division,* pp. 63–66, 1978.

[9] *3033 Processor Complex Theory of Operations, Vols. 1–5,* SY22-7001-SY22-7005, IBM. Corp., 1978.

[10] W. Joy *et al., 4.2BSD System Manual,* Tech. Rep., Comput. Syst. Res. Group, Comput. Sci. Divi., Univ. California, Berkeley, July 1983.

[11] P. L. Leach, P. H. Levine, B. P. Douros, J. A. Hamilton, D. L. Nelson, and B. L. Stumpf, "The architecture of an integrated local network," *IEEE J. Select. Areas Commun.,* vol. SAC-1, pp. 842–857, Nov. 1983.

[12] H. Levy *et al., Computer Programming and Architecture - The VAX-11.* Bedford, MA: Digital, 1980.

[13] J. Moussouris *et al.,* "A cmos risc processor with integrated system functions," in *Proc. COMPCON,* San Francisco, CA, Mar. 1986, pp. 126–131.

[14] *Series 32000 Databook,* National Semiconductor, Inc., 1984.

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

[16] G. Pfister *et al.,* "The IBM research parallel processor prototype (RP3): Introduction and architecture," IEEE, 1985.

[17] R. F. Rashid and G. Robertson, "Accent: A communication oriented network operating system kernel," in *Proc. 8th Symp. Oper. Syst. Principles,* Dec. 1981, pp. 64–75.

[18] D. M. Ritchie and K. Thompson, "The Unix time-sharing system," *Commun. ACM,* vol. 17, pp. 365–375, July 1974.

[19] C. Russell and P. Waterman, "Variations on Unix for parallel-processing computers," *Commun. ACM,* vol. 30, pp. 1048–1055, Dec. 1987.

[20] A. Spector *et al.,* "Support for distributed transactions in the tabs prototype," in *Proc. 4th Symp. Reliability Distributed Software Database Syst.,* Oct. 1984.

[21] A. Z. Spector, D. Duchamp, J. L. Eppinger, S. G. Menees, and D. S. Thompson, "The Camelot interface specification," Camelot Working Memo 2, Sept. 1986.

[22] A. Tevanian, R. Rashid, M. Young, D. Golub, M. Thompson, W. Bolosky, and R. Sanzi, "A Unix interface for shared memory and memory mapped files under Mach," in *Proc. Summer Usenix,* June 1987.

[23] A. Tevanian, Jr. "Architecture-independent virtual memory management for parallel and distributed environments: The Mach approach," Ph.D. dissertation, Carnegie-Mellon Univ., Dec. 1987.

[24] F. Waters, Ed., *IBM RT Personal Computer Technology,* International Business Machines Corp., 1986.

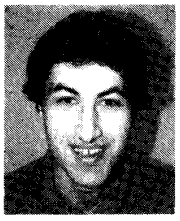
[25] W. Wulf, R. Levin, and S. P. Harbison. *Hydra/C.mmp: An Experimental Computer System.* New York: McGraw-Hill, 1981.

[26] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron, "The duality of memory and communication in the implementation of a multiprocessor operating system," in *Proc. Symp. Oper. Syst. Principles,* Nov. 1987.

Richard Rashid graduated with honors in mathematics from Stanford University, Stanford, CA, in 1974. He received the M.S. and Ph.D. degrees in computer science from the University of Rochester, Rochester, NY, in 1977 and 1980, respectively.

He is an Associate Professor of Computer Science and has been on the faculty of Carnegie-Mellon University, Pittsburgh, PA, since September 1979. While at the University of Rochester, Dr. Rashid participated in the design and implementation of the RIG operating system and Rochester Virtual Terminal Management System. Since joining Carnegie-Mellon, his responsibilities have included the direction of the CMU Distributed Sensor Testbed project, CMU's distributed personal computing project (SPICE), and the Mach multiprocessor operating system project. He is responsible for the design and implementation of a network interprocess communication facility for UNIX, the Accent network operating system kernel, and the Mach multiprocessor operating system. He has also participated in the design of the CMU ITC VICE/VIRTUE distributed file system.

Dr. Rashid is a past member of the DARPA UNIX Steering Committee and CSNet Executive Committee. He is a current member of the DARPA Distributed Systems Architecture Board and he is the current chairman of the ACM System Awards Committee.

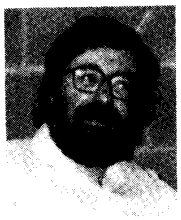


Avadis Tevanian, Jr., received the B.A degree from the University of Rochester, Rochester, NY, in 1983, and the M.S. and Ph.D. degrees in computer science from Carnegie-Mellon University, Pittsburgh, PA, in 1985 and 1987, respectively.

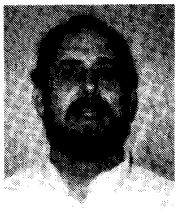
He is currently Chief Operating System Scientist at NeXT, Inc. He has worked on the Mach system since its inception—being one of the original designers and a major contributor to its implementation. He has worked on many ports of the Mach system and has written many papers and lectured extensively on various aspects of Mach.

Michael Young is a Ph.D. candidate in the Department of Computer Science at Carnegie-Mellon University, Pittsburgh, PA.

His current work is on an interface that allows user programs to control the management of permanent memory in the Mach operating system.



David Golub has been a Senior Research Programmer on the Mach project since 1986. He formerly worked at PERQ Systems, where he did extensive work on the Accent operating system, a prototype for much of the Mach work.



Robert Baron received the Bachelor's, Master's and Engineer's degree in electrical engineering from the Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, in 1977.

He has been at Carnegie-Mellon University, Pittsburgh, PA, since 1983, originally working on the Accent project and working on Mach since 1984. His research interest are in multiprocessors. From 1977 to 1983, he was with the Bell Telephone Laboratories. Initially involved in the development of a telephone service representative support system using UNIX. Later he worked for the UNIX support organization. At the Massachusetts Institute of Technology, he was at the Laboratory for Computer Science and worked on high-level languages.

Mr. Baron is a member of Sigma Xi, Eta Kappa Nu and Tau Beta Bi.

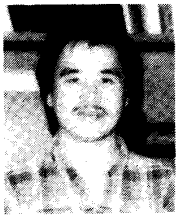
David Black received the B.A. and M.A. degrees in mathematics and the B.S.E. degree in computer science and engineering from the University of Pennsylvania, Philadelphia, and the M.S. degree in computer science from Carnegie Mellon University, Pittsburgh, PA

He is a graduate student and Ph.D. degree candidate in the Department of Computer Science, Carnegie Mellon University. His primary research is in the area of operating systems with particular interests in multiprocessors and scheduling issues. He has been involved with the Mach project since 1985.

Mr. Black is a member of Phi Beta Kappa, Tau Beta Pi, Eta Kappa Nu, Pi Mu Epsilon, and Sigma Xi.

William J. Bolosky received the B.S. degree in mathematics from Carnegie-Mellon University, Pittsburgh, PA, in 1985.

He is a graduate student in Computer Science at the University of Rochester, Rochester, NY, with interests in multiprocessor operating systems. He was formerly employed by the Carnegie-Mellon University Computer Science Department, where he worked on Mach's external paging and did the Mach IBM PC/RT port.



Jonathan Chew received the B.A. degree in applied math from the University of California, Berkeley, in 1983.

He is a Research Systems Programmer for the Mach Project. He has been working in the Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, since October 1985. Before that, he worked on Accent at PERQ Systems Corporation.