Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments: The Mach Approach

Avadis Tevanian, Jr.
December 1987
CMU-CS-88-106

Department of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Submitted to Carnegie Mellon University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

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 N00039-87-C-0251.

To my parents
and my brothers.

Acknowledgements

I would like to thank the many people who have helped me to finish this thesis research. Rick Rashid, my advisor, provided a great deal of help in realizing the final product as well as good advice all throughout my years as a graduate student. I would also like to thank the other members of my thesis committee: Alfred Spector, Eric Cooper and Dennis Ritchie.

There are many others who have helped in various ways. Mike Young and Dave Golub provided much assistance with the virtual memory implementation. Others, including David Black, Bob Baron, Bill Bolosky, Rich Sanzi and Jonathan Chew provided help as well. All helped with Mach as a whole and have contributed to making Mach a success. Mike Accetta, Glenn Marcy and Rudy Nedved provided interesting points of view on design decisions. Catherine Copetas provided help by carefully proofreading this document. Sid Chatterjee helped with LaTeX.

Finally, I would like to thank my family. My parents offered continuous support for my work. My brothers helped me to realize there is more to life than computers.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Recent technological advances in memory managed architectures, multiprocessor systems and software architectures dictate a re-evaluation 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 thesis describes the design and implementation of a new, portable, memory management system and evaluates it by direct comparison with commercially developed memory management systems. It is in production use at CMU, a number of industrial organizations and research laboratories, and is a central component of the Mach operating system, a distributed and parallel operating system developed at Carnegie Mellon University. At the time of this writing, it runs on more than half a dozen uniprocessors and multiprocessors including the DEC VAX family of uniprocessors and multiprocessors, the IBM RT/PC, the SUN-3, the Encore Multimax, the Sequent Balance 21000 and a number of experimental computers.

The implementation and evaluation of Mach, described in this thesis, demonstrates that it is possible to build a virtual memory system that exhibits:

- *architecture independence*: A wide range of paged architectures can be supported by carefully separating all data structures and algorithms into machine-independent and machine-dependent sections.
- *multiprocessor and distributed system support*: Features important in the implementation of parallel and distributed systems, such as shared memory and integrated memory management and message passing, can be provided.
- *advanced functionality*: A small set of general memory management mechanisms can

be used to implement features such as copy-on-write, shared libraries, mapped files and user-implementable paging objects.

Moreover, these features can be realized without an adverse effect on performance. In fact, the performance of this virtual memory system is shown often to exceed that of commercially developed memory management systems targeted at specific hardware architectures.

1.1 Architecture Independence

Traditionally, highly portable operating systems have been based on simple memory management schemes. One of the reasons UNIX[28,6] has been so portable, for example, is that its virtual memory system has traditionally provided little functionality beyond separate address spaces and the ability to use more virtual memory that is physically available. This lack of functionality allowed it to be ported to widely available, non-memory managed architectures. Advances in hardware design, however, have resulted in the ability to provide memory management at such a small cost that the omission of such a feature is now unlikely in all but the most inexpensive computer systems. In fact, virtual memory management is increasingly becoming a standard feature — defined as part of a processor architecture and implemented in processor chips such as the DEC MicroVAX[15], Motorola 68030[33] and Intel 386[25].

Sophisticated virtual memory designs, in contrast, have usually been tied to a single computer. Systems such as Accent[41] and Multics[9], for example, provided many advanced memory management functions, but were targeted at specific hardware configurations. The recent proliferation of hardware memory management schemes has doomed such single system approaches, necessitating a design and implementation that is decoupled from any particular memory management unit. For example, although the VAX, 68030 and 386 all provide memory management as a standard feature, they do so in dissimilar ways. The VAX provides flat page tables[16,17], the 68030 provides multi-level page tables[33] and the 386 provides a combination of segments and page tables[24].

The Mach virtual memory design achieves portability across a wide range of architectures without sacrificing functionality. Few assumptions are made about the availability of particular memory management hardware. Mach's internal data structures are not patterned after any specific architecture. Instead, they are derived from basic assumptions about how virtual memory is used, motivated later in this thesis.

To deal with a wide variety of hardware memory management schemes, it is necessary for the data structures and algorithms used to implement Mach virtual memory to be separated cleanly into a machine-independent part and a machine-dependent part. Mach's data structures are designed in such a way that all information about the virtual memory system is maintained in machine-independent form. The hardware's representation of virtual memory is effectively a cache of the machine-independent data structures. Taking this approach greatly simplifies writing machine-dependent modules. It should be possible to implement Mach virtual memory on any hardware memory management unit that can generate recoverable faults upon the access of a page for which it has no valid mapping.

Finally, the key to a successful solution to this problem is a clear, simple definition of machine-dependent routines that must be provided for each hardware architecture. A machine-dependent implementation is required to manage hardware physical address maps (*pmaps*). For a machine like the VAX, a pmap corresponds to a VAX page table. On the other hand, for a machine like the RT/PC, a pmap corresponds to a set of allocated segment registers. The machine-dependent section is also responsible for implementing page level operations on pmaps. This interface can be reasonably implemented on both purely paged and paged/segmented architectures.

1.2 Multiprocessor and Distributed System Support

The design of the virtual memory system's internal data structures and algorithms was influenced by the types of hardware configurations (as opposed to memory management architectures) currently available as well as those likely to be available in the future. Several major changes in the types of hardware configurations have taken place since virtual memory systems were first built. For example:

- multiprocessor systems — both tightly coupled and loosely coupled multiprocessors are being commercially developed. These systems make it necessary to support sharing of virtual memory and require a parallel implementation.

- distributed systems — most computer installations consist of many connected nodes forming high speed local area networks. A distributed system needs the ability to efficiently transfer data between nodes on a network.

- large memories — machines with many megabytes of primary memory are increasingly available due to the continuing improvements in memory chip density. Large memory sizes require that the underlying data structures be able to represent an address space compactly as well as manage the available physical memory resources efficiently.

Many virtual memory systems were designed before these types of computer systems were available and therefore never addressed issues important to these types of systems. Even advanced designs, such as the one found in Accent, do not address all of these issues.

1.3 Advanced Functionality

The design described in this thesis provides mechanisms that allow for advanced virtual memory functionality. At the heart of this functionality is modular access to low-level paging functions such as page-in and page-out. Virtual memory functionality, such as mapped files, is easily implemented using these mechanisms.

The Mach virtual memory design and implementation supports large, sparse address spaces, necessary for large Artificial Intelligence applications such as vision and speech understanding. Address space fragmentation that occurs as the result of advanced use of virtual memory, such as mapped files and copy-on-write message passing, is handled efficiently by a judicious choice of data structures and algorithms. Flexible sharing and protection primitives allow parallel applications to share data without the need to sacrifice protection.

All virtual memory primitives are supported on all hosts that run Mach, easing the creation of distributed applications. If special-purpose hardware features, available only on some machines, had been exported to the user, applications depending on those features would have become machine-dependent. Such applications would not be distributed easily in an environment of heterogeneous machines. Further, the exported primitives restrict read/write sharing to an inheritance mechanism, avoiding the need for the kernel to address issues related to network shared memory. The exported interface can be implemented reasonably on any paged architecture.

1.4 Overview of Thesis

Since the virtual memory design is a central component of the Mach operating system, it is necessary to be familiar with Mach as a whole. Chapter 2 provides an introduction to Mach including its underlying philosophy, abstractions and functions.

After the short background on Mach, the Mach virtual memory interface is both motivated and presented in Chapter 3. Next, the implementation of the virtual memory system is described in Chapter 4. This includes detailed descriptions of key data structures and algorithms used in the machine-independent implementation as well as interfaces to machine-dependent code and other kernel interfaces.

Chapter 5 examines some applications of the virtual memory system and Chapter 6 reports on its performance. These chapters validate the utility of the system. Next, Chapter 7 discusses experience gained by working with various flavors of memory management. The impact of a variety of memory architectures, both uniprocessor and multiprocessor, is examined in Chapter 8.

The thesis concludes with Chapter 9 which examines related work in the area and Chapter 10 which presents a summary of the thesis. A discussion of possible future work is also provided in Chapter 10. Several appendices provide detailed information on topics discussed throughout the thesis.

Chapter 2

Mach Background

This chapter presents a brief overview of the Mach kernel. The underlying philosophy, abstractions and functions of the Mach design are examined. In addition, Mach's relation to UNIX is described. More details on Mach can be found in the references[1,51].

2.1 Mach Philosophy

The Mach operating system design is typified by the following three attributes:

simplicity: The kernel exports only a small number of user visible abstractions. Keeping this number small results in a simpler kernel as well as simpler user programs. Many operating system functions are implemented outside the kernel in user programs rather than inside the kernel proper.

extensibility: It is possible to export kernel functions to user state. This allows user programs to implement many of the functions performed by the kernel itself. More importantly, user programs can implement these functions in ways that the kernel could not. For example, transparent network message passing is implemented by a user program in Mach.

integration: All of Mach's basic abstractions are designed to work together.

The Mach design anticipates the following computer architectures:

- multiprocessors — Multiprocessor hardware architectures are already commercially feasible. Many improvements in hardware design can be expected in this area in each of the three major multiprocessor types: UMA, NUMA and NORMA (described

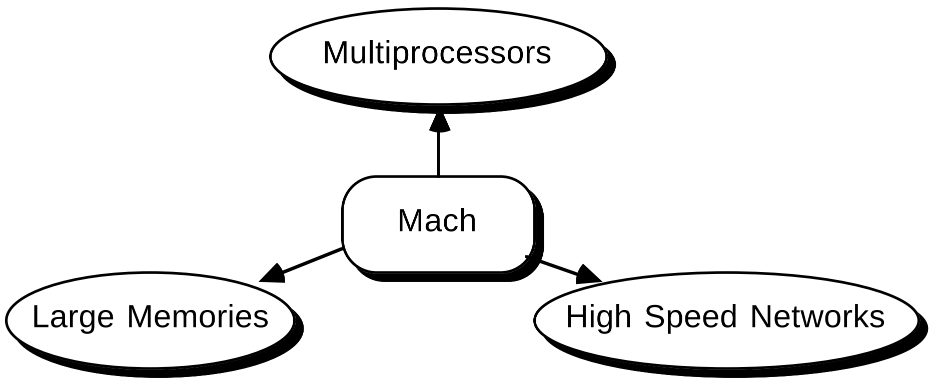

Figure 2.1: Mach anticipates future computing technologies

later in Chapter 8). As software engineers gradually learn how to use these machines effectively, the multiprocessor architectures will no doubt increase their role as general computing resources.

- large memories — With the seemingly endless improvements in low cost, dense memory chips, it is clear that machines will have, at a minimum, many megabytes of primary memory. Many machines will have hundreds of megabytes, and even gigabytes, of main memory. Large memories impact system design in two important ways:

  1. Algorithms that manage memory must be able to efficiently scale from a few megabytes to a few gigabytes.
  2. Using main memory as a large cache of secondary storage, always an attractive system feature, becomes even more necessary in order for a system to perform as well as possible.

- high speed networks — Improvements in network technology, especially in available bandwidth, are likely to spur interest in using network connected computers in tightly coupled ways. Current technology, with low bandwidth compared to internal bus structures, makes tightly coupled use of loosely coupled machines very inefficient.

The central component of a Mach-based operating system environment is the Mach kernel. Typical operating system services are layered above the Mach kernel as a set of system servers. Since the Mach kernel is a simple base for system servers, it is readily portable and adaptable to the wide range of computing architectures present today and anticipated in the future. Since servers provide most of the traditional system services, different software environments can be run easily in different hardware environments.

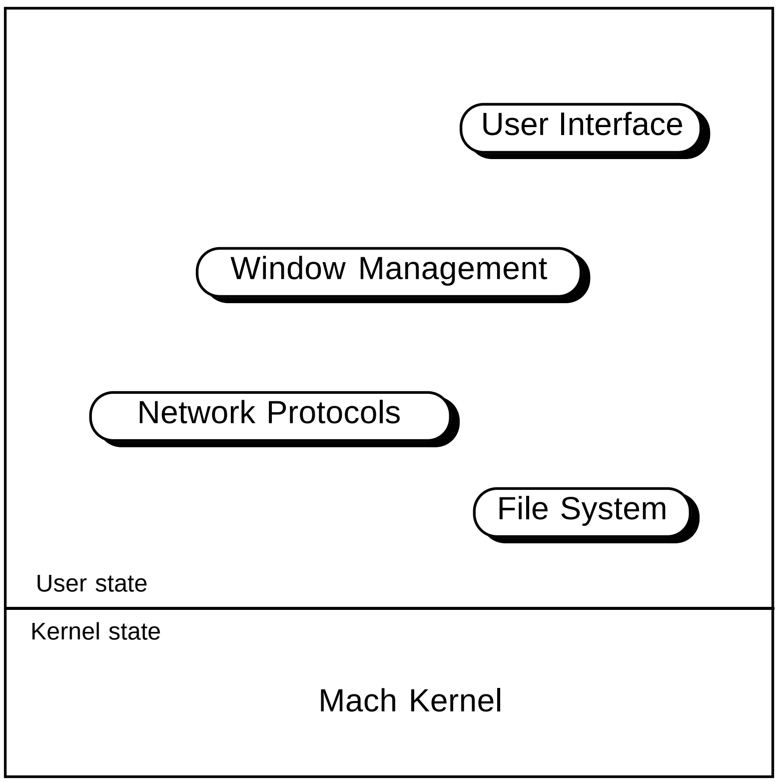

Figure 2.2: Mach Operating System philosophy.

Execution and Memory Management	
task	The unit of resource allocation: large address space and port rights.
thread	The unit of CPU utilization.
memory object	Internal unit of memory management.
Communication	
port	A protected communication channel, accessible only via send and receive capabilities.
message	A typed collection of data objects.

Table 2.1: Mach kernel abstractions.

Since the Mach kernel is a simple, extensible, communication-based kernel, it is well suited for use in a distributed environment. Object oriented programming can be performed with protected communication channels as object references. An interface specification language (MiG) can be used to specify message interfaces without the difficulties of programming at a message level[27].

2.2 Kernel Abstractions and Functions

The Mach kernel exports only five abstractions, the *task*, *thread*, *port*, *message* and *memory object*. These are summarized in Table 2.1.

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). 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. The traditional notion of a *process* is represented by a task with a single thread of control.

A port is a communication channel – logically a queue for messages protected by the kernel. Ports are the reference objects of the Mach design. That is, objects managed by the kernel, such as tasks and threads, are represented to the user as ports. Ports are used in much the same way that object references are used in an object-oriented system. *Send* and *Receive* are the primitive operations on ports.

Communication is accomplished by sending messages to ports. 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. Port capabilities include:

- *send* rights, which correspond to the capability to send a message to a port. Send rights may be held by any number of tasks.

- *receive* rights, which correspond to the capability to receive a message on a port. Receive rights may be held by only one task.

- *owner* rights, which correspond to the capability to gain receive rights on a port when it is deallocated. Like receive rights, owner rights may be held by only one task.

Since receive and owner rights may be held by only one task, the act of sending these capabilities in a message causes that capability to be deleted from the sending task. However, sending send rights in a message does not automatically remove that right from the sender. Since the only way to gain a capability is through message passing operations, the Mach kernel guarantees secure communication within a host.

The Mach kernel automatically queues messages for tasks executing on its machine. However, transmission of messages between separate Mach hosts is performed transparently by an intermediate server task. This server task, known as the *network message server*, maintains mappings of local "proxy" ports to global "network" ports. It forwards messages using network protocols of its choice. In addition, it is free to make decisions related to security, or lack of it, depending on the environment in which it is run[44].

Mach provides a separate, paged address space to each task. Each address space is represented internally as a collection of memory objects. Each memory object is a repository of data which can be mapped into the address space of a task. The data for a memory object may be provided and managed by the kernel or user applications.

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 operations performed on the object it represents.

The indirection provided by message passing allows objects to be placed arbitrarily 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 compared to conventionally implemented operating systems. One of the keys to efficiency in Mach message passing 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.

2.3 Mach and UNIX

The Mach kernel, in its purest form, provides only the basic primitives necessary for building distributed and parallel applications. A complete operating system environment suitable for use by developers or end users is provided as a layer on top of the Mach kernel. Since Mach makes few traditional operating system decisions within the kernel itself, it is possible to provide completely different operating system environments.

The current implementation of the Mach kernel exists as the underpinning for a 4.3 BSD UNIX compatible system. In this system, the Mach kernel implements the features particular to an operating system kernel as well as the features provided by the Mach kernel interface. UNIX compatibility is provided by the original 4.3 BSD implementation — modified for use with Mach. In effect, many of the internals of 4.3 BSD have been replaced with the Mach equivalents.

Although much of the original 4.3 source remains unchanged, two important types of change have taken place:

1. Major changes in which entire portions of 4.3 have been replaced, e.g., virtual memory management and processor scheduling.
2. Minor changes in which small changes have been made to 4.3 code to handle the different interfaces provided by Mach, e.g., device drivers and configuration code.

The net result is a system that is not only binary compatible with 4.3 BSD, but is even bug for bug compatible in almost all features. This has led to the ability to run virtually all vanilla 4.3 programs, thus supporting 4.3-based developers and users.

Although such high compatibility with 4.3 is desirable, it has had its drawbacks:

- A large effort is required to assure that the compatibility code still runs after Mach related changes are made. This is necessary because the original UNIX code was not designed to run in a Mach-like environment. Thus, many assumptions made by UNIX kernel code are not valid in a Mach environment (e.g., multiple threads with simultaneous access to a single resource).
- The interfaces to the services provided by the Mach kernel, such as virtual memory management, are radically different than what is assumed by UNIX kernel code.

> Therefore, it has been necessary to modify those parts of UNIX that used the older interfaces.

The fact that such a complete emulation of 4.3 exists for Mach has allowed interested parties to experiment with Mach without giving up their familiar development environments. Further, it is possible to make direct performance comparisons between the Mach virtual memory system and the BSD virtual memory system, as well as other versions of UNIX derived from BSD UNIX.

Chapter 3

Virtual Memory Interface

From a user's perspective, the most important aspect of the Mach Virtual Memory design is the exported interface. Many factors have influenced the set of primitives exported to the user in support for virtual memory functionality. The virtual memory primitives exported by Mach have been carefully chosen to allow maximum functionality for the user without compromising the ability to use the primitives in a distributed environment or posing serious portability problems.

In this chapter, the virtual memory functionality exported by Mach is both motivated and described. Potential uses of virtual memory and some of their related potential pitfalls are examined.

3.1 Virtual Memory Needs

One of the simplest functions performed by a virtual memory system is that of basic paging. Paging systems have existed for years and provide the ability for programs, or sets of programs, to utilize more memory than is physically available in a single machine. When the memory utilization on a machine exceeds the physical memory in that machine, the virtual memory system uses a *page replacement algorithm* to move some pages to secondary storage (pageout), making those pages available for use. When the pages that were moved to secondary storage are needed, the virtual memory system moves them from secondary storage into primary memory (pagein).

Basic paging support represents the lowest level of support that is provided by the virtual memory system. While it was a sufficient level of support for past software architectures, it is no longer sufficient for modern computing systems. In addition to basic paging support, the Mach virtual memory system provides the following:

- *consistent virtual memory interface*: The Mach virtual memory interface is supported on all Mach hosts.

- *parallel processing*: Flexible protection primitives and the ability to share data aid the creation of efficient parallel and distributed applications. Sharing can be achieved without giving up protection.
- *modular paging*: The paging operations are logically abstracted out of the virtual memory system. Different paging mechanisms can be added easily to the system.
- *simplicity*: The Mach virtual memory interface is designed to be as simple as possible without sacrificing functionality.

Underlying these features is the support for large, potentially sparse virtual address spaces. Large Artificial Intelligence systems, such as vision and speech understanding applications, process large amounts of data. In addition, advanced use of virtual memory, such as mapped files and copy-on-write message passing of large amounts of data, cause fragmentation of address spaces, potentially resulting in sparse use of virtual address spaces.

3.1.1 Consistent Virtual Memory Interface

Exporting a consistent virtual memory interface to the user, regardless of particular machine architectures, aids application developers in the creation of distributed applications. Since the same interface is supported on all Mach hosts, components of a distributed system can be placed on any Mach host and be guaranteed a consistent set of virtual memory primitives. If special purpose hardware features, available only on some machines, were to be exported to the user, applications depending on those features would become machine-dependent. Such applications would not easily be distributed in an environment of heterogeneous machines.

The exported interface can be implemented on any paged architecture. Depending on the actual hardware, some functions will have efficient implementations. Some functions, while still implementable, may suffer some performance degradation on certain architectures. For example, machines which do not allow virtual address aliasing (multiple virtual addresses for a single physical address) will cause extra page faults to be generated.

3.1.2 Parallel Processing

Multiprocessors are commercially feasible to build. Perhaps more importantly, researchers are finding multiprocessors to be valuable vehicles for harnessing computational horsepower. A virtual memory system that addresses the issues of virtual memory support in a parallel environment allows efficient parallel applications to be built on multiprocessor architectures.

Support for multiprocessors occurs at two levels in the virtual memory design. First, at the application level, a virtual memory interface that allows efficient data sharing can provide support for parallel applications. A lack of efficient data sharing may cause highly parallel applications to incur unnecessary overhead in communication costs. Second, at the system level, a parallel virtual memory implementation can exploit the parallelism available in a multiprocessor system. If the implementation is not parallel, it is likely to be a system bottleneck in large systems.

It is clear that some form of shared memory should be provided for certain specialized applications which intend to use multiprocessing for computational horsepower. Although the Mach task and thread mechanism does provide a means for many computational entities to share memory, it does have its pitfalls:

- Not all multiprocessors allow for efficient parallel execution of threads within the same task on multiple processors. This is a potentially serious problem on multiprocessors that do not provide cache consistency on shared memory. On such machines, it may be necessary to make an entire task's writable address space uncached.

- Threads within a task are completely unprotected from each other. All threads within a task have access to all of that task's resources. Some applications require protection between their computing entities.

While shared memory is most likely to be used for parallel processing applications, it is important to remember that the virtual memory system also supports distributed applications. Keeping this in mind, the primitives that allow for shared memory are designed to work in both the parallel and distributed domains.

The main problem with shared memory mechanisms and distributed computation is that a poor choice of primitives can result in sharing relationships in which tasks on separate hosts attempt to share memory. If this is allowed, then to provide consistent semantics, the Mach kernel would be forced to implement coherent network shared memory. While this is technically possible (see [31] for example), there are many difficult problems associated with network shared memory. Consider, for example, that the two hosts wishing to share memory may be of a different hardware type. It may be necessary for some data type conversion to take place in addition to maintenance of cache consistency. Clearly, the kernel should avoid these pitfalls.

To prevent this problem from occurring, the underlying primitives have been designed to restrict the way sharing is specified, making it impossible for any sequence of basic operations to result in network shared memory. In particular, sharing is restricted to an inheritance mechanism. Shared memory can only be created as the result of a *task_create* operation, with memory shared between the parent and child tasks. Since the child is created on the same host as the parent, it is guaranteed that all shared memory resides within a host.

3.1.3 Modular Paging

Past virtual memory designs have tied paging operations very closely to the paging system as a whole. In particular, past paging systems have been very inflexible in implementing different paging mechanisms (*pagers*).

A common approach to paging is to reserve some amount of space on a disk, or multiple disks, which will be used for paging/swapping purposes. A single pager, executing as a central component of the system, then reads and writes from the reserved area in a paging specific way. For a system to span a large range of hardware architectures and software systems, both present and future, it is not appropriate to "hardwire" such paging mechanisms into the system and force them to be used in all environments. Instead, Mach provides a generic mechanism that allows several different pagers to be implemented. In a sense, this is not a new idea as it is similar to how device drivers are integrated into an operating system. Nevertheless, it is a key idea that has been applied to the virtual memory system.

Some user-level applications need the ability to explicitly control how virtual memory is managed. Consider Camelot[45], a transaction processing system. Camelot needs the ability to manage *recoverable* virtual memory, memory whose contents can be determined even after node failure. Traditional operating systems require that a transaction processing application use a special kernel modified to provide its new virtual memory paradigm, or that the transaction processing application reimplement its own buffer pool management on top of physical or virtual memory.

Consider also that various types of network shared memory, with varying amounts of consistency, can be implemented by carefully managing the access each host has to a shared page. It would be unreasonable to modify the kernel whenever a new paradigm was to be implemented. For these reasons, it should be possible to export the implementation of basic virtual memory functions into user level tasks. Such tasks, referred to as *external pagers*, are not the topic of this thesis. Nevertheless, the existence of external pagers affects the internal design of the virtual memory system. More information on external pagers is available in [53].

3.1.4 Simplicity

One of the key goals of the Mach system as a whole is simplicity. This is no less true in the virtual memory system. As a result, all of the exported virtual memory primitives are designed to be very simple to understand and use.

Most users will not find it necessary to deal with the virtual memory system at its primitive level. Instead, they will use higher level abstractions for memory management. For example, *malloc* and *free* provide a simple paradigm for memory allocation in a UNIX/C style programming environment. Simplicity at this level is the responsibility of the designer of the higher level abstractions.

Traditionally, only a handful of programmers have been responsible for providing the layer above the kernel primitives. This has largely been true because the primitives below that layer have provided very little functionality, especially in the realm of virtual memory management. Thus there have been few higher level abstractions exported to application designers.

However, with Mach virtual memory management, the way virtual memory can be used expands. This results in the following important developments:

- It is likely that more high level abstractions having a basis in the use of virtual memory will be used in the future.
- Application designers will look directly at the primitives provided by the kernel for virtual memory management since they can be used advantageously without a layer of abstraction.

The net result is that it is not reasonable to assume that only a handful of programmers will ever use the kernel primitives directly. Instead, it is likely that a large number of programmers will find themselves using these primitives. This reinforces the need for simplicity.

Simplicity is achieved by adhering to the following two guidelines:

1. The number of kernel primitives should be small.
2. The arguments to the primitives should be straightforward. All primitives take similar arguments and potential options are kept to a minimum.

The combination of these guidelines results in a set of primitives that are easy to remember, easy to understand and easy to use.

3.2 User Virtual Memory Interface

The Mach Virtual Memory Interface is best thought of as being divided into several functional groups:

- Address space manipulation including the allocation and deallocation of virtual memory.
- Memory protection allowing flexible use of memory protection hardware.
- An inheritance mechanism for creation of address spaces.
- Miscellaneous primitives that formalize access to statistics maintained by the Mach kernel, access other tasks' virtual memory and describe a task's address space.

A detailed description of the Mach Virtual Memory Interface can be found in Appendix A.

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

Table 3.1: Mach Virtual Memory Interface Functional Groups

3.2.1 Task address spaces

In the Mach system, virtual memory resources are bound to tasks. Each task has a single address space. The size of the address space is limited only by the addressing restrictions of the underlying hardware. For example, a task running on an RT/PC can address a full 4 gigabytes of memory while the VAX architecture allows at most 2 gigabytes of user address space.

Mach, like popular memory management hardware, treats an address space as a sequence of pages. Each Mach system has a page size which corresponds to the size of the page that Mach is using on that particular machine. The Mach page size is not necessarily the same as the underlying hardware page size. Instead, it can be any multiple of the hardware page size that is a power of 2.

For simplicity, all parameters to Mach primitives that correspond to addresses or sizes of ranges use byte units (or word units, if that is the basic addressing mode of the hardware). Nevertheless, operations are only meaningful on page boundaries.

Each page in a task's address space is considered allocated or unallocated. An unallocated page may not be addressed by threads in the task while an allocated page may be directly addressed. Virtual memory is allocated using the *vm_allocate* primitive and is deallocated using the *vm_deallocate* primitive. Allocated virtual memory does not necessarily consume system resources. Physical pages are not allocated until virtual addresses are referenced, at which time a page is zero filled. Even pages that have been referenced are not guaranteed to be in primary memory. Instead, they may be stored on a secondary storage device and will be paged in when necessary. Virtual memory that is managed by an external pager is allocated using the *vm_allocate_with_pager* primitive.

3.2.2 Protection

Virtual memory protection occurs at the page level. Each allocated page in an address space has two protection codes associated with it:

- The *current* protection corresponds to the protection code associated with a page for memory references.
- The *maximum* protection 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 (MMUs) do not distinguish between read access and execute access. Therefore, on such MMUs, 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 the current protection or the maximum protection. No current protection can 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. The maximum protection may be used by an application to provide read-only access to shared memory. By setting a task's maximum protection to read-only, it is guaranteed that the task can never gain write permission.

3.2.3 Inheritance

New address spaces are created only as a side effect of task creation. In a pure Mach environment, task creation, hence address space creation, is performed by the *task_create* primitive. Other environments, like UNIX, indirectly create tasks using, for example, the UNIX *fork* primitive.

Task creation in Mach creates a new address space that is either empty or based on an existing address space. When creating a new address space (child) based on an existing address 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 has access to.

3.2.4 Miscellaneous Virtual Memory Operations

Several miscellaneous virtual memory operations complete the interface. While usually not necessary for normal memory manipulations, these primitives round out the interface by formalizing those functions often needed by advanced applications.

Normally, 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 copy a range of virtual memory within an address space efficiently 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 contiguous region of virtual memory for which each virtual page exhibits the same properties. The description includes 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.

3.3 Other Virtual Memory Interfaces

The Mach Virtual Memory Interface is only one of many possible virtual memory interfaces that could be made available to the user. The interfaces provided by other operating systems, for example, must be preserved for emulation of those systems under Mach. For the most part, other interfaces can be easily implemented using the Mach interface as a base.

Emulation of UNIX memory management primitives illustrates how the Mach primitives can be used to emulate other interfaces. The UNIX model of a process's address space is very simple. Each process has exactly three segments: a *text* segment, a *data* segment and a *stack* segment. The text segment contains executable code for the process. The data segment contains both initialized and uninitialized data. The stack segment contains the stack for the process. The size of the text segment is fixed at the time of a UNIX *exec* call. However, both the data and stack segments may grow as the process executes. The primitives that operate on a UNIX address space are simple:

stack growth: stack growth is performed automatically by the UNIX kernel.

exec: sets up a new address space based on an executable file.

fork: creates a new address space that is a copy of the caller's address space.

brk: extends the size of the data segment.

For each possible UNIX process address space manipulation there is a natural Mach emulation:

- Stack growth can be implemented in one of two ways:
  1. Since the maximum size of a stack for a UNIX process is known in advance, that entire region can be allocated. No resources will be allocated until the memory is actually used because of the lazy evaluation mechanism employed by Mach.
  2. The stack can be allocated incrementally, a small number of pages at a time, when faults in the stack segment are detected. This form of emulation has no advantage over the other except that it is similar to the way UNIX implementations perform stack growth.
- The *exec* primitive is implemented by mapping the executable file into the process address space.
- The address space side effects required of *fork* are implemented directly by the *task_create* operation.
- The brk primitive is implemented using *vm_allocate* to allocate additional address space to conform to the end of the "break" region.

Each of these functions is implemented using virtual memory primitives that are available to the user. Therefore, from a virtual memory perspective, this type of emulation can be provided at either kernel or user level.

Clearly, the UNIX model of process memory is simple, hence the simple emulation. However, even more advanced virtual memory primitives can be emulated easily. For example, the *mmap*-based interface proposed for 4.2 BSD[28] can be easily implemented using file mapping, *vm_protect* and *vm_inherit*.

Chapter 4

Implementation

In this chapter, the implementation of the Mach virtual memory system is described in detail. The underlying goals of the implementation are first discussed. Next, the algorithms and data structures used in the implementation are described. Finally, the strategy used in the implementation is reviewed.

4.1 Implementation Features

The implementation of the virtual memory system satisfies several goals. Most importantly, the implementation pays much attention to portability, as the creation of a machine-independent virtual memory system is the paramount goal of this thesis. A second important goal is support for multiprocessor systems; hence, a parallel implementation. Other goals, although less important than portability and parallelism, are operating system independence, paging store independence and external pager support.

4.1.1 Machine Independence

While software designers are increasingly able to cope with variations in instruction set architectures, operating system portability often suffers in the area of virtual memory management. Most of the difficulty in dealing with virtual memory architectures in portable operating systems like UNIX arises because of the proliferation of hardware memory management units. Advances in hardware technology have increased the ease with which a hardware vendor can build a proprietary memory management unit. Often, the vendor believes that his design is “better” than any other design, leading to a perceived competitive edge.

Most MMUs are logically equivalent, with each MMU offering little advantage over the others. Although there are many different paged MMUs in use, the basic functionality afforded by each is the same. An individual MMU can be characterized by the way it exports that functionality. For the most part, a paged MMU supports:

- Separate address spaces, described by some data structure such as a table.
- Page level access to:
  - A pointer to a physical page frame.
  - A *valid* bit indicating whether or not the physical page frame should be used when translating the corresponding virtual address.
  - Several *protection* bits enabling various types of access to a physical page. A typical MMU distinguishes between read and write access, and further distinguishes between access by the kernel or the user.

Most memory management units also provide other functionality such as:

- A *modify* bit indicating whether or not a page has been modified.
- A *reference* bit indicating whether or not a page has been referenced.

Due to the diversity of memory management units used in commercially available systems, vendors of those systems have traditionally chosen one of two possible scenarios for their operating system support for virtual memory:

1. Provide a set of primitives that directly corresponds to those features of the MMU. This option allows applications to use all MMU features, but in a non-portable way.
2. Restrict, or omit entirely, the set of primitives to allow portable software to be created. Only those primitives that are trivially implementable on any hardware base are provided.

The first option is indicative of an operating system that is tied closely to a specific hardware base, such as VMS[18]. Although this option allows all of the hardware's features to be used, the use of such features may impede future portability. The second option is indicative of an operating system with portability as a major goal, such as UNIX. While this option aids the creation of portable software, it does so in an unacceptable way: the functionality provided to the user is unnecessarily restricted. Further, implementations for new memory management architectures are often based on versions already available in 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 the VAX memory mapping architecture internally – in effect treating it as a machine-independent memory management specification.

The implementation of Mach virtual memory must confront the problem of providing very flexible virtual memory support on a wide variety of architectures. To solve this problem the implementation is split into a machine-independent part and a machine-dependent part. The split is based on the assumption that there exists a minimally functional paged MMU. There are no assumptions made about any type of data structure that is directly manipulated by an MMU, e.g., a page table.

The machine-independent part of the implementation is responsible for managing all high level, machine-independent data structures. These data structures represent all operating system knowledge of the state of virtual memory systemwide. All mapping information, for any page in the system, can be reconstructed from the machine-independent data structures at fault time.

The machine-dependent part of the implementation is responsible for the management of physical address maps (called *pmaps*). For example, on the VAX a pmap corresponds roughly to a VAX page table. On the RT/PC a pmap is a set of allocated segment registers. The machine-dependent part implements page level operations on pmaps and ensures that the appropriate hardware map is operational whenever the state of the machine needs to change from kernel to user or user to kernel state. All machine-dependent mapping is performed in a single system module, referred to as the *pmap module*.

One of the more unusual characteristics of the machine-independent to machine-dependent interface is that the pmap module need not keep track of all currently valid mappings. Virtual-to-physical mappings may be discarded at almost any time to improve either space or time efficiency. New mappings need not always be made immediately but can often be lazily evaluated (see section 4.1.7). In order to cope with hardware architectures that make invalidations of virtual-to-physical mappings expensive, the pmap module 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 exception to this rule is that some kernel mappings must be maintained by the pmap module. These mappings, corresponding to the static layout of the kernel, must be kept complete and accurate.

4.1.2 Address Space Management

Even though modern memory management hardware typically provides a 32-bit address space, few applications are able to make use of the entire address space due to operating system restrictions. There are two dominating problems involved with using an entire 32-bit address space:

- Virtual memory systems often have problems scaling up to a 32-bit address space. In particular, a system that requires swap space allocation for all virtual memory, regardless of whether or not a page is ever paged out, can run out of swap space before all of the physical memory in a machine is used. This is a common occurrence on most current UNIX configurations as main memory sizes increase faster than the amount of disk space one is willing to devote to paging.

- Clearly, one can not expect to use every byte of a full 32-bit address space (4.3 gigabytes) without that much memory, either primary or secondary. However, there are applications that benefit from a large address space by using it sparsely. For example, extensive use of mapped files can easily fragment an address space, leaving "holes" of many megabytes when a file is destroyed. As another example, an implementation of Common Lisp uses high order address bits as data type tags, making the implementation more efficient[34].

To a large extent, these problems exist in current systems because they were never considered real problems. The Mach virtual memory implementation addresses these issues by carefully choosing data structures and algorithms that can support large, sparse address spaces efficiently.

4.1.3 Multiprocessor Support

The need to support multiprocessors in the virtual memory system raises some important issues in the implementation:

- The implementation must support shared memory. Although the Mach kernel interface restricts shared memory to an inheritance mechanism, it is possible that other interfaces, provided by other systems, will export primitives that allow more unrestricted memory sharing. Therefore, the implementation can support unrestricted shared memory within a Mach host even though the Mach kernel exports no primitives to make such sharing possible.
- Since Mach is designed to run on multiprocessors providing support for parallel applications, it is a parallel application itself. All algorithms are designed to run in parallel. In turn, all data structures are designed to allow highly parallel access and to avoid potential deadlock conditions.

These issues cannot be ignored, or even glossed over, as has been done in the past. The ability to share memory for read and write between tasks is very important for highly parallel, tightly coupled applications. Without it, this type of application is forced to use expensive synchronization primitives and pay large communication costs.

While it is possible for a non-parallel implementation to provide multiprocessor functionality (such as shared memory), a parallel implementation is necessary in order to prevent the virtual memory subsystem from becoming a bottleneck in large parallel systems. However, it is certainly the case that a parallel implementation has many extra costs associated with it:

- Data structures must be carefully designed to allow parallel access and avoid deadlock.
- Algorithms must run in parallel, properly locking access to all shared data structures.
- Locking requires a non-zero amount of overhead. This overhead may indeed be significant depending on the hardware's ability to perform synchronization operations.

The issues related to parallel data structures and algorithms are in fact non-trivial. In a non-parallel system access to data structures can occur in very unstructured ways. A parallel system, however, must be very careful with its data structure layout and hence its locking protocols. For example, if data structure *A* points to data structure *B*, then it is very difficult to maintain a backpointer from *B* to *A*. The reason for this is potential deadlock on accessing the data structures. If a processor accesses *A* then *B* (following the pointer) it will lock *A* then try to lock *B*. If another processor is simultaneously trying to access *B* then *A* (following the backpointer), it could deadlock trying to lock *A* after it had locked *B*, but before the other processor gained the lock on *B*. In this example, each processor is holding the lock the other is trying to acquire.

4.1.4 Thread Support/Mach Kernel Interface

At first glance, it is not apparent that the Mach thread mechanism impacts the virtual memory implementation. In fact, the implementation has no direct knowledge of threads *or* tasks. Instead, the implementation deals only with address maps and their related pmaps. However, the fact that multiple threads can execute in parallel on a multiprocessor is important to the implementation. An obvious example of this occurs in fault handling. Since multiple threads may be executing, multiple processors may have the same address map in use. Therefore, multiple processors may simultaneously fault on the same address in the same address space. This requires proper locking that would not be necessary if it were guaranteed that only one processor could fault within an address space at any given time.

The Mach Kernel Interface also presents some problems for the implementation. In particular, the Mach Kernel Interface makes it possible for a task to affect the virtual memory of another task (if it has that task's port capability). This means that user-level threads can invoke operations that operate in parallel on a single task. Once again, this requires proper locking.

4.1.5 Operating System Independence

For the most part, major operating system subsystems are closely tied to the operating system as a whole. Although they may be implemented in a modular way, they still depend on many of the interfaces provided by the rest of the system. This is not the case with the Mach virtual memory implementation.

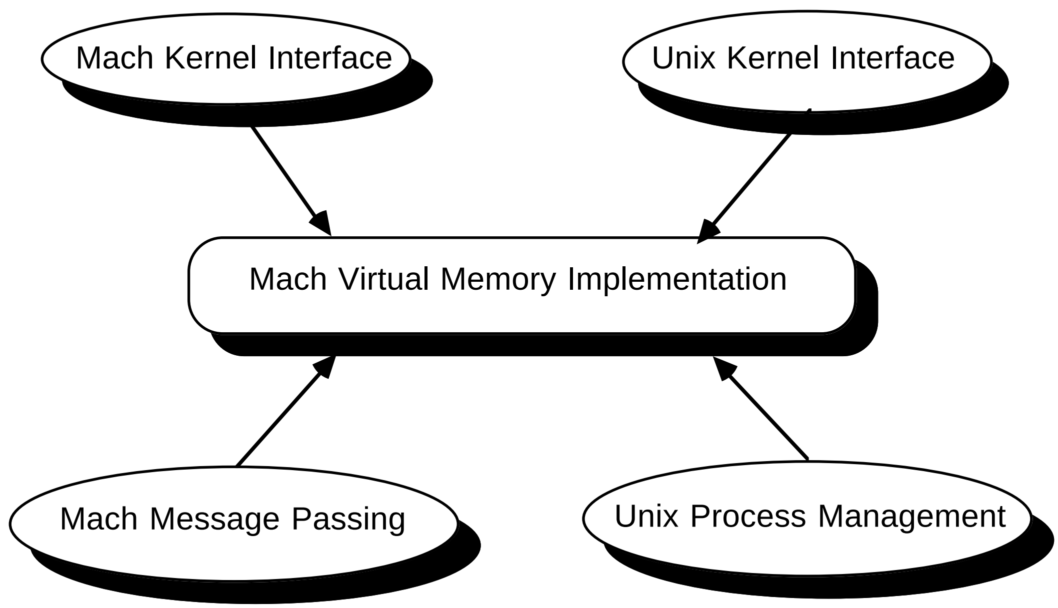

Figure 4.1: Mach/UNIX uses of virtual memory implementation

Mach virtual memory is implemented in a way that is almost completely decoupled from the rest of the system. It makes few assumptions about other kernel functions and is easily adaptable to different systems and/or different interfaces.

The implementation, as a result, has very clean interfaces to the rest of the Mach system. Those interfaces are used in many ways (see figure 4.1):

- The Mach Kernel Interface to virtual memory (e.g., vm_allocate).
- Mach message passing, sending large amounts of data efficiently using copy-on-write.
- The UNIX kernel interface to virtual memory (e.g., brk).
- UNIX process management for virtual memory (e.g., normal paging).

Operating system independence became a goal, in part, because the initial implementation was done within UNIX. However, it was apparent that the other advantages of operating system independence, such as increased modularity and the ability to use the implementation in different systems, clearly outweighed the minimal extra programming effort required.

4.1.6 Paging Store Independence/External Pager Support

The physical memory in a Mach system is used as a cache of secondary storage. This cache is managed by the virtual memory subsystem. However, management of data in secondary storage is not necessarily performed by the Mach kernel proper. In fact, since Mach is an extensible system, designed to run in a wide variety of environments, it would be inappropriate for the kernel to make decisions about management of secondary storage.

The typical approach for managing virtual memory in secondary storage is to dedicate some amount of secondary storage for paging purposes. A common way to do this is to preallocate disk blocks to a special file used for paging and swapping. This is done by VMS for example. A similar approach is used in 4.3 BSD, the primary difference being that entire file system partitions are reserved for paging.

Using this type of paging, it becomes very difficult to integrate virtual memory with other types of data management (e.g., mapped files). The main problem is that the paging system only understands how to page to special areas of a disk, or within a special file, often using specialized routines for those accesses. In effect, the paging routines circumvent the normal file system processing that occurs, making such integration difficult.

The Mach approach assumes no knowledge of secondary storage sources for paging purposes. Instead, the Mach virtual memory implementation defines a simple *pager interface* to which any client may conform. This interface exists in two forms: the internal pager interface and the external pager interface.

The internal pager interface is a simple interface specifying primitives that correspond to operations such as pagein and pageout. This interface can be accessed only within the kernel. A table of interface routines is used to indirectly call appropriate paging routines. This interface is appropriate for use by kernel clients wishing to make data available via virtual memory access. For example, files in the UNIX file system are made available for virtual memory mapping by an *inode pager*. The inode pager transforms pagein and pageout requests to file system accesses.

The external pager interface is a more complex interface that not only allows operations such as pagein and pageout, but also allows for direct cache management operations. This interface is accessed using a message interface, allowing paging operations to be performed in either kernel or user state by tasks anywhere in a distributed system of Mach hosts. Although the external pager interface is not the subject of this thesis, its existence does impact the virtual memory implementation. In particular, the existence of the external pager mechanism dictates that the internals be structured so as to allow proper cache management as specified by the external pager (a task that implements the external pager interface).

4.1.7 Lazy evaluation: map-on-reference and copy-on-write

Lazy evaluation of a function is a technique that postpones the evaluation of a function for as long as possible in the hope that the function will never need to be evaluated. Using lazy evaluation techniques on operating system functions that are expensive to evaluate can vastly improve overall system performance.

The virtual memory implementation makes extensive use of lazy evaluation in order to increase efficiency in both time and space. As will be seen later, functions that are lazy evaluated require that certain data structures be maintained, often for the purposes of "remembering" that an operation has yet to be evaluated.

The key to implementing lazily evaluated functions successfully is to postpone operations for as long as necessary. For many algorithms, it is not always obvious how long an operation can be postponed, or if it can be postponed at all. However, in a virtual memory based system, the memory management hardware automatically notifies operating system software whenever a virtual page has an invalid virtual mapping. This feature, present in all MMU equipped machines, provides the system with *exact* knowledge as to how long mapping operations can be postponed.

Two types of faults generated by an MMU are especially important:

- *invalid page* - a virtual page has no physical translation.
- *protection violation* - a virtual page has a valid physical translation, however the access desired is not allowed.

Invalid page faults are used by many modern operating systems to perform operations such as zero fill on demand as well as more general demand paging. For example, a typical way page faults are used to implement demand paging of executables is to use page table entries to store disk addresses. A system using this technique determines the disk address of each page of the executable, then for each page, places that address in the corresponding page table entry. In essence, the placeholder for a physical page number in the entry is replaced with the disk address. To work properly, the system turns off the valid bit, indicating to the hardware that the entry is not a proper virtual translation. When the hardware accesses that address, a fault occurs and the system then performs the necessary disk I/O operations.

Mach takes the potential for lazy evaluation afforded by invalid page faults a step further: absolutely no physical mappings are made until they are used. In this way, Mach postpones creation of page tables and lookup of disk addresses until needed. The same is true for zero fill memory. Mach makes no mappings until a fault is detected whereas a traditional operating system sets a flag in a page table entry indicating zero fill on demand. Since page tables are not allocated until needed, it is possible to allocate large amounts of memory at nearly zero cost until it is used.

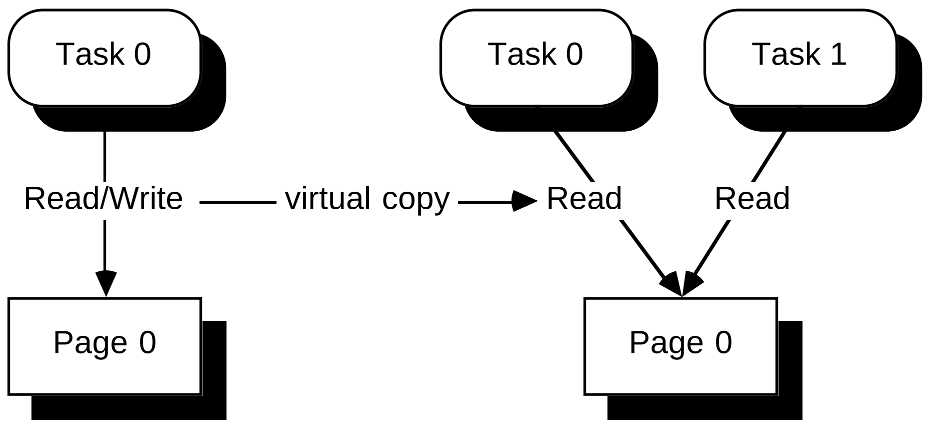

Figure 4.2: Sharing a page copy-on-write

As an example, Mach implements stacks for UNIX processes by allocating enough virtual memory to hold the maximum size allowed for each process. It is then not necessary to automatically expand the stack since it is already fully expanded. Since Mach lazily evaluates the allocation function, it is no more expensive to allocate the whole stack than it is to allocate it a page at a time — regardless of how much of the stack is ever used.

Hardware-detected protection violations offer an opportunity for an additional important optimization using lazy evaluation: *copy-on-write*. Copy-on-write is the postponement of a copy operation until the data to be copied is written. It is implemented by allowing multiple read-only virtual mappings to the same “virtual copied” physical page (see Figure 4.2). When a task tries to write into the page a protection fault is generated. At this time, the page is physically copied into a new page and a new virtual mapping is set up for the newly created page.

The copy-on-write optimization is important in many respects:

- Memory that should be copied when written, but shared for read (such as a shared library) is very efficiently implemented without special primitives or restrictions.
- Pages that are copied, but are never written after the copy, are shared. This reduces the overall memory load of the system.
- No CPU time is spent copying pages that are never written — they are simply shared. The copy operation only occurs when the data is written.
- Since mapping operations are also lazily evaluated, virtually copied pages do not even incur mapping costs if they are never accessed.

These conditions occur frequently in normal system operation. For example, the UNIX fork operation is efficiently implemented using copy-on-write. Those pages that are never modified are never copied.

4.2 Data Structures and Algorithms

Four basic memory management data structures are used in Mach (see figure 4.3):

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 roughly correspond 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 the hardware representation of an address space (page tables, for example).

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 mapping data structures required by the hardware. All important virtual memory information is maintained by machine-independent code. In general, the machine-dependent part of Mach maintains only those mappings crucial to system execution (e.g., the kernel map and the mappings for frequently referenced task addresses) and can 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.

4.2.1 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 be linked simultaneously into several lists:

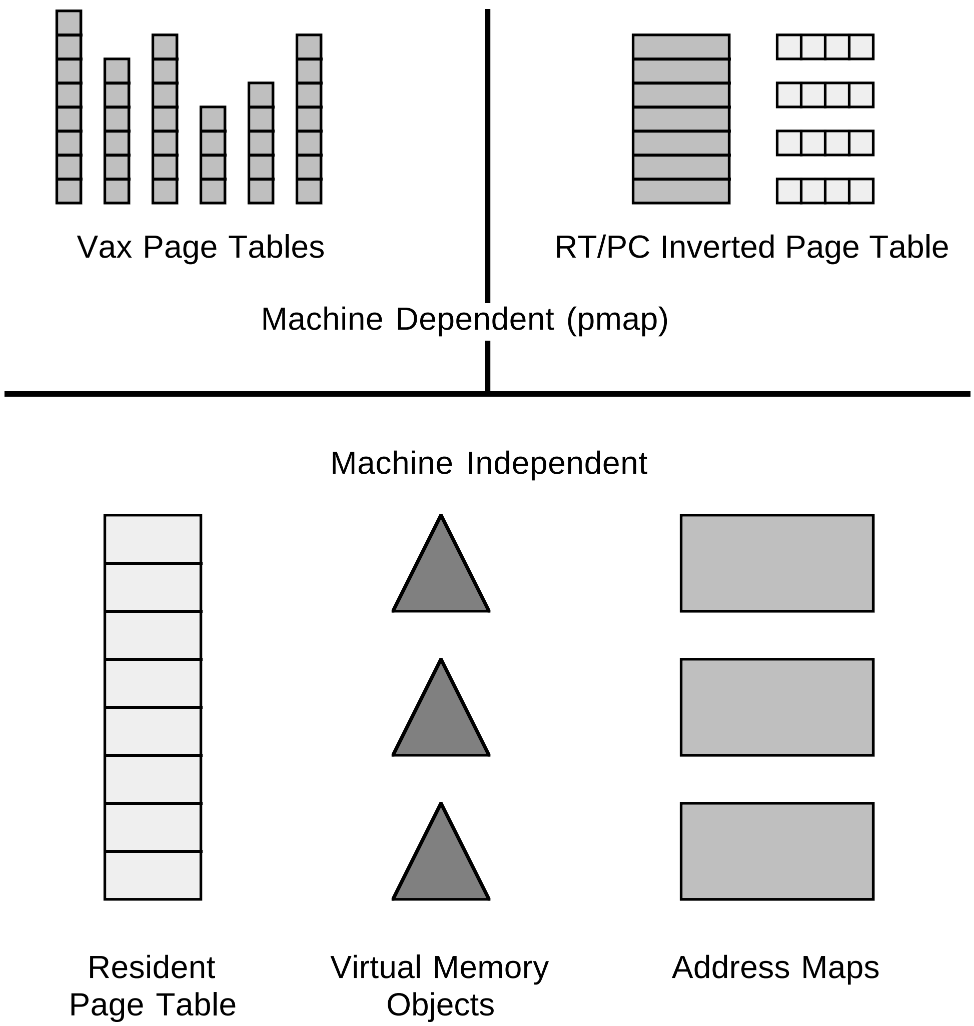

Figure 4.3: Data structure overview — machine-independent and machine-dependent data structures

- 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. 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.

4.2.2 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 (see Figure 4.4).

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. Operations that operate on a subset of a region corresponding to an entry generally 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. Figure 4.5 summarizes the address map entry data structure.

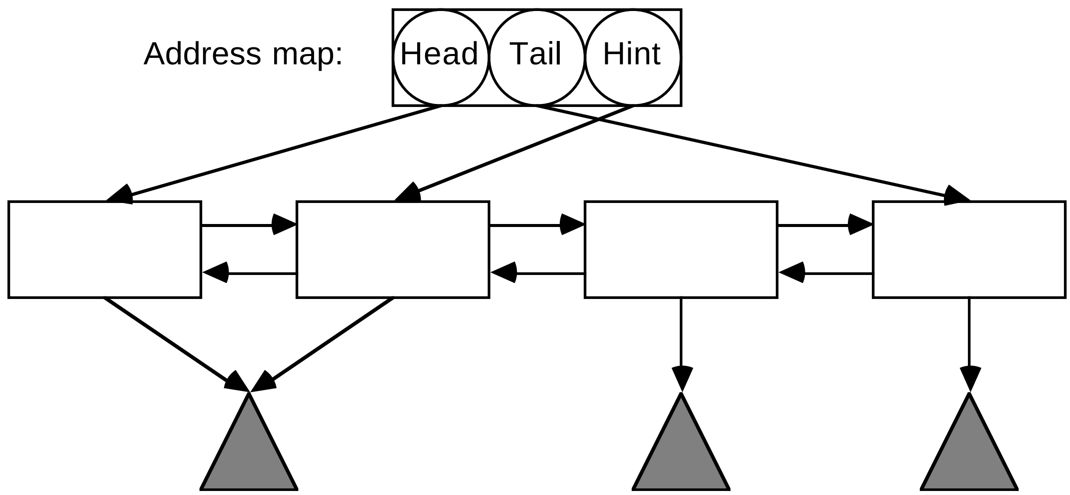

Figure 4.4: A simple address map

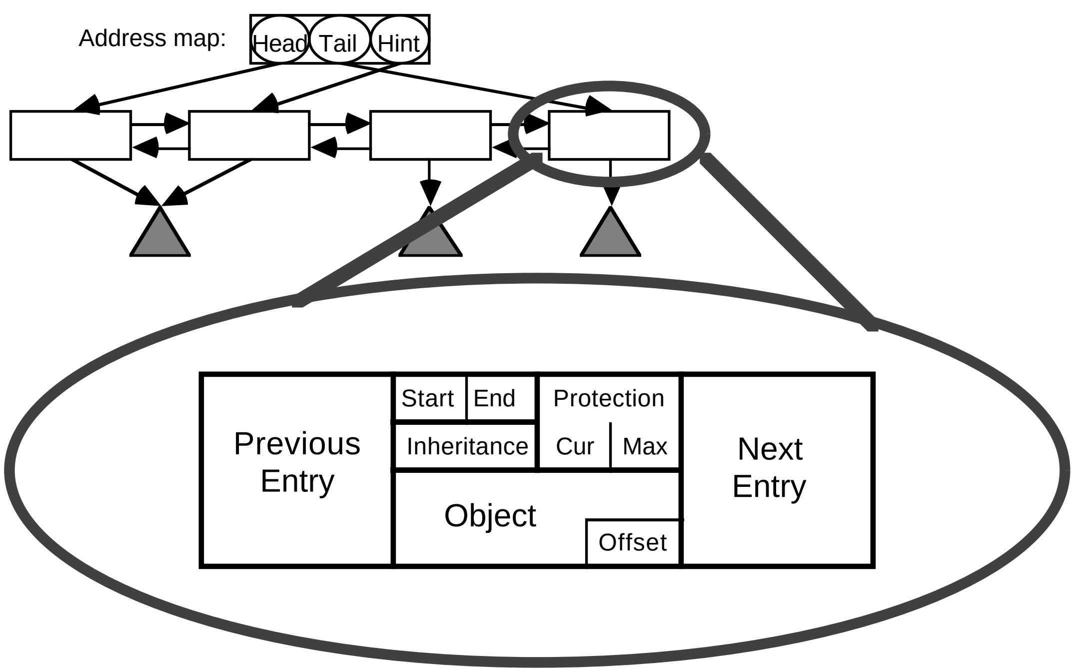

Figure 4.5: An address map entry.

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 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 [21]).

4.2.3 Memory Objects

A Mach address map need not keep track of backing storage. Instead, backing storage is implemented by Mach *memory objects*. Logically, a virtual 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 virtual memory object is mapped into an 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 in which the data in the object is 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.

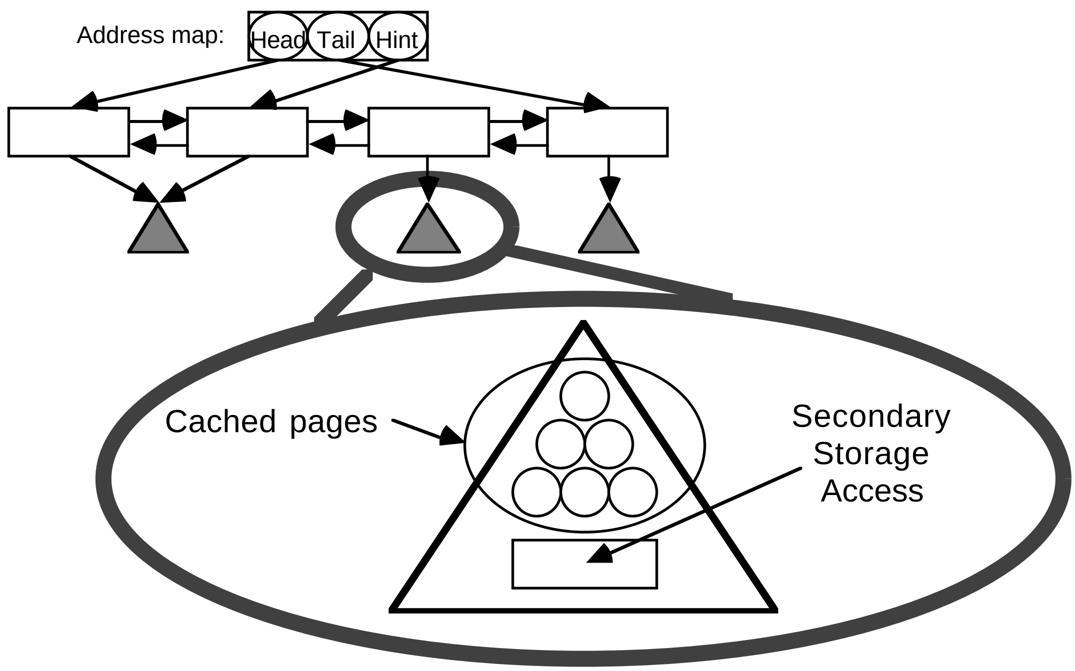

Figure 4.6: A virtual memory object

Conceptually, a memory object represents secondary storage or, more specifically, the data that is implemented by the pager for the object (e.g., a file for a file system pager). The act of mapping parts of an object into a virtual address map makes that data available for direct access through that address space.

The kernel acts as a cache manager for objects, using physical memory as a cache of the object's data. References to data in an object that is not in the physical memory cache are translated into paging requests on the pager. 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 access the pager routines (see figure 4.6).

As was pointed out in Section 4.2.1, 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.

4.2.4 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. This new page is the one 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 that 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-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, creating what is termed a *shadow chain* (see Figure 4.7). When 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.

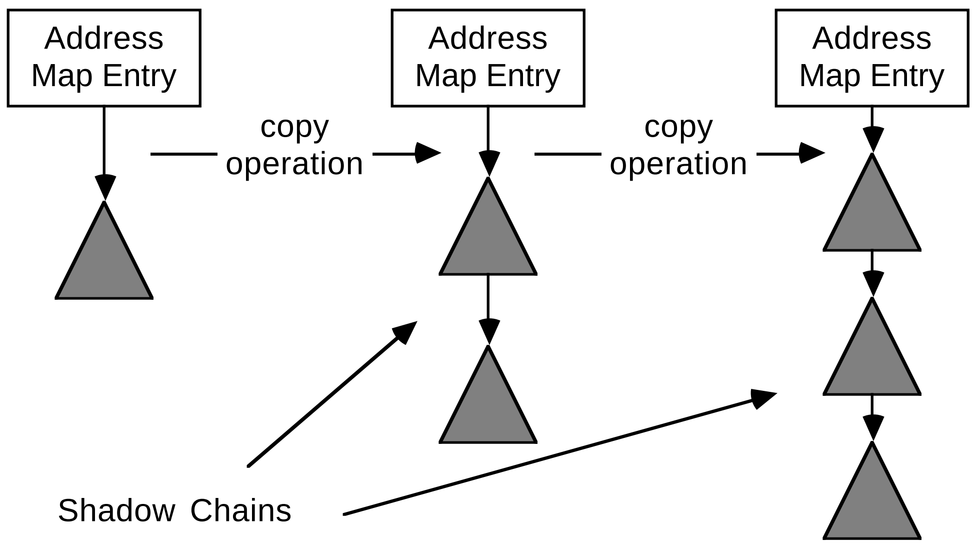

Figure 4.7: Shadow chains as a result of virtual copies.

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.

Sharing maps are created as part of the *task_create* operation whenever an inheritance value of SHARED is detected (see Figure 4.8). 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 the full range of task-to-task address space sharing to be permitted (e.g., 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 only created if absolutely necessary. This simplifies map operations and obviates the need for garbage collection of sharing maps.

4.2.5 Copy Objects

While pagers are usually only concerned with simple pagein/pageout operations, the complete pager specification allows full cache management, including page flushing and access restrictions, to be performed by an external pager. This in itself presents few problems to the virtual memory system implementation — it simply means that the required hooks into the system be available.

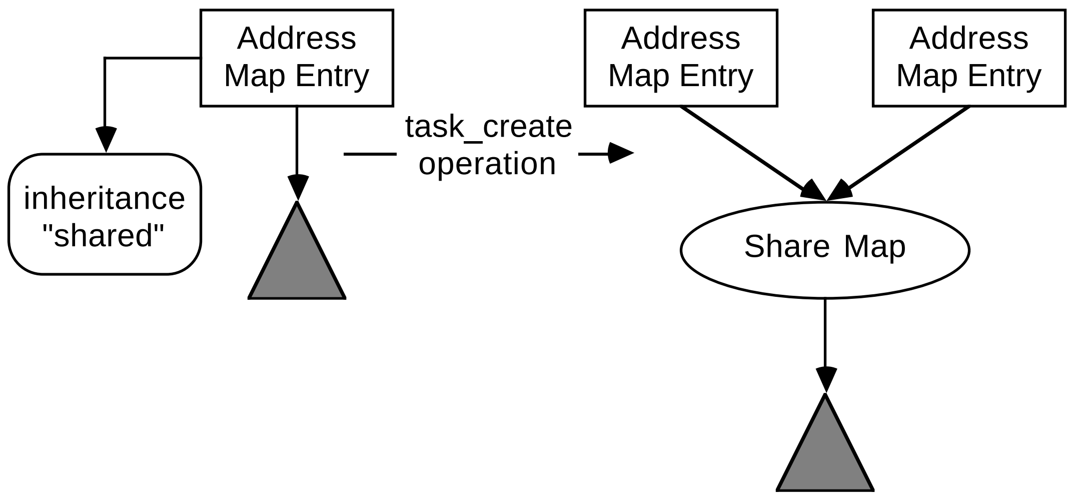

Figure 4.8: Share map creation on task_create operation.

However, the implementation of copy-on-write using shadow objects presents a complication.

An external pager must be able to manage its paging spaces regardless of whether or not the copy-on-write optimization is being used. That is, the copy-on-write optimization is a transparent optimization - transparent not only to users, but also to external pagers. The complication arises when a shadow object is created, and the object being shadowed is managed by an external pager. In this case, shadowed pages are placed in the new object, which the external pager has no knowledge of (and can never have knowledge of since copy-on-write is a transparent operation). Correct external pager operation may require the ability to manage these pages — impossible if they are now in a different object.

As a result, it is not appropriate to use shadow objects to shadow directly those objects managed by external pagers. However, it is still necessary that there be two separate objects to implement copy-on-write: the one containing the original, uncopied pages (for all mappings that have not yet copied the page) and one containing the new, copied pages (for the mappings that have been accessed for write). Rather than use a shadow object, another type of object is used: the *copy object*. The copy object is a repository for the *original* pages of the original, copied object, whereas the shadow object is a repository for the *copied* pages of the original, copied object. Copy objects are used when copying an object managed by an external pager, whereas shadow objects are used when copying all other objects (copy, shadow or objects with no pagers).

When a copy-on-write fault occurs on an object with a copy object, the original page is moved to the copy object. A new page is allocated and the data from the original is copied into it. The page is then placed in the original object. With shadow objects, the original object holds the pages at the time of the copy. However, with copy objects the original object holds the current

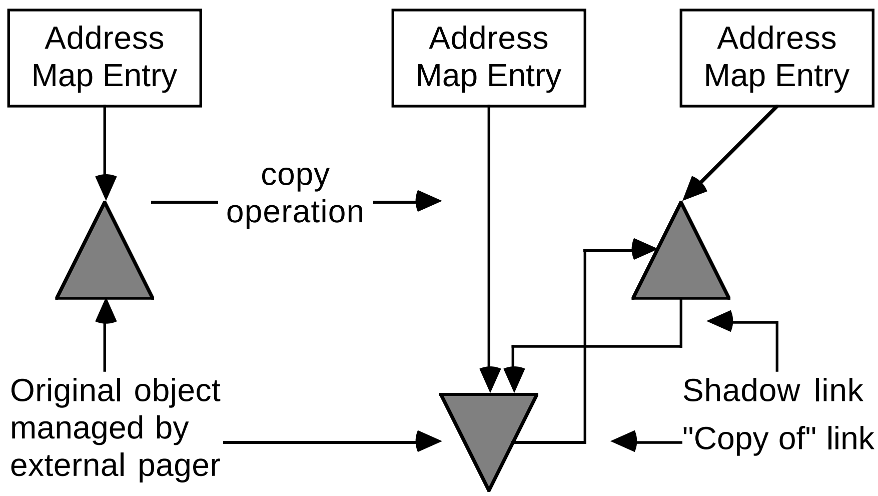

Figure 4.9: External pager managed objects and copy objects

pages — the original pages are placed in the copy object. The copy object, like the shadow object, must still consult the original for pages it does not have. Therefore, the copy object has a pointer to the original (the shadow link). Since the original can push pages into the copy object, the original must have a pointer to the copy (the copy link). Figure 4.9 depicts the relationship between objects managed by external pagers and the resulting copy objects.

In essence, copy objects and shadow objects are the same — in fact, all objects use the same data structure definition. The only difference lies in which objects hold original or copied pages. All other properties of objects are the same. Not surprisingly, as is the case with shadow objects, repeated virtual copy operations can cause chains of copy objects to be created. In fact, since copy objects provide all the functionality that shadow objects provide, shadow objects could be completely eliminated. However, the need to collapse object trees, described in the next section, makes it advantageous to use shadow objects for most copy-on-write operations.

4.2.6 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 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 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 CPUs has resulted in complex object locking rules.

4.2.7 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. By retaining the physical page mappings for such objects subsequent reuse can be made 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 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 is 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.

4.2.8 The PMAP Interface

The *pmap* module is the sole machine-dependent module in the virtual memory implementation. The interface to this module has been carefully designed to support a wide variety of hardware MMUs, and could even be considered for direct implementation in hardware or firmware. The interface assumes only a simple, paged MMU architecture, and is appropriate for designs ranging from complex page tables to TLB-only designs.

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.

Since the machine-independent page size is not necessarily the same as the hardware page size, the pmap module must be prepared to deal with multiple page modifications, even if the machine-independent code is assuming a single page. For simplicity, all virtual addresses, offsets and sizes passed to the pmap module are passed in byte numbers, not page numbers. Further, the machine-independent code guarantees proper machine-independent page alignment of parameters where applicable.

The interface to the pmap module is split into several functional groups:

- Pmap handle maintenance such as creation and destruction of pmaps.
- Pmap range operations - those primitives that change mappings for a range within a pmap.
- Pmap global operations - those primitives that change the state of (potentially) many pmaps simultaneously.
- Operations that allow the kernel to query the status of various mappings.
- Physical memory manipulation routines.
- Use information, in which the kernel provides information about which CPUs are using which pmaps.
- Miscellaneous optional operations which may be used by an implementation for added efficiency.

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 to specify which pmap to affect. *Pmap_reference* and *pmap_destroy* increment and decrement reference counts on pmaps. When the reference count on 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 go higher than one 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.

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 4.1: Mach Pmap Interface Functional Groups

Address space specific calls are used by the machine-independent part of the system to notify the pmap module 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 (machine-independent) page.

A few kernel functions cause multiple address spaces to be affected simultaneously. 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. In this case, 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 implementation to inform the machine-dependent implementation of changes in mappings, there is also a primitive that enables 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 purpose 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 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 especially important for implementations that must perform some type of TLB consistency.

Finally, the pmap interface is completed with miscellaneous primitives. These routines are optional; 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 last 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 presented to the pmap module using the other primitives (using *pmap_enter* for example). These miscellaneous primitives represent redundant information that may be used for performance optimizations.

It is interesting to note how the interface to the pmap module was determined. The pmap interface is not an attempt to abstract all of the functionality provided by popular MMU designs. Instead, it is an interface to the page-based functionality needed by the machine-independent algorithms. Each primitive in the interface provides a specific function as required by machine-independent code.

The full pmap interface, including parameter specifications, is presented in Appendix B.

4.2.9 Page Replacement

One of the basic functions that must be performed by the kernel is physical memory cache management. For the most part, this consists of caching pages the first time they are faulted. However, given that the physical memory of a machine is of a fixed size, it is only a matter of time before the physical memory cache becomes filled, necessitating *page replacement*.

Many different page replacement algorithms have been used by virtual memory systems. Some examples are as follows:

- *least recently used (LRU)* — The LRU algorithm selects the least recently used page as the page to replace. While this is often a good indicator of which page is best to replace, it can be difficult if not impossible to compute.
- *first in, first out (FIFO)* — The FIFO algorithm selects pages to be replaced on a first in, first out basis, regardless of use. While this is not necessarily as good as LRU as an indicator of the best page to replace, it is easily computed using a queue.
- *global clock* — The global clock algorithm sequentially scans all physical memory, turning off reference bits during the scan. When pages need to be replaced, the scan continues, looking for pages without a reference bit turned on. When those pages are found they are replaced.
- *working set* — Working set algorithms operate on a per address space basis. Each address space is usually given a working set size. Page replacement occurs within the working set based on other algorithms such as LRU or FIFO.

Detailed descriptions of various page replacement algorithms can be found in [11], [10], [13] and [12].

In deciding what type of page replacement algorithm would be appropriate for Mach, several issues were considered:

- The algorithm, in keeping with Mach philosophy, should be simple. A complicated algorithm is likely only to waste CPU cycles.
- There is no desire to try to make optimal decisions as to which page is best to replace. Such knowledge is certainly beyond the realm of the kernel.
- External pagers, or even external page replacement tasks, could be used to perform "intelligent" page replacement functions based on domain-based knowledge in special environments.
- Working set based algorithms are not necessarily appropriate for a Mach-like environment where server-based applications have a working set extending beyond a single address space.

Based on these issues, a simple page replacement algorithm was determined. The algorithm is a modified-FIFO. Pages are kept on one of three FIFO queues:

1. the *free* list: The free list is a list of pages that are free for use. Memory allocation dequeues pages off the free list.

2. the *active* list: The active list is a list of all pages that are actively in use in some address space. When pages are allocated, they are removed from the free list and inserted into the end of the active queue.

3. the *inactive* list: The inactive list is a list of pages that are not in use in any address space, but were recently in use. This list can be thought of as containing pages that are not referenced, but still remain in the physical memory cache.

Page replacement is performed by a special kernel thread called the *pageout daemon*. The pageout daemon is run whenever the free page count is low and page replacement is necessary. Page replacement occurs by removing pages from the inactive list, placing them on the free list. The pageout daemon always maintains a small number of pages on the inactive list by moving pages from the active list to the inactive list (then removing mappings to those pages). This algorithm results in a FIFO-like page replacement, except that pages on the inactive list are *reactivated* if they are faulted on. The inactive list serves as a second chance for pages about to be replaced.

Overall, the algorithm is very simple. Computation of the next page to replace is always done in constant time (a dequeue operation). Although FIFO is not a good predictor of the best page to replace, the second chance afforded by the inactive list allows those pages that are actively being used to remain in memory, even though they have circulated through the FIFO queue.

4.2.10 Locking Protocols

All algorithms and data structures used in the virtual memory implementation are designed to be used in a multiprocessor environment and are therefore fully parallel. Parallelism is achieved by following a few, simple locking rules throughout the implementation. There are several major types of locks used in the implementation:

- *map locks* are used to guarantee exclusive access to address map data structures,
- *object locks* are used to guarantee exclusive access to physical memory resources cached within an object,
- *hash table bucket locks* are used to guarantee proper access to the object/resident page table hash table on a per bucket basis, and
- *busy page locks* are used to indicate that some operation is pending on a given physical page.

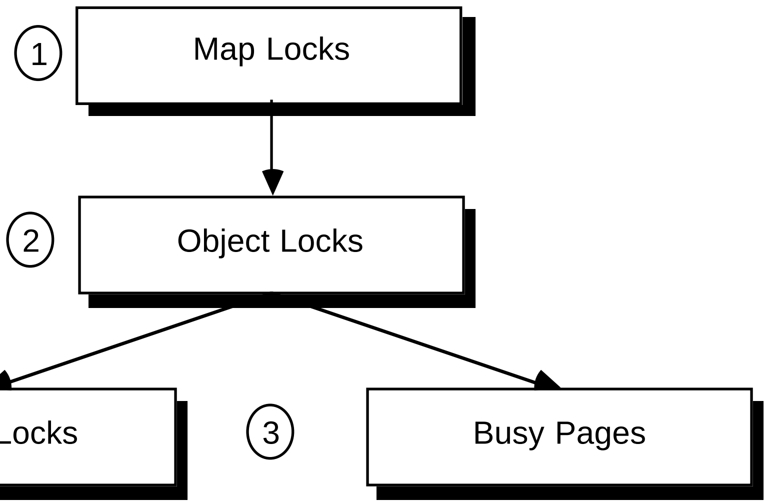

Figure 4.10: Locking protocols in the virtual memory implementation

To prevent deadlock conditions, all algorithms gain locks using the same ordering: map locks, object locks, then bucket locks or busy page locks (see Figure 4.10). Busy page locks, which are themselves protected by object locks, may be held after all other locks are released. Therefore, busy page locks are used as long-term locks on single pages while allowing access to other pages in an object. Locks gained on multiple objects simultaneously can never deadlock since they are always gained by following the object tree created by shadowing operations.

There are two exceptions to these locking rules - both related to object locks. The first pertains to the pageout daemon. In this case, the pageout daemon acquires page-based locks followed by an object lock (for the object corresponding to the same page), which is a violation of the protocols. In addition, the existence of copy objects means that object locking may not necessarily follow a tree. The copy object is represented by a pointer back into the tree, a possible deadlock condition. Both exceptions are handled by first testing if the object lock is already held, and backing off if it is.

4.2.11 The Page Fault Handler

The Mach page fault handler is the hub of the Mach virtual memory system. The kernel fault handler is called 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.

4.2.12 Implementation Code Size

The size of the virtual memory implementation is summarized in figure 4.11. The total size, about 16K, represents the amount of code that would be required for a minimal kernel. The addition of UNIX support, including support for paging to and from UNIX file systems increases these totals. Also, implementation of the pmap module increases this total by an amount that depends on the complexity of the hardware.

These sizes are negligible, especially when compared to highly functional systems such as 4.3 BSD which can easily exceed 300K of code. In fact, all of this code represents a small portion of the total amount of code required to emulate 4.3 BSD on a Vax (see Figure 4.12). This is especially important for potential Mach users depending on a small kernel — and willing to give up UNIX functionality for it.

4.3 Generic OS Support

Although the virtual memory implementation is largely operating system independent, it does use some of the generic kernel facilities provided by the Mach kernel implementation. If the Mach virtual memory implementation were to be used within another operating system, the following features would need to be provided:

- *scheduling primitives* - simple primitives allowing a thread, process or other executable entity to be put to sleep and awakened.

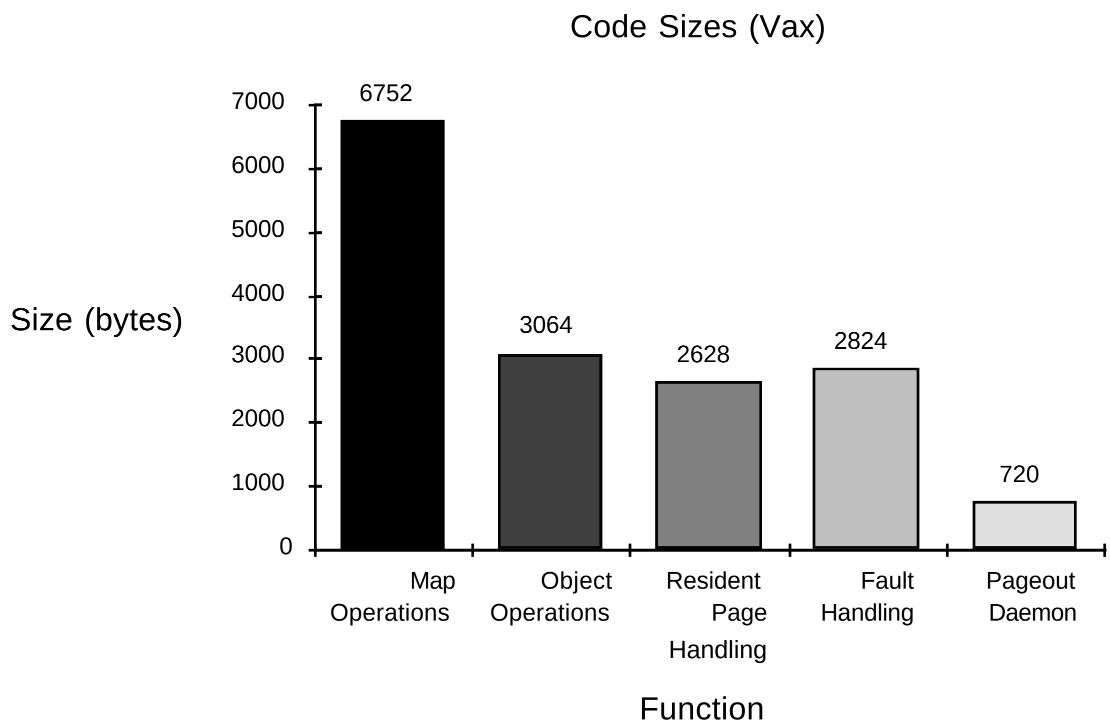

Figure 4.11: The object size of the machine-independent virtual memory implementation broken down into modules (Vax architecture).

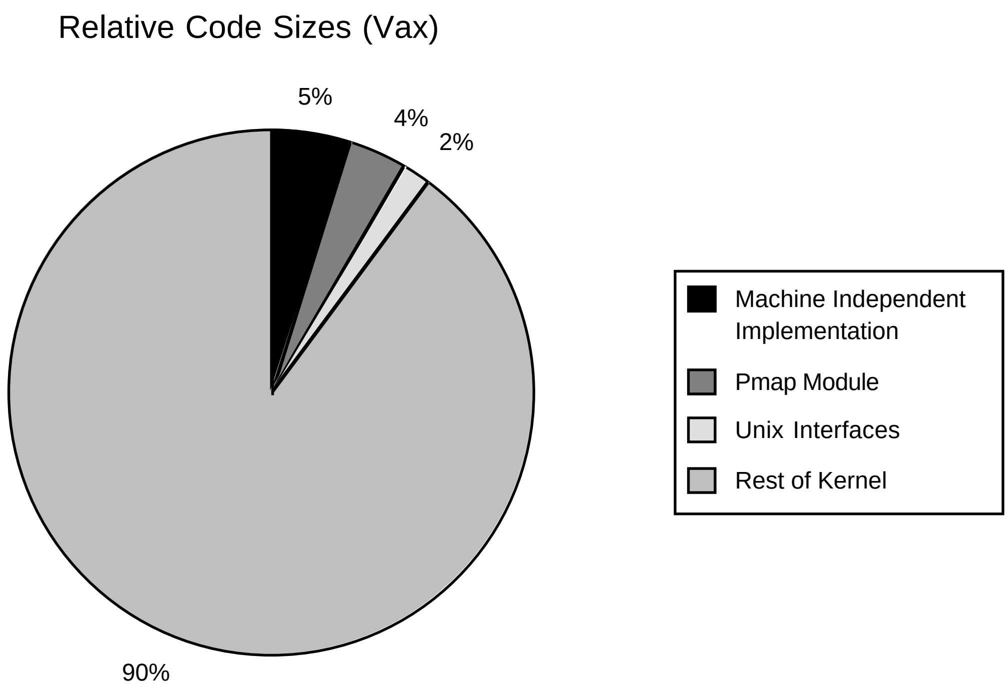

Figure 4.12: Relative code sizes on a Vax.

- *lock primitives* - primitives that allow for both spin and blocking locks. Support for multiple reader locks allows greater parallelism.
- *zone package* - a fast, small memory block allocator.

All of these features are provided by the Mach kernel. Most were originally implemented for purposes of the virtual memory system. However, they have been found to be of general use in other aspects of the system, including Mach task and thread management as well as the inter-task communication implementation.

4.3.1 Scheduling Primitives

The Mach virtual memory subsystem uses the following scheduling primitives:

- *thread_wakeup*: wakeup all threads waiting on the specified event.
- *assert_wait*: indicate that the current thread will soon wait for the specified event.
- *thread_block*: switch the next runnable thread of highest priority, or go idle if there is no such thread.

One additional primitive, *thread_sleep* is simply a shorthand for a combination of *assert_wait* and *thread_block*. In particular, the semantics of *thread_sleep* are summarized by the following C fragment:

```
thread_sleep(event, lock)
        event_t  event;
        lock_t   lock;
{
        assert_wait(event);
        unlock(lock);
        thread_block();
}
```

In general, when a condition arises that requires the executing thread to sleep (wait for a pagein operation, for example), the kernel indicates the desire to sleep using *assert_wait*, releases any locks that cannot be held while sleeping, then calls *thread_block* to reschedule the processor. When the condition that caused the sleep no longer exists, the kernel invokes *thread_wakeup* to continue the execution of threads waiting for that condition.

4.3.2 The Lock Package

The lock package used by the virtual memory implementation represents tradeoffs in both speed and functionality. The system exports two different types of locks: *simple* and *complex*. Simple locks may only be used for mutual exclusion and always spin when waiting for the lock. Complex locks provide a much higher degree of functionality, including:

- *multiple reader, single writer* locks — this allows multiple processors to access simultaneously a lock for read access. Upgrading a read lock to a write lock is also supported.
- *sleepable* locks - a lock set to *sleepable* will invoke a sleep operation if the lock is held by another processor. When the lock is available, a wakeup operation will occur.
- *recursive* locks - a lock may be set to allow a thread to transparently gain a lock in a recursive fashion. In particular, a recursive lock will not cause deadlock if the thread that holds the lock tries to gain the same lock.

By their very nature, complex locks are much more expensive to acquire and release than simple locks. Most data structures use simple locks for performance reasons. However, map locks use complex locks to support parallel fault handling within a map (using read locks), to block map operations automatically (using sleep locks), and to avoid complex locking protocols when it is necessary internally to invoke the fault handler (using recursive locks).

Appendix C contains a detailed presentation of the primitives exported by the lock module.

4.3.3 The Zone Package

One of the needs of the virtual memory implementation is the management of memory for data structures such as address maps, address map entries, objects, and hash table entries. In many cases, these data structures must be allocated or deallocated in time critical pieces of code. Therefore, speed of allocation and deallocation is of paramount importance.

Memory for such data structures is organized into *zones*. A zone is a collection of free data structures, managed as a linked list. If the zone is non-empty, which is the normal case, allocation only requires a list removal operation, and deallocation a list insertion operation. If the zone is empty, a deallocation still only requires a list insertion operation, but an allocation requires more complex processing. In particular, more memory is allocated, then split into small blocks and placed in the zone's list of free elements.

Associated with each zone are:

- a *free list* — a linked list of free elements available for immediate allocation.

- an *element size* — the size of the elements that are maintained in the zone.
- a *current size* — the amount of virtual memory in use by the zone.
- a *maximum size* — the maximum amount of virtual memory that can be used by the zone. This is useful for catching zones that "leak" memory.
- an *allocation size* — the amount of virtual memory to allocate when the zone becomes empty.
- a *name* — a character string used for debugging purposes.

The zone package provides a simple, efficient mechanism for the allocation and deallocation of many data structures used by the kernel. It is used extensively not only by the virtual memory implementation, but also by the rest of the Mach kernel. Appendix D lists the full interface to the zone package.

4.4 Implementation Strategy

The implementation of the Mach virtual memory system was done in several, distinct steps:

1. user mode development,
2. UNIX process handling,
3. internal Mach use, and
4. exported Mach interface.

Initial development of the high level data structure management routines, such as address map and address map entry maintenance was done within user-level programs. Almost all of the machine-independent code was written and embedded in a user-level test program. Features debugged in this way included map manipulation, object manipulation, fault handling and page maintenance.

Once debugged in user-mode, the implementation was inserted into the Mach/UNIX kernel, replacing the 4.3 BSD virtual memory system. At this time, the VAX pmap module was written and the machine-independent code continued to be debugged. The Mach virtual memory system then provided virtual memory support for UNIX process handling. In particular, *fork* and *exec* were recoded to use the Mach mechanisms for address map creation and file mapping (for demand paging purposes). UNIX process stacks were simulated using the ability to allocate sparse address spaces. Finally, the *brk* system call was implemented in terms of Mach internal operations (*vm_allocate*). Shortly after achieving a stable VAX-based system, the system was ported to the RT/PC, resulting in the second pmap module.

Once UNIX process virtual memory management was stable, more advanced use of the Mach virtual memory system began. This took the form of providing support for internal Mach mechanisms. Most importantly, Mach message passing used the virtual memory system to send large amounts of data copy-on-write between processes efficiently. Previously, large amounts of data were sent using wired down kernel memory, resulting in strict limitations on the amount of data that one could send. Using the new functionality, these restrictions were eliminated.

Finally, after making extensive use of the virtual memory system internal to the kernel, the Mach virtual memory interface was exported to the user. This allowed full functionality to be exploited by the user - and for the first time, the use of inherited shared memory.

This implementation path allowed various parts of the implementation to be tested in isolation, aiding debugging tremendously. More importantly, once the implementation was embedded into the Mach/UNIX kernel, many users could start running the kernel. This resulted in the early discovery of bugs that might have remained undetected longer.

However, along with the advantages of this development path was a glaring difficulty: integration with the UNIX kernel. Although UNIX required little in terms of virtual memory management, its integration into the rest of the system presented a problem. In particular, portions of the BSD memory management code were spread throughout the kernel, necessitating many changes in areas that would have otherwise been unrelated to the virtual memory system.

Chapter 5

Applications

Providing advanced virtual memory functionality such as copy-on-write, shared memory and generalized paging in and of itself is of little utility. More importantly, it is necessary that there exist an application base that can make use of the features in a positive way. Such applications exist in both the kernel and user level software. Some of these applications are examined in this chapter.

5.1 File mapping

Whereas virtual memory is usually considered to be volatile, temporary storage, files are usually considered to be non-volatile, long term storage. As such, the physical media used for virtual memory or files are different. On one hand, virtual memory is usually backed by high speed memory chips offering no data retention between crashes, power cycles, etc. On the other hand, files are usually backed by disk storage which is much slower to access than memory chips, but offers long term storage capabilities.

Many systems make strong distinctions between memory access and file access. For example, UNIX provides a read/write/open/close model for file access. This model is adequate for applications that sequentially process information in files. However, applications that process file data in random fashions find themselves spending a considerable amount of time *seek*ing to new file positions and buffering data. A UNIX-like model was appropriate for the hardware architectures existing when UNIX was designed — small memory machines. But with large memory, MMU-equipped machines now widely available, limiting applications to that form of file access is unnecessary.

Systems with advanced virtual memory functions, such as Multics and Accent, allowed files to be *mapped* into virtual memory. File mapping allows an application to view a file as an array of bytes of virtual memory. The application need not perform buffering or seek operations, it simply accesses the virtual memory that corresponds to the file as if it were normal memory.

In both Multics and Accent, file mapping was a key primitive provided by the system directly. In Mach, file mapping is provided easily using the pager interface. A file is mapped by allocating memory, with the file system acting as the pager for the allocated memory.

In Mach, file mapping is used for reading data only. However, other mapped file interfaces have been designed to address a variety of issues. For example, the 4.2 BSD mapped file interface (*mmap*) was designed to address two shortcomings of previous UNIX systems: a lack of shared memory between processes and the need to simplify processing of file data[28]. Early UNIX systems offered no shared memory access but provided a stylized method for accessing sequential file data through *read* and *write* system calls. Applications desiring random access to data would use UNIX's *seek* operation or buffer their data themselves, often incurring unwanted system overhead. A mapped file facility could allow a user to treat file data as normal memory without regard to buffering or concerns about sequential versus random access. It would also provide an obvious mechanism for sharing memory by allowing more than one process to map a file read/write simultaneously.

Despite its obvious convenience, the notion that all memory sharing should be channeled through a mapped file interface presents a number of problems:

- A file is (in principle) a permanent on-disk data structure must be maintained consistent against crashes. The use of disk files to exchange temporary data can put an unnecessary I/O load on the system and impact performance.
- A mapped file facility must take into account the sharing of remote (network) files. In order to handle remote file systems (e.g., SUN NFS), the operating system must be intimately involved in maintaining network data consistency.
- Sharing semantics are limited to those supplied by the kernel. In particular, an application program cannot use domain specific knowledge to allow less than full consistency in sharing file data. This can result in inefficiency in the handling of data sharing across node boundaries.

These problems typically have led to compromises in the actual mapped file semantics provided. Most have either assumed that modifications to read/write mapped files are not guaranteed to be consistent in the face of multiple writers, or they guarantee consistency only for those processes which share files on a single network node.

5.1.1 Implementing file mapping: map_fd

UNIX emulation under Mach has been extended to allow file mapping of UNIX files based on file descriptors. A single system call, *map_fd*, is provided for file mapping:

```
map_fd(fd, offset, addr, find_space, numbytes)
        int             fd;
        vm_offset_t     offset;
        vm_offset_t     *addr;
        boolean_t       find_space;
        vm_size_t       numbytes;
```

Map_fd is called with an open UNIX file descriptor (*fd*) and if successful results in a virtual copy of the file mapped into the address space of the calling UNIX process. *Offset* is the byte offset within the file at which mapping is to begin. The offset may be any byte offset in the file; page alignment is not required. *Addr* is a pointer to the address in the address space of the calling process at which the mapped file should start. This address, unlike the offset, must be page aligned. If *find_space* is TRUE, the kernel will select an unused address range and return it in **addr*. The number of bytes to be mapped is specified by *numbytes*.

The implementation of *map_fd* is a straightforward application of internal Mach primitives for virtually copying regions of memory and external pagers[40,53]. When a request is made for a file to be mapped into a user address space, the kernel creates a temporary internal address space into which the file is mapped. This mapping is accomplished with the *vm_allocate_with_pager* primitive. The kernel specifies that new memory is to be allocated and that the new memory will be backed by the internal kernel *inode pager*. Then the file data is moved to the process address space by a call to *vm_write*. Once this is done, the kernel deallocates the temporary map.

5.1.2 Using file mapping for C standard I/O

The Mach mapped file mechanism has been used to build a new version of the C library buffered I/O package. When a file is *fopen*ed it is mapped in its entirety into the caller's address space. The semantics of the buffered I/O package are not changed. The existing *stdio* buffer has, in effect, been enlarged to the size of the file. Only the data buffer is changed when a write takes place. The file is not guaranteed to change on disk until a *fflush* or *fclose* takes place. As with normal buffered I/O, if two processes have the same file open for reading and writing, there is no guarantee how the reads and writes will intermix. A read may get new information off the disk copy of the file, or it may use information that was already buffered.

In addition to improved performance, the use of file mapping also has the effect of reducing the memory load on the system. In a traditional UNIX implementation, *fopen* would allocate new memory to the calling process and copy the data from the UNIX buffer cache into that new memory at the time of a *read*. Using this new package and Mach file mapping, each new call to *fopen* will reuse any physical memory containing file data pages, reducing the number of I/O operations.

In addition to traditional buffered I/O calls, the mapped file version of buffered I/O has been extended with a new call that allows an application program to access directly the mapped file data. This further improves performance by eliminating the copying of data by *fread* and *fwrite*. The new routine is called *fmap* and is compatible with buffered I/O.

To read map a file with *fmap* the user calls:

```
stream = fopen("filename", "r");
data = fmap(stream, size);
```

where *data* is a pointer to a region of virtual memory corresponding to the location of the buffered file contents, and *size* is the suggested size for the data buffer. If *size* is zero, the implementation will choose a suitable size. As before,

```
bufsize = fbufsize(stream)
```

returns the actual size of the buffer. Once *fmap* is called, the user can reference file data by using the data pointer and any offset less than *bufsize*. The user may also mix *fseek*, and *fread* calls with direct data references. Once the user is finished with the file the call

```
fclose(stream);
```

should be used to deallocate the virtual address space used by the mapped file.

To map a file for writing the user would:

```
stream = fopen("filename", "w");
data = fmap(stream, size);
```

where *size* is used as an initial buffer size. If *size* is zero, the implementation will choose a suitable size. Initially, the buffer will be zero-filled. Once *fmap* is called, the user may write into any part of the file with an offset less than *bufsize*. An *fwrite* or *fseek* call with an offset greater than *bufsize* will cause an error. To expand the buffer size, the user may call *fmap* again with a larger size parameter. The calls

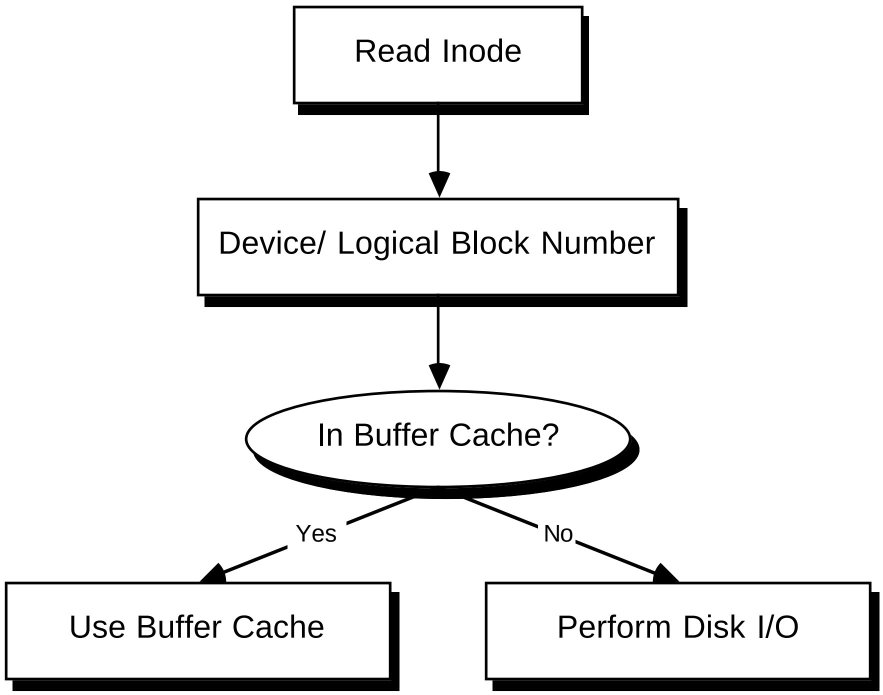

Figure 5.1: File system reads using buffer cache.

```
fflush(stream);
fclose(stream);
```

continue to work as before. Similarly, *fmap* may be used with files opened for append and read/write may be *fmap*ed.

5.1.3 UNIX kernel file mapping

The UNIX kernel implements file read and write operations by translating them into direct accesses to a mass storage device. A specialized cache, called the *buffer cache*, is used to minimize actual I/O operations. In particular, a read or write on an *inode* is translated into a device and logical block number on that device. The buffer cache is searched for that device and block combination. For *read* operations, if the data is in the cache it is used. If the data is not in the cache, an I/O operation takes place, leaving the data in the cache for later use (see figure 5.1). For writes, the data is placed in the cache and written to mass storage asynchronously.

While the use of a buffer cache increases the overall performance of UNIX systems, it has its problems:

- The buffer cache uses some amount of fixed memory. This memory cannot be used for other purposes, such as process memory. Therefore, choosing a small buffer cache results in inefficiency in disk caching; choosing a large buffer cache results in extra paging of process memory.
- The lack of integration of the buffer cache and paging system causes problems when both must access common data on disks. In particular, the paging system must ensure that stale data is not left in the buffer cache and vice versa.

The file mapping primitives provided by Mach have been used to support normal file access as implemented by the UNIX-emulation component of the kernel. This immediately makes a machine's entire physical memory space available for caching all of secondary storage — including task memory and file data.

Using file mapping for file I/O in the UNIX-emulation component logically causes the following changes in the implementation:

- An *open* operation is translated into a file mapping operation.
- A *read* or *write* operation is translated into a simple byte copy of memory to or from the mapped file. Actual file operations that manipulate the data on the disk are handled by the paging system. For example, an actual file read operation is triggered by a page fault (see figure 5.2).
- A *close* operation is translated into a deallocation of the memory corresponding to the mapped file.

In practice, however, using file mapping in the kernel is far more complicated. Some of the issues that must be dealt with are as follows:

- The virtual memory space available to the kernel is usually somewhat more restricted than that of user address spaces. Therefore, it is not possible to assume that all mapped files can simultaneously fit in the kernel's virtual address space. In fact, it is not even possible to assume that a single file can fit into that address space if the file is large.
- Write operations cause the data to be copied from the user's address space to the mapped file. However, this by itself does not guarantee that the data will be written to disk since it simply exists as virtual memory.

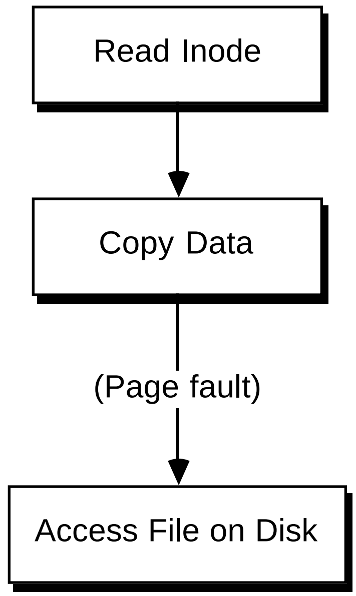

Figure 5.2: File system reads using file mapping instead of the buffer cache.

- The 4.3 BSD file system implementation assumes that the buffer cache holds current data. Using this assumption, it sometimes makes modifications to files by changing the data in the buffer cache. This is done, for example, under some conditions when increasing the size of a file. Unfortunately, a file mapped into virtual memory may not have these changes properly recorded.

- The synchronization used to guarantee atomic modifications to *inodes* is complicated by the fact that a paging operation can be transparently invoked. It is therefore not possible to access any data that may generate faults eventually leading to an I/O operation on an *inode* while holding a lock on that *inode*.

Each of these problems can be solved, but the solutions usually add complexity or inefficiency to the implementation. For example, to solve the problem created by the restricted kernel virtual address space, it is necessary to *window* files. Windowing files means that only some part of a file, the window, is mapped into virtual memory. To perform file operations on mapped buffers, the file is windowed - fixing the range of the file to be accessed in virtual memory. The operation takes place (as a byte copy), then the window is released. The actual implementation of windowing used by the kernel leaves files mapped for as long as possible. Virtual address ranges used by file mapping are kept on an LRU list and are reclaimed for use in other file mapping operations whenever necessary.

It is interesting to note that although extra windowing operations are needed to perform read and write operations, they actually cost less than one might expect. This is largely due to the object cache, which keeps files cached in memory even though they may not be actively referenced (mapped).

All file write operations, implemented as byte copies into kernel virtual memory, are immediately followed by data flushes to the corresponding *inode* on disk. This guarantees that the data will be kept permanently on disk, preserving the file system semantics. However, it has the disadvantage that files used for temporary data — a common occurrence in UNIX systems — are written to disk when, in fact, they may be deleted in the near future.

5.2 Copy-on-write

One of the key performance tricks used in the virtual memory system is copy-on-write. Although copy-on-write is a completely transparent optimization that the user can not make use of directly, its existence makes it possible to implement portions of the system very efficiently. Two of the most important improvements gained by using copy-on-write occur in the implementation of Mach IPC and UNIX fork.

5.2.1 IPC

The Mach IPC subsystem makes use of the virtual memory system to copy large amounts of data in messages. When a task sends a message with a large amount of data, the data is copied (using virtual memory primitives rather than a byte copy) into a special address space used by the kernel: the *ipc map*. The ipc map, which is never directly accessible by the kernel or any user task, is simply a software map used as a handle for the copied virtual memory. When another task performs a receive operation that will receive the data, it is copied into the receiving task's virtual address space (see figure 5.3). At this point, the entire region of virtual memory has been copied (virtually copied) into the receiver's address space and is now available for use as normal virtual memory. Absolutely no data has been physically copied — and none will be until either the sender or receiver tries to modify data.

The effects of using copy-on-write techniques for message passing were examined in detail in Accent[21].

5.2.2 Fork

In the early days of UNIX, the typical process size, limited by address restrictions on the PDP-11, was only a few kilobytes. At that time, the UNIX *fork/exec* primitives, used for process creation and execution of other programs, did not suffer high costs due to copying an address

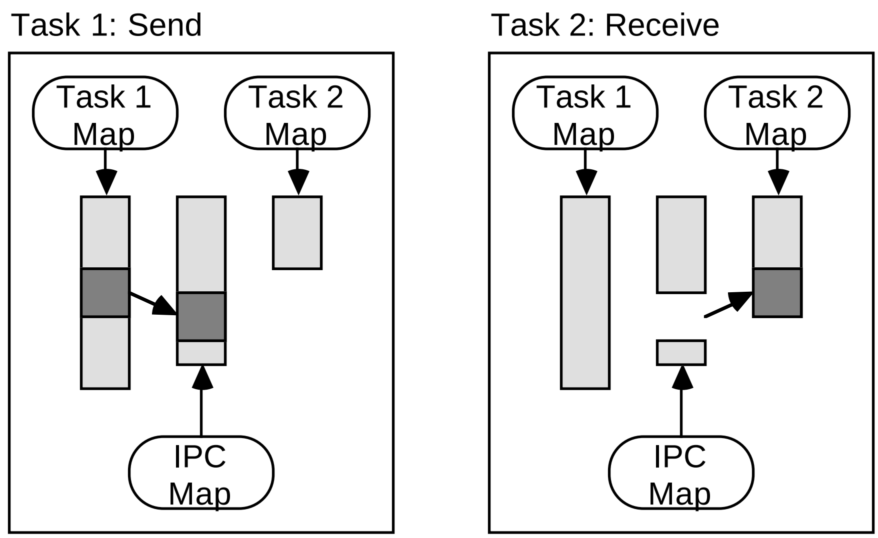

Figure 5.3: Using virtual memory to implement IPC.

space. However, modern applications running on modern UNIX systems are rarely so small. In fact, applications usually measure in the hundreds of kilobytes rather than tens. With this type of increase in software systems, the cost of copying an address space for a *fork* and then throwing it away for an *exec* becomes much more noticeable.

A simple solution to this problem, available in BSD UNIX, is the *vfork* primitive. *Vfork* is intended as a replacement for *fork* when the user is going to follow the *fork* with an *exec*. *Vfork* has the advantage that most of the *fork* semantics are preserved, but no data is copied. Unfortunately, not all of the *fork* semantics are preserved. In fact, to implement *vfork* the parent process is suspended until the child process terminates or *exec*s. Both the parent and child process use the same address space, removing the need to copy the data. However, this has one unfortunate outcome: any modifications made to the address space by the child are reflected to the parent. This severely limits what the child can do.

The *vfork* primitive is a clear example of what can happen to an operating system when the proper primitives or optimizations are not available. In this case, the missing optimization is copy-on-write, which could not be easily added to the BSD virtual memory system. In Mach, the copy-on-write optimization is a central, transparent feature. Therefore, simply implementing the UNIX *fork* and *exec* primitives in terms of appropriate Mach virtual memory primitives yields the desired copy-on-write optimization.

5.3 Shared Libraries and Program Loading

Shared libraries provide the capability to reduce both the memory usage and disk storage required by a large collection of programs that share program code. Shared libraries have been implemented in many ways. Most of the design decisions involved in shared library implementations do not pertain to virtual memory management.

In fact, it is possible to implement forms of shared libraries without virtual memory management. For example, the Macintosh Toolkit, largely a subroutine library implementing the Macintosh user interface, need not be linked into every binary. Instead, it is available to all programs via a mechanism which traps to ROM routines. The lack of memory management in the Macintosh does not prevent this form of shared library. However, the Macintosh by no means supports a general shared library mechanism.

Moving from a non-virtual memory machine like the Macintosh to a machine supporting separate, protected virtual address spaces actually increases the difficulty of implementing shared libraries. In the Macintosh, the shared library is at a fixed, well-known address addressable by any program directly. In a virtual memory based machine, it is necessary for a virtual to physical mapping to exist between some virtual address and the physical address (or secondary storage address) corresponding to the shared library.

Most operating systems provide facilities for shared libraries in specialized ways, treating shared libraries as special system objects. For example, VMS supports shared libraries as a special type of executable program that only uses memory resources. System V UNIX supports shared libraries by fixing them at a specific virtual address, then using internal mapping primitives to share the actual code[3]. Some operating systems do not provide for shared libraries at all. In this type of operating system, a general shared library mechanism has been deemed unnecessary (for example, the Macintosh) or other mechanisms are employed to minimize the need for shared libraries. In the case of the latter, 4.3 BSD UNIX supports a notion of shared text segments, which allow all invocations of the same program to share all code.

The fact that shared library schemes are often developed using specialized mechanisms (e.g., VMS or System V) or missing entirely (e.g., 4.3 BSD UNIX) is a direct result of the lack of appropriate virtual memory functionality. A properly designed virtual memory system can allow shared library implementations to be developed easily without depending on specialized system facilities.

Mach has no need for such specialized system facilities. Instead, the mapped file interface is used to implement both program loading and a general form of shared libraries. In the current Mach system, there are two types of program loaders. The first program loader executes in the kernel and implements the UNIX *exec* system call. This loader handles both *a.out* and *COFF* format binary files for binary compatibility with existing systems (*a.out* is used on Berkeley derived systems, *COFF* is used on System V derived systems). The second loader executes in a user task and handles *MACH-O* format binary files. Both loaders use mapped files.

5.3.1 MACH-O File Format

The MACH-O format is flexible enough to handle a multitude of applications:

- *fully resolved binaries*: Executable files that can be loaded directly into virtual memory and executed.
- *unresolved object files*: Object modules that contain external references.
- *shared libraries*: A special form of object module that may be shared by many binaries.
- *core file*: Snapshots of an executing task's virtual memory space and the state of its threads.

In addition, the format provides enough backward compatibility with older formats (e.g., a.out) to salvage most existing code for debuggers and related applications.

The MACH-O format can be thought of as a sequence of commands to be executed by a program loader. The layout of a MACH-O file is summarized as:

```
header
command_id, command_info
command_id, command_info
          .
          .
          .
command_id, command_info
```

Each command consists of a command identifier followed by a command-dependent number of arguments. Some of the commands supported are:

- **MAPFILE**: map part of a file into virtual memory.
- **VM_REGION**: pass virtual memory to a new task.
- **LOADFILE**: load commands from another MACH-O file.
- **THREAD**: create a thread in the task.

The *header* contains a magic number indicating MACH-O format. It also contains other useful information such as version information and a machine-type specifier. Finally, the header specifies the type of file represented, e.g., executable, object file or shared library. The loader directives (commands) are allocated contiguously after the header information.

5.3.2 MACH-O directives

The MAPFILE directive causes data in a file to be mapped into the address space. The data to be mapped is specified by an *offset* and a *length*, both in bytes. The file containing the data is specified by a normal file name. A null file name indicates zero-filled memory. Version information, contained in the *version* field is discussed later in Section 5.3.3. This directive is typically used to map code or data into an address space.

The VM_REGION primitive permits user programs to pass regions of shared or copy-on-write memory from parent to child. This directive, in combination with vm_inherit, allows memory regions to be shared among heterogeneous programs. For example, suppose program A wants to communicate via shared memory with programs B and C. Prior to executing A, the programmer (or user) must decide where in memory the shared region should be. Next, the programmer would use the *convert* program (described later in section 5.3.5) to insert the required VM_REGION command into the loader directive list for programs B and C. With this initial state, program A can simply *vm_allocate* the memory region, set the memory region's inheritance to VM_INHERIT_SHARE with *vm_inherit*, and finally execute programs B and C.

The LOADFILE directive indicates to the loader that an additional library, specified by a pathname, is to be loaded. If the pathname is not absolute, then the search for the named library proceeds along the user's LPATH environment variable. The *major* field within the version information must match the version information provided in the library's header. Recursive LOADFILE directives are supported for libraries which in turn share code with other libraries.

The THREAD directive allows all registers (excluding any machine-dependent privileged processor status registers) to be specified by the user. All information provided by this directive is passed directly to the *thread_mutate* call provided by the Mach kernel. Multiple THREAD commands can be specified, resulting in the automatic initialization of a multithreaded task.

5.3.3 Version Information

Unlike traditional UNIX libraries, where code and data are statically included in executable files at link time, programs using shared libraries have a serious version control problem. Changes to the exported routines of a shared library can have a profound effect on the correctness of any executable file which references it. With the old style statically linked format, this would not be a problem, since the library code is included directly in the file image. Whether a program works is independent of any later changes to the libraries, provided the libraries are correct when the program is linked. To help alleviate this problem, the MACH-O file format encodes version information in both the header and in the LOADFILE directives. When executing a file which references a shared library, the major version number found in the LOADFILE directive of the program is compared with the major version number found in the header of the shared library. If they agree, the shared library and the executable file are assumed to be compatible. The minor version number is not used by the loader but can be used by library maintainers to identify the current library version.

5.3.4 MACH-O program loader operation

The MACH-O program loader operates by scanning a load file and executing commands as necessary. In the typical case, it uses the *map_fd* call to map portions of files into its address space. Since copy-on-write is used at the base of the virtual memory primitives it is possible to share both code and writable data. Each task that writes data within a shared library will get a new copy as each page is written for the first time. Pages that are not written will be physically shared by all tasks.

The loader keeps track of the allocated regions of the child's address space while it is executing the loader directives. Thus, the loader prevents multiple allocation of the same memory region and deallocates all unused regions in the child's address space as a final step.

5.3.5 The Convert Program

A special program, *convert*, aids the management of MACH-O format files. The convert program provides three distinct functions:

- it creates a shared library from a collection of object files (.o),
- it links programs with shared libraries and stores the resulting executable file in MACH-O format, and
- it converts standard a.out format executable programs into .macho format files.

When used in conjunction with other system utilities, the convert program allows one to create shared libraries and executable files in MACH-O format.

5.4 Shared Memory

Several research projects at CMU are actively using the shared memory facilities of Mach to build efficient distributed and parallel applications. For the most part, shared memory is used to avoid some of the communication costs inherent in distributed applications based solely on inter-task communication. Examples are the Camelot distributed transaction system [45] and the Agora knowledge representation system[8].

5.4.1 Camelot

The Camelot Disk Manager manages backing store for recoverable objects and coordinates access to a stable storage log. Data servers, which manage the recoverable objects, must communicate with the Disk Manager to pin objects and spool descriptions of modifications to the log. To reduce the cost of these operations, shared memory queues are used for communication. Each data server has a shared memory queue but no access to other data server queues. The shared memory queue is implemented as a fixed array with a head and tail pointer.

The queue is used to asynchronously queue *pin* and *log* messages to the Disk Manager. No special synchronization is required between the data server and the Disk Manager. The head of the queue is read and written by the data server, and only read by the Disk Manager to determine if the queue is empty. Conversely, the tail is read and written by the Disk Manager, and only read by the data server to determine if the queue is full.

A detailed description of Camelot can be found in [46].

5.4.2 Agora

The Agora system is designed to support the development of parallel applications written in multiple languages. The fundamental abstractions of Agora allow parallel applications to share data structures, regardless of their location in a distributed environment. Both communication and memory sharing are used to implement the system.

A speech recognition system, programmed using Agora, illustrates how Agora uses both communication and memory sharing to implement the shared memory abstraction. Agora maintains a shared blackboard structure in which hypotheses are placed and evaluated by multiple cooperating agents. The blackboard physically resides on a multiprocessor host. All accesses to the blackboard are through a procedural interface that determines if shared memory or communication must be used. Agents use shared memory to directly modify the blackboard. Message passing is used between loosely coupled components of the system that collect data, perform low level signal processing and display results. The total system distributes itself among a collection of machines consisting of a VAX 8200 with four CPUs and a number of MicroVAX IIs, IBM RT/PCs and Sun 3s interconnected via several Ethernets and IBM token rings.

Chapter 6

Performance

This chapter reports on the performance of the Mach virtual memory system. Performance is examined at two distinct levels: low-level operations and overall performance characteristics. In addition, the effects of using Mach's file mapping primitives are examined from a performance perspective. Where appropriate, direct comparisons are made with other, sometimes commercially developed, virtual memory systems.

6.1 Tools

In order to measure the performance of the virtual memory system, it was necessary to build a suite of several test programs. The test suite consisted of several small programs, each able to measure a specific aspect of the virtual memory system. The test programs fell into the following categories:

- *generic UNIX*: These programs restrict themselves to UNIX features. They are used to measure the performance of UNIX and Mach.
- *Mach specific*: These programs use Mach specific features. They are used to measure the performance of functions available only in Mach.

Test programs that run under both Mach and UNIX were used to measure zero-fill performance, fork performance and file system throughput. Mach specific programs were used to measure page fault overhead, copy-on-write costs, address map manipulation overhead and file system throughput using file mapping. In addition, other Mach specific programs were used to measure the parallel performance of the system.

Several other tools were used in order to tune and measure the performance of the system:

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

Table 6.1: Performance of zero fill per K of data.

- *kernel statistics*: The kernel maintained several statistics (accessed via the *vm_statistics* interface).
- *kernel profiling*: Kernel profiling was used to determine which parts of the implementation consumed the most CPU resources.
- *VM tracing*: A special program was used to monitor the virtual memory related activity of other programs.

All system timing was performed using the timing facilities provided by UNIX. In most cases, these timers had a resolution that depended on the rate of an interrupting clock (100 Hz on Vax, 50 Hz on Sun-3, 64 Hz on RT/PC). Timing on the Multimax, based on its microsecond timer, was a notable exception.

6.2 Low-Level Performance

6.2.1 Uniprocessor Performance

Tables 6.1 and 6.2 compare the performance of Mach's zero fill (filling a page with zeros on demand) 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 .25 ms. This leaves .3ms and .95ms of overhead for the Mach and BSD fault handlers, respectively (per K). Since the Mach page size is 4K, the actual overhead is approximately 1.2ms. This compares favorably with the .95ms 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 6.2). Since Mach implements fork using copy-on-write, it avoids the copy costs that inflate

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

Table 6.2: Performance of fork primitive (256K process)

	DEC		SUN		IBM
Operation	MicroVAX-II	VAX-8800	3/160	3/260	RT/PC
Remapping fault	1.18ms	.31ms	.66ms	.31ms	.94ms
vm_allocate/vm_deallocate	.83ms	.22ms	.46ms	.22ms	.38ms
vm_copy	1.22ms	.24ms	.64ms	.36ms	.79ms

Table 6.3: Performance of Mach VM primitives

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 (see 5.2.2). However, the time for a basic fork under Mach (24ms on MicroVAX-II) is comparable to a BSD vfork (17ms on MicroVAX-II).

The time to perform a fork operation on a UNIX system without copy-on-write is directly proportional to the size of the process that is executing fork (see Figure 6.1). In Mach, this time is directly related to the amount of memory that is used (see Figure 6.2). However, since Mach uses page faults to trigger copy operations, a fork can be more expensive under Mach if a large amount of data is used. In figure 6.2, the effect of modifying a varying percentage of data in the child (by writing the first byte of a page) is measured. This figure shows that Mach continues to outperform UNIX on the Sun 3/260 until more than 70% of a 512K forked process is modified.

The cost for other frequently performed primitives is summarized in Table 6.3.

6.2.2 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 megabytes of primary memory.

Figure 6.3 shows the parallel performance of zero fill operations. Performance increases with the number of processors in a nearly linear fashion. The deviation from the ideal case of perfectly linear speedup is most likely due to bus contention since zero filling data generates a large amount

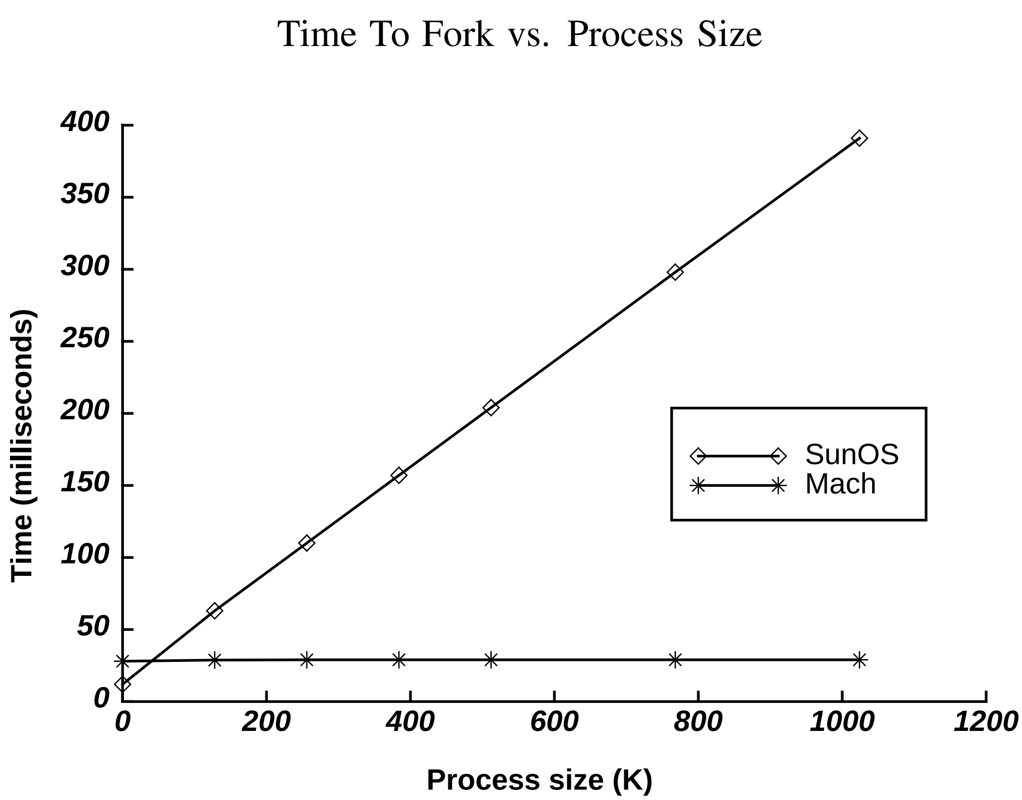

Figure 6.1: Time to fork on a Sun 3/260, no pages touched in child

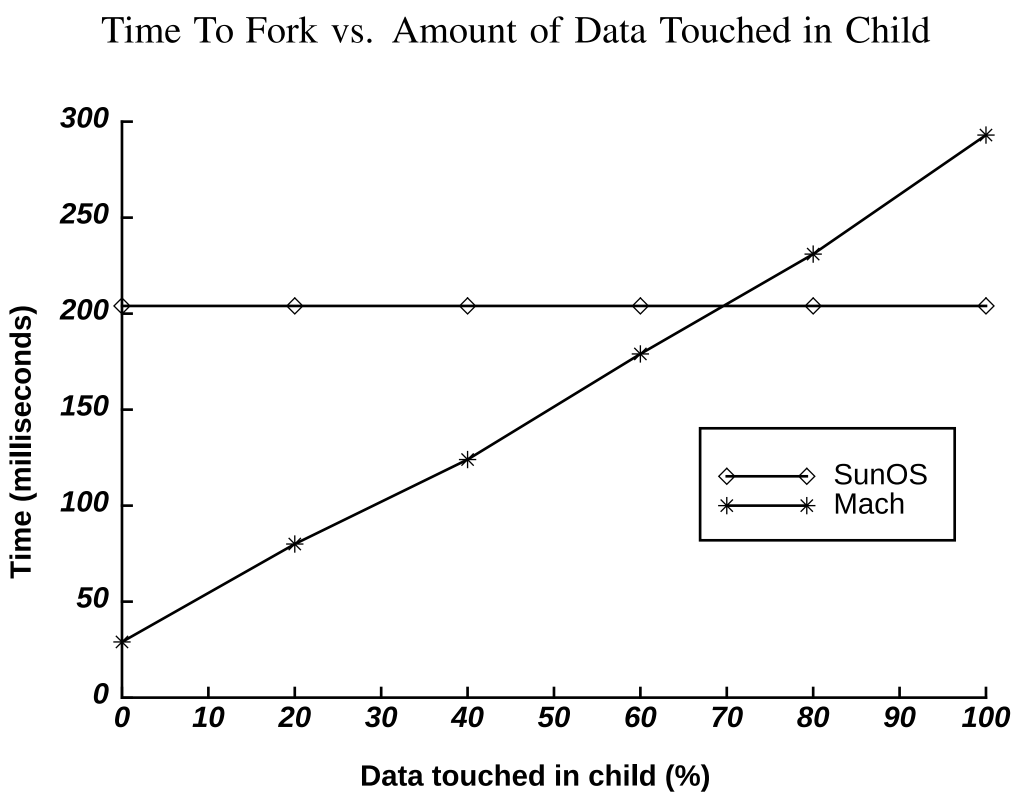

Figure 6.2: Time to fork versus amount of data written in child on a Sun 3/260 (process size is 512K)

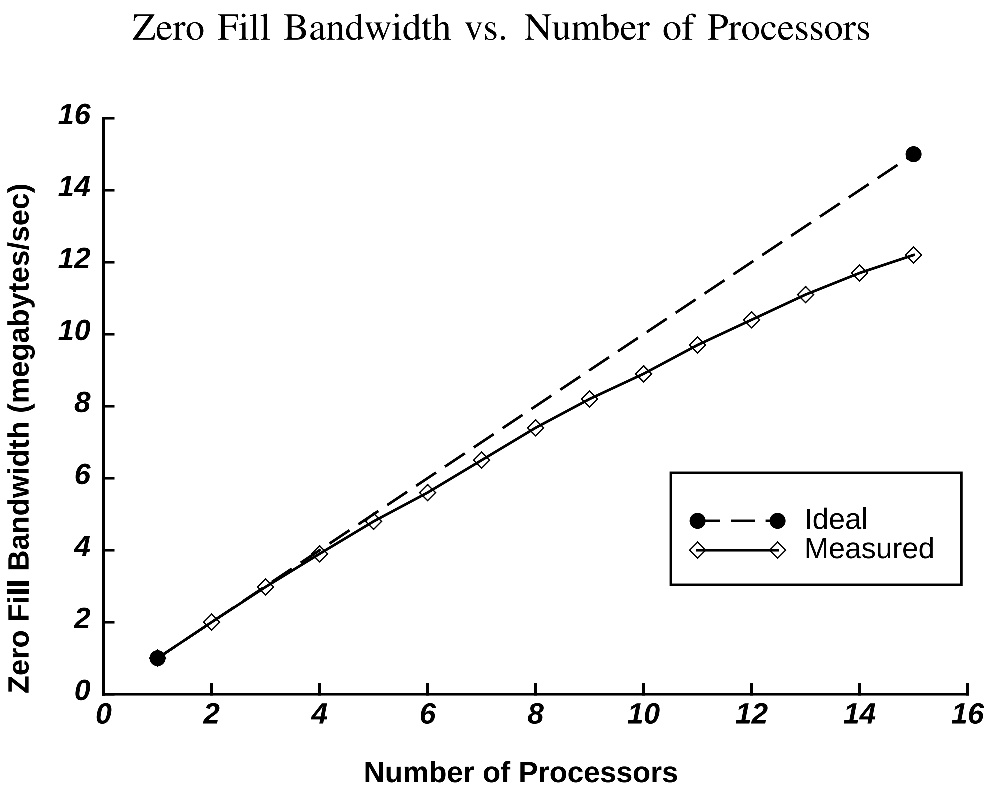

Figure 6.3: Parallel performance of zero fill operations on Multimax

of bus traffic.

As shown in Figure 6.4, the parallel performance of page fault handling does not perform as well as zero fill. Performance increases linearly until approximately 5 processors. Performance then levels off at 6.7 processors. This test, which causes processors to page fault at their maximum rate, shows that the worst possible parallel performance of the fault handler is just under 7 processors.

The worst case parallel performance of about 7 processors is not surprising. Manipulation of the paging queues (active, inactive and free) is performed under a single lock. The fault handler obtains this lock at least twice in order to process each page fault. The first time, it removes the page from either the active or inactive queue to prevent pageout from occurring while the page is in use. The second time, it places the page at the end of the active queue, making the page eligible for pageout as well as simulating LRU.

Manipulation of these queues, performed while the paging queue lock is held, can easily account for 1/7th of the time necessary to handle a page fault. Elimination of this single point of

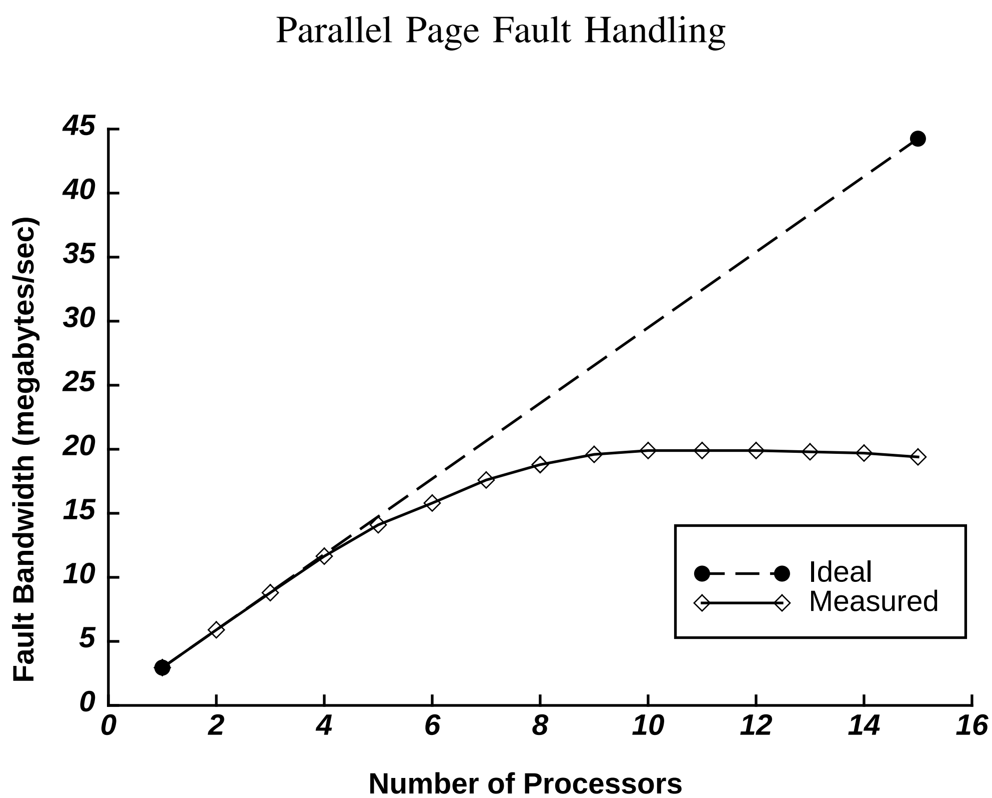

Figure 6.4: Parallel performance of page fault operations on Multimax

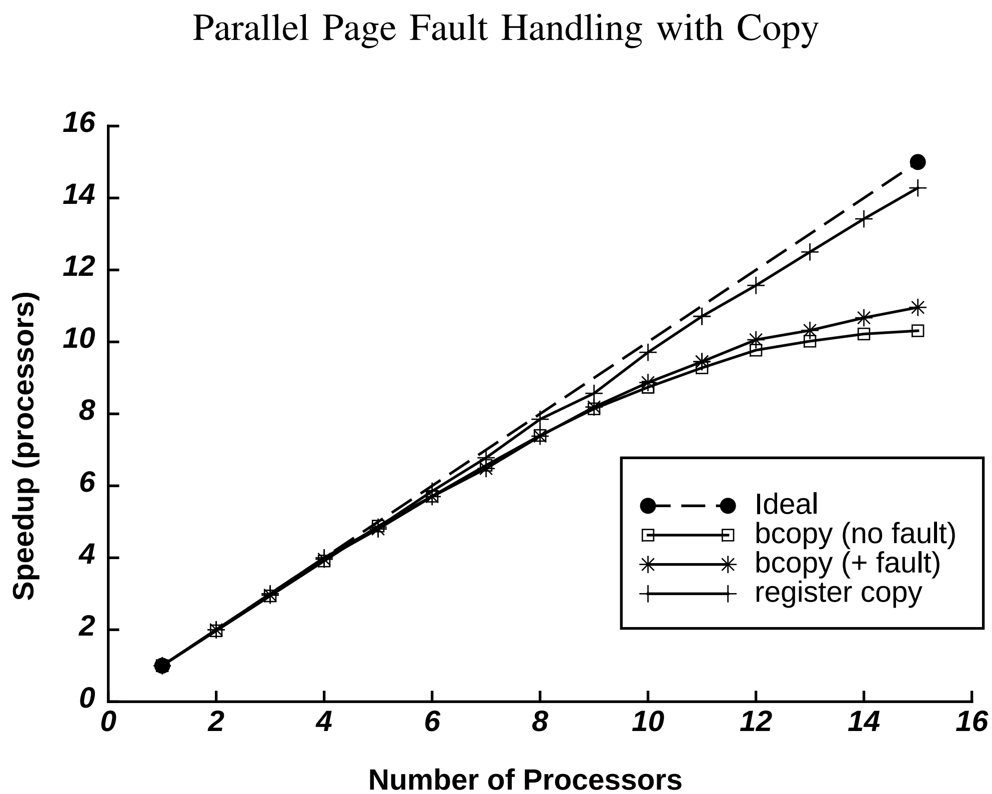

Figure 6.5: Parallel performance of page fault operations (with copy) on Multimax

serialization would result in a system that exhibited higher parallel fault performance.

Figure 6.5 shows the effects of manipulating the data that is faulted. The data is manipulated in three significant ways:

- *bcopy (no fault)*: Data is 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 is simply copied and no page faults are generated.
- *bcopy (with fault)*: Data is copied using the UNIX C library bcopy routine, but 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.

This shows that worst case parallel performance exhibited in figure 6.4 is unlikely to be a

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

Table 6.4: Overall performance: compiling a subset of programs in /bin.

bottleneck on multiprocessors with approximately the Multimax's number of processors. Assuming that faulted data is processed in some way, the cost of this processing is likely to dominate the fault handling cost. The three types of data manipulation measured in Figure 6.5 show:

- The parallel performance of copying faulted data is no worse than the parallel performance of data that need not be faulted. In fact, it is slightly better because it reduces the percentage of bus write operations per unit time (i.e., the amount of parallelism possible increases because execution of the fault handling code, largely read intensive, reduces the bus bandwidth required).
- A fast algorithm that examines each byte, but does not generate excessive bus write operations (e.g., move each byte into a register), can exhibit nearly ideal parallelism.

In particular, the minimum time to examine each byte in the page is achieved using the bcopy routine. However, when the bcopy routine is used, any loss in parallelism is due to the machine's architecture rather than virtual memory system bottlenecks. Real applications, which usually do additional processing, will need to be more concerned with the machine's architecture than the virtual memory system's parallelism.

6.3 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 6.4 compares the performance of Mach with 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 an unmodified 4.3 BSD release. Both the 4.3 kernel and Mach kernel were used on this system. The Sun 3/260 tests were performed on separate Suns due

Machine	MAP1	MAP2	READ1	READ2	STDIO
DEC MicroVAX-II	.24	2.9	.20	1.0	.037
Sun 3/260	1.1	13.3	.85	3.0	.18
IBM RT/PC	.29	3.4	.23	1.9	.039

Table 6.5: File system throughput in megabytes per second.

to the fact that the Mach kernel is not completely compatible with Sun specific kernel extensions. However, the same compilation environment was used in each case.

The most important information contained in Table 6.4 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 previously terminated. The reduction in I/O operations has the direct effect of lowering the elapsed time.

The user and system times for the compilation test are nearly identical for Mach and UNIX. This shows that the virtual memory system does not impose extra overhead. It is also the case that the Mach systems under test supported multiple threads per task in its UNIX emulation. This support adds as much as 10% overhead to system time[49]. In addition, user time is slightly affected due to more expensive clock tick handling (again, because of thread support).

6.4 File Mapping Performance

To illustrate the effects of file mapping on performance, figure 6.5 shows the results of processing file data in different ways. In particular, the data is processed in the following ways:

- *MAP1*: Maps the file and examines each byte.
- *MAP2*: Maps the file and references each page, but does not examine each byte.
- *READ1*: Uses the *read* system call and examines each byte.
- *READ2*: Uses the *read* system call, but does not examine each byte.
- *STDIO*: Uses *fgetc* from the UNIX stdio package to read each byte.

The MAP2 and READ2 tests show that the maximum throughput possible is higher using file mapping primitives. This increase, due to the elimination of a copy operation, is especially high when data copying costs are high relative to the machine's processor speed (e.g., a speedup of

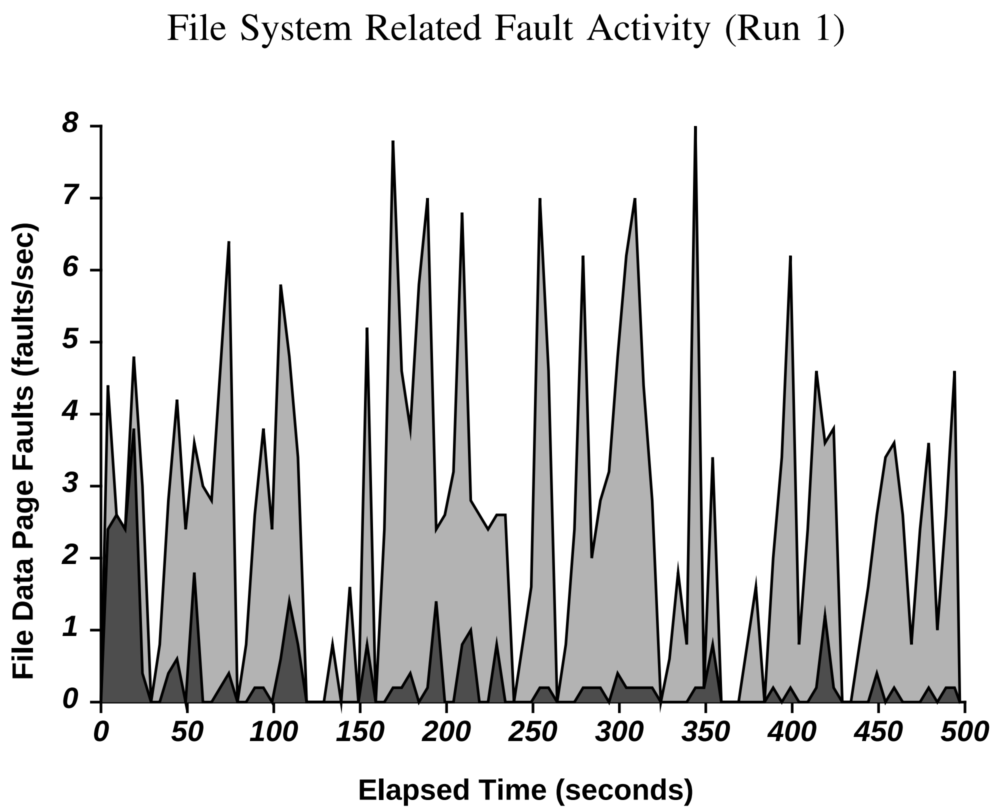

Figure 6.6: Page faults generated as the result of mapped file references during the first run of a compilation test.

4.4 times on a Sun 3/260). Even when all data is referenced (MAP1 and READ1), file mapping still performs at least 25% better than the read operation.

Figures 6.6 and 6.7 illustrate the effects of using file mapping to implement read and write inside the kernel. These figures show the virtual memory activity while running the compilation tests described in section 6.3. In each figure, the lightly shaded area corresponds to page faults that occur due to data references on files (from both the read/write interface and demand paging of executables). The dark area corresponds to those faults that actually require I/O operations to occur on the file system (as opposed to hits in the memory cache).

Figure 6.6 corresponds to the first run of the compilation test. In this figure, it is shown that nearly all references hit the cache. Those that miss correspond to the initial loading of the compiler and the first reference of each source file. Figure 6.7 corresponds to a second run of the same compilation. In this example, every file system page is already cached and there is no need to perform any disk I/O operations for file system data.

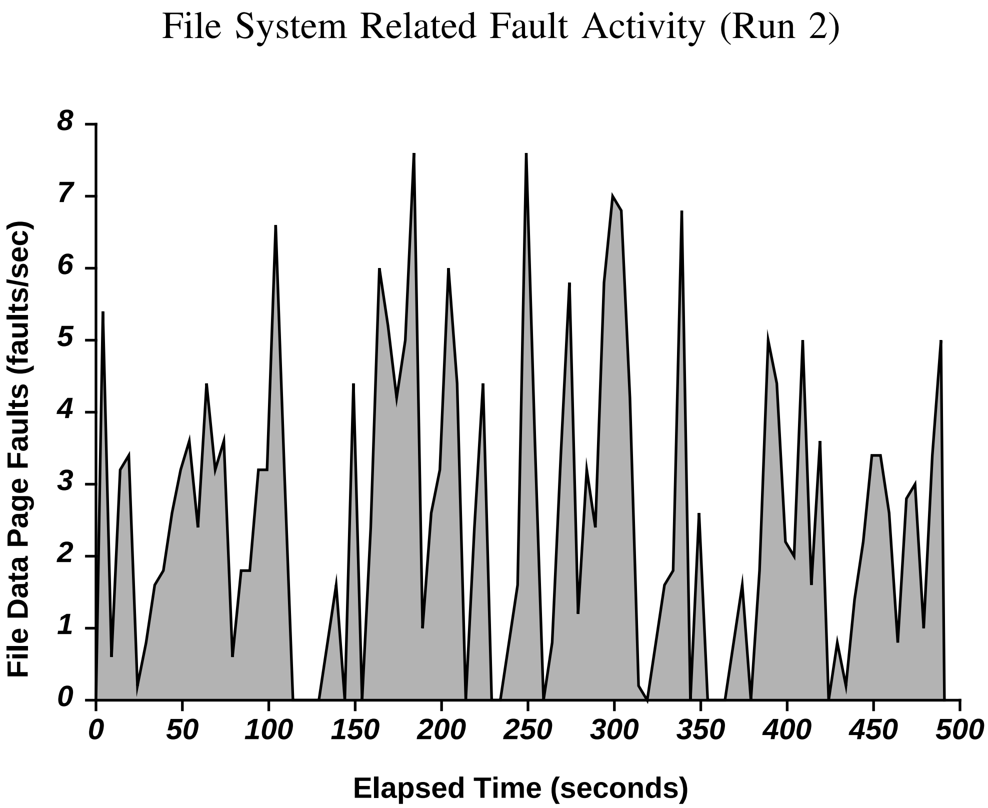

Figure 6.7: Page faults generated as the result of mapped file references during the second run of a compilation test.

6.5 Conclusions

This chapter has shown that although the Mach virtual memory system requires manipulation of data structures that are far more complex than their UNIX counterparts, the performance of low-level operations is still better in Mach. This is largely the case because the Mach data structures and algorithms are designed based on how systems are used rather than on how hardware can easily read them. Although some primitive operations are more costly in Mach than other systems (e.g., basic fault time), lazy evaluation techniques, such as copy-on-write, and object caching, made possible with the high level, machine-independent data structures, usually result in better overall performance.

It has been shown that the virtual memory system does not present a bottleneck for parallel applications on Multimax-like multiprocessors. Although fault handling does not exhibit complete parallelism, the parallelism is sufficient for applications on this class of multiprocessor. In fact, it has been shown that the machine's architecture is more likely to cause a degradation in performance than the virtual memory system.

Finally, this chapter has shown that the use of file mapping can result in an improvement of applications that are limited by the bandwidth of file system throughput. Applications that can be recoded to use file mapping primitives can eliminate the built in overhead associated with the UNIX read/write or standard I/O interfaces. Further, by integrating the file system with virtual memory, it is possible to make more effective use of primary memory as a cache. This reduces the number of actual I/O operations that must take place.

Chapter 7

Hardware Experience

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 the Mach virtual memory implementation consists of a single code module and its related header file for each architecture. In addition to improving portability, this separation of software memory management from hardware support 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.

7.1 Porting Mach VM

The Mach virtual memory code described in this thesis 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 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 and invalidation of hardware address translation buffers were required.

Implementations of Mach on the Sun 3, Sequent Balance and Encore MultiMax have each

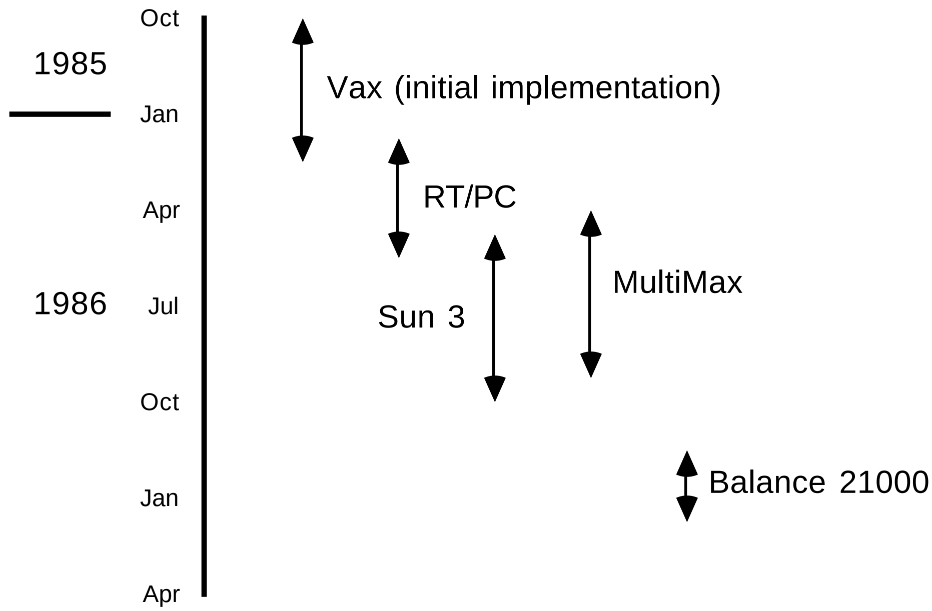

Figure 7.1: The timing of the Mach implementation and several ports.

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. Figure 7.1 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 auto-increment of a bit field which exercised a compiler bug in the RT/PC compiler. The only other change was to eliminate an informational printout of some automatically computed paging parameters since it interfered with other diagnostic output generated by the MultiMax.

7.2 Assessing Various 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 lookaside buffer (TLB) 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.

7.2.1 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 addressability 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 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 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.

As specified by the VAX architecture, each page table is broken down into two regions:

- *P0 region*: The P0 region typically contains code and data for a process.
- *P1 region*: The P1 region is typically used for the process' stack.

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 multi-threaded Mach applications that need a stack for each thread.

The IBM RT/PC does not use per-task page tables[52]. 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 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 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 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 implement sparse addressing reasonably, 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.

Both the Encore Multimax and the Sequent Balance 21000 use the National 32082 MMU[36]. 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. Once again, this requirement is very restrictive in large systems[1].
- 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.

7.2.2 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 supports TLB consistency. In order to guarantee such consistency when changing virtual mappings, the kernel must determine which processors have an old mapping in

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

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 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 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 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 if that protection is increasing. TLBs (or page tables) containing stale data will cause a protection fault that will be handled properly by the fault handler.

Chapter 8

Memory Architectures

In its simplest form, the memory subsystem of a machine's architecture can be thought of as a fixed size array of bytes. For the most part, this type of memory architecture is an accurate description of available hardware architectures. The Mach virtual memory system is based on this type of memory architecture. This chapter studies other types of memory architectures in both the uniprocessor and multiprocessor domain. The impact of these architectures on the virtual memory system is examined.

8.1 Uniprocessor Memory Architectures

In uniprocessor systems, the memory subsystem is typically a large main memory with a small cache. The large main memory usually consists of inexpensive, slow memory. High memory performance (bandwidth) is gained with the use of a small (in relation to main memory size), fast cache memory which satisfies most memory references.

Cache memories can be either *physical* or *virtual*. A physical cache operates on physical addresses whereas a virtual cache operates on virtual addresses. A physical cache can achieve the desired performance gains at no cost in system support as it is completely transparent to software. However, a virtual cache, which is not transparent to software, requires direct management by system support software.

With the exception of the virtual address cache, uniprocessor based memory architectures generally present few problems for a virtual memory system. Nevertheless, the existence of virtual address cache systems present problems that must be addressed if a system is to run on such a machine. For example, a virtual address cache that does not write through to memory on all writes (a *write-back* cache) will leave stale data in main memory until the operating system forcibly flushes cached data back to main memory. This means that it is impossible to share memory for both read and write within a single address space at different virtual addresses. An

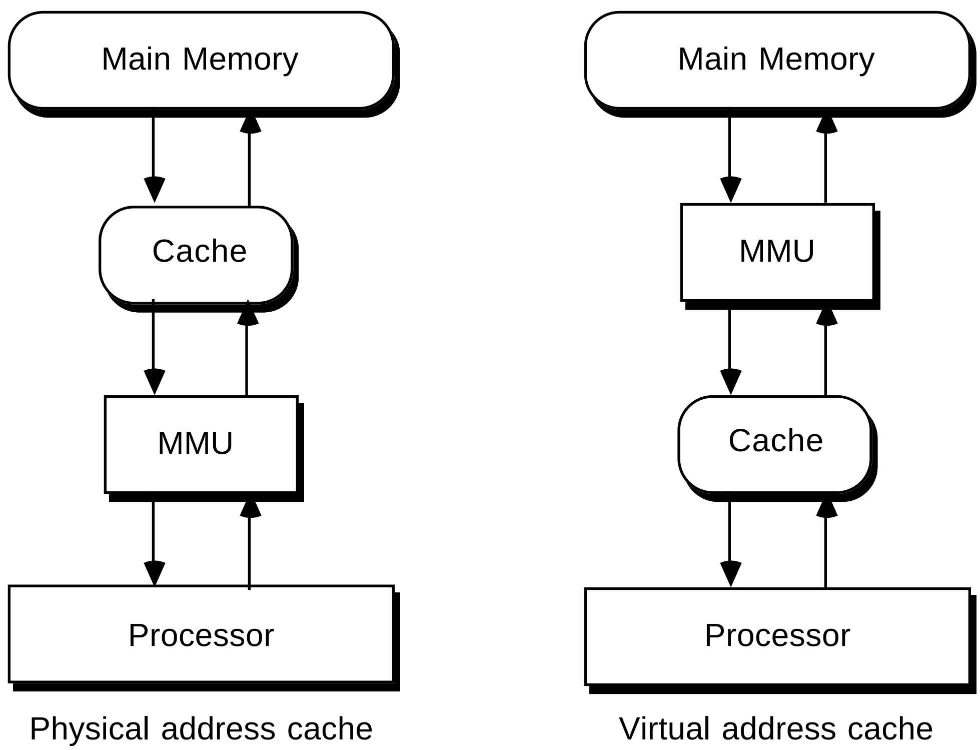

Figure 8.1: Simple cache architectures: virtual vs. physical

attempt to perform such sharing would result in temporary inconsistencies in the data at each virtual addresses, and could even lead to the loss of data.

The separation of machine-independent and machine-dependent memory management algorithms and data structures facilitates operating system support for virtual caches. All modifications to virtual to physical mappings are well defined by calls to the pmap module. This allows the pmap module to perform cache flushing when necessary on a completely machine-dependent basis — transparent to all machine independent algorithms.

In addition, the exported virtual memory interface reduces many of the potential problems associated with a virtual address cache. For example, read/write sharing can only occur by inheritance, guaranteeing that shared memory is at the same virtual address. Further, memory that is shared copy-on-write is only shared for read at different virtual addresses, preventing the possibility of a task writing data that could become stale.

8.2 Multiprocessor Memory Architectures

Multiprocessor architectures raise additional concerns for virtual memory systems. The problems that arise in multiprocessor systems can be characterized as follows:

- memory subsystems do not necessarily provide uniform access to all memory from any processor.
- cache coherency between caches on separate processors is not necessarily guaranteed.
- even when cache coherency is guaranteed, coherency of translation lookaside buffers for virtual address translation is not necessarily guaranteed.

To better understand some of the problems caused by multiprocessor memory architectures, it is useful to consider several types of such machines. There are three types of multiple instruction stream, multiple data stream (MIMD) multiprocessors that have gained commercial acceptance:

1. <u>u</u>niform <u>m</u>emory <u>a</u>ccess multiprocessors (*UMAs*) with access to fully shared memory and nearly uniform memory access times for all data from all processors,
2. <u>n</u>on-<u>u</u>niform <u>m</u>emory <u>a</u>ccess multiprocessors (*NUMAs*) with access to a shared memory in which individual processors can access some memory rapidly and other memory more slowly, and
3. <u>no</u> <u>r</u>emote <u>m</u>emory <u>a</u>ccess (*NORMAs*), message-based multiprocessors which communicate over an internal or external network.

UMA systems are the most prevalent in the commercial marketplace. They are the easiest of the three types to use because they are the most similar to traditional multiprogrammed uniprocessors. Shared memory is typically provided on some kind of shared bus with individual processors accessing the bus through a cache.

On UMA machines with caches, the caches are transparently kept coherent. Several different strategies can be used to guarantee cache coherency. Some strategies that have been used in commercial multiprocessors are:

- *cache invalidation*: A write to shared memory causes a cache invalidate signal to be sent to each processor. Each processor then checks to see if the data is in its local cache and invalidates it if appropriate.
- *snoopy caching*: Each processor's cache *snoops* the memory bus watching for write transactions. When a cache sees a write transaction, it checks to see if the data is in its local cache and invalidates it if appropriate.

These cache coherency strategies work well for multiprocessors with small to medium numbers of processors. It is possible to build machines with dozens of processors, providing nearly uniform access times with data consistency. However, depending on the architecture, cache flushing and bus saturation can result in non-uniform memory access times for some algorithms. Some examples of UMAs are the Encore Multimax, Sequent Balance, DEC VAX 8300 and VAX 8800.

One of the problems with building UMAs is that they often rely on a shared global bus. As the number of processors increases, the bus saturates, limiting scalability of the number of processors and memory bandwidth. NUMAs typically avoid this scaling problem by associating with each processor a local memory. A local memory can be accessed by other processors, but only at a time cost greater than that for its own local memory. The typical processor to processor interconnect is a communication switch with several levels of internal switching nodes. While such switches can be made to accommodate large numbers of communicating processors, they add the characteristic NUMA remote memory access delay.

The difficulties in keeping cache contents consistent through switches have led most NUMA designers to either:

1. provide no cache memory, or
2. provide caches without consistency

Machines that provide no cache memory avoid the cache consistency problem completely, but at the expense of a potential performance loss. Machines that provide caches generally do not provide for consistency. Although it is possible to implement cache consistency in a switch based machine, no existing commercial system does so. Machines with inconsistent caches provide mechanisms for software to manage data coherency. The first, and easiest, way to do this is to disable caching for data that is shared and modifiable. However, this can cause needless performance loss due to high switch traffic. Another mechanism is to allow data to be cached and force software to flush caches when necessary.

Some of the earliest experimental multiprocessors were examples of NUMAs. For example, CMU's CM*[26] multiprocessor was essentially a combination of small clusters of processors. Each processor had its own local memory and could address the local memories of other processors. The time to access memory depended on its location. A processor could access memory local to it fastest. Memory attached to processors within the same cluster was accessed at a second, slower access time. Finally, memory accessed between clusters was accessed at a third, slowest access time.

A more modern example of a NUMA machine is the BBN Butterfly[7]. Processors in the Butterfly communicate by means of a "Butterfly Switch," the complexity of which increases only logarithmically with the number of processors. The Butterfly avoids cache consistency problems by providing no per-processor cache. Remote memory access times are roughly 10 times greater than local memory access times.

Although NUMA designs allow multiprocessors larger than UMA designs to be built, it is possible to build still larger multiprocessors based on NORMA designs. NORMAs are the easiest type of multiprocessor to build and, in a sense, have existed for as long as local area networking has been available.

NORMA multiprocessors are distinguished by the fact that a processor can not directly access the memory of another processor. The typical NORMA consists of a number of processors interconnected by some network topology. For example, the Intel HyperCube uses a multi-level cube structure with 10MHz Ethernet as the connecting technology. Slow interconnections between processors usually results is less tightly coupled processors, hence the lack of hardware support for remote memory access.

UMA, NUMA and NORMA categories represent trade-offs in multiprocessor designs. The ability to easily increase the number of processors in a single machine is generally gained at the expense of remote communication. For example, large NORMA multiprocessors can be built with ease, but offer the loosest coupling. UMA multiprocessors offer very tight coupling, but are difficult to build in large configurations. NUMA multiprocessors lie between NORMA and UMA machine in terms of remote communication speeds. Some data points that represent the level of coupling are current, commercial systems of each type. There is the Hypercube, in which communication times are measured in the hundreds of microseconds, versus five microseconds for the Butterfly and considerably less than one microsecond (on average) for a Multimax (see figure 8.2).

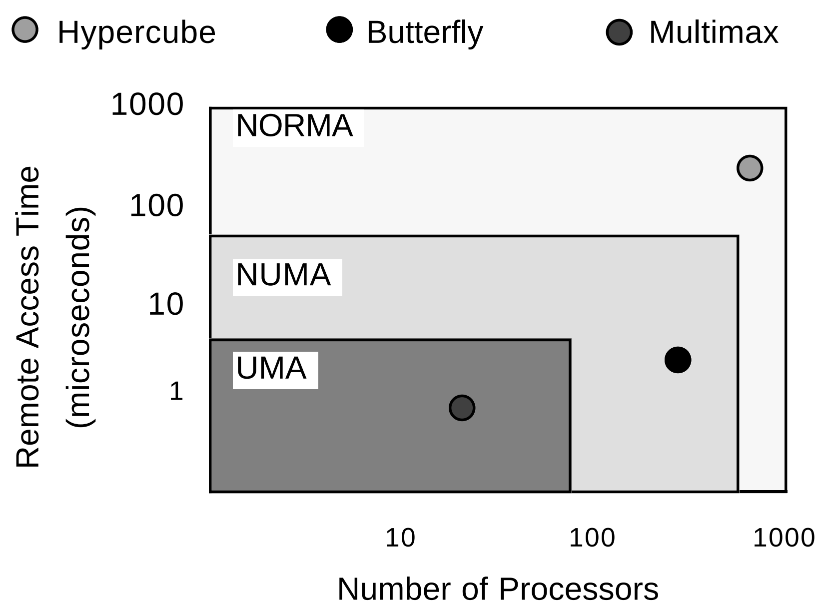

Figure 8.2: Trade-offs in UMA, NUMA and NORMA architectures

8.3 NUMA Architectures: Friend or Foe?

Non-uniform memory access machines attempt to increase the number of processors in a machine by using an interconnect (e.g., a communication switch), that can scale better than the interconnect that may be present in a uniform memory access machine (e.g., a bus). The net result is a machine in which processors may access all memory in the machine, but at a cost that is non-uniform, depending on the memory location.

At first glance, NUMA machines appear to be very attractive multiprocessor architectures: they offer both large numbers of processors and memory sharing. Without question, the large numbers of processors is an attractive feature, but the remote memory access delay inherent in a NUMA machine raises serious questions about the utility of such machines. In particular, an application that makes extensive remote data references is likely to suffer from a large performance degradation when compared to running the same application with local data references. Further, data that is shared by tasks on many processors may result in memory hot spots - degrading the performance of a communication switch to that of a bus.

Essentially, NUMA architectures are good because shared memory based applications can be easily built. But, NUMA architectures are bad because shared memory applications will not necessarily run efficiently on such architectures. Several solutions to this problem are possible:

- Force the application designer to tailor his application to a specific NUMA machine.
- Provide a set of tools that will allow an application designer to create an application the runs efficiently on a NUMA without necessarily becoming intimately familiar with any particular architecture.
- Allow the application designer to program as though the machine was a UMA and depend on operating system support for efficient execution of the application.

Forcing the application designer to tailor his application to a specific NUMA machine may, in many cases, be the only feasible means by which an application can run optimally on that machine. For those applications where the highest possible performance is of paramount importance, this is perhaps the best solution. However, using such programming practices is extremely very shortsighted in that the application may not easily be able to take advantage of future hardware architectures without complete redesign. Even worse, the application may not even run efficiently on slightly differing NUMA architectures.

Clearly, applications which do not depend on architecture specific knowledge will be more likely to run efficiently on a wider range of architectures. Tools that allow an application to automatically partition itself in such a way as to execute efficiently on a particular architecture can be valuable in achieving architecture independence.

However, from the point of view of providing an efficient virtual memory system on NUMA architectures, user level tools provide no help. Since it is advantageous to allow the user to think in terms of programming a UMA, the ideal solution is for the kernel to provide the illusion of a UMA machine even though the hardware may be a NUMA.

8.4 The NUMA layer

In a UMA machine, the virtual memory implementation is structured such that machine-independent code does all of the virtual memory management. Occasionally, the machine-independent code contacts the pmap module in order to update hardware page maps. The model of virtual memory used by the machine-independent code is that of a UMA — just the illusion that is desired. However, the machine-independent code has absolutely no knowledge of the actual underlying memory architecture. It simply manages some number of physical pages in the system.

In the case of a UMA, the pages (resident page structures) that are managed by the machine-independent code have a one to one correspondence to physical memory pages present in the hardware. However, there are no actual dependencies on this one to one relationship in any part of the implementation. This key fact makes possible the introduction of the *NUMA layer*. The NUMA layer exists between the machine-independent implementation and the pmap module. In effect, the NUMA layer exports the pmap interface, but instead of performing normal pmap operations, it first translates *logical* pages to *physical* pages (see figure 8.3).

In effect, the machine-independent code manages logical pages (although it doesn't realize that the pages are only logical). The NUMA layer is responsible for the logical to physical page translation. The NUMA layer, which is architecture specific, manages the logical to physical translations in whatever way is appropriate for the architecture or expected job mix. For example, a NUMA layer could have a policy in which all read-only data is replicated in the local memories on each processor referencing the data. In a case like this, the logical to physical page translation would be a one-to-many relationship. Operations affecting the logical page would have to be invoked for each physical page resulting is several actual pmap operations. Interestingly enough, the NUMA layer for a UMA machine can be thought of as the one-to-one mapping of logical to physical pages - the simplest possible NUMA layer.

8.5 Architecture Specific Software

The challenges to implementing virtual memory systems alluded to in this chapter resurface at the user application level. There are, at the highest level, several types of environments in which an application designer operates:

- the traditional, uniprocessor environment,
- the multiprocessor environments, which can further be subdivided into UMA, NUMA and NORMA based environments, and

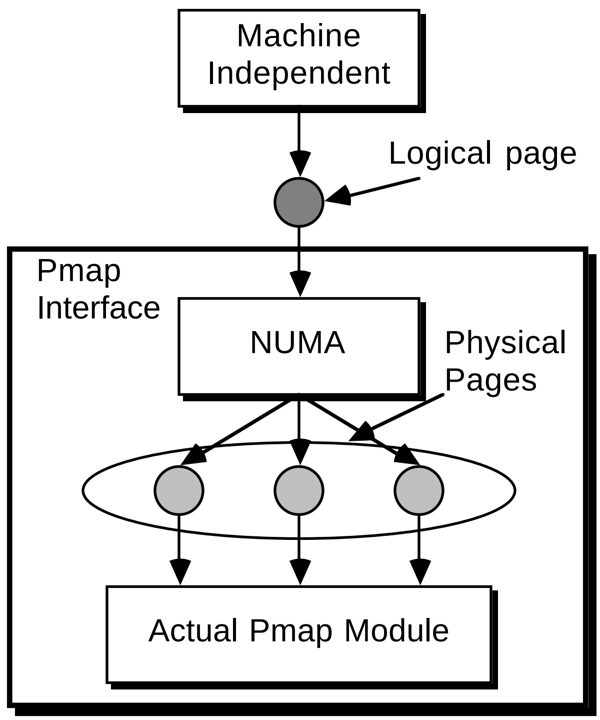

Figure 8.3: The NUMA layer

- the distributed environment.

The uniprocessor environment represents the simplest environment to work in. This is the environment that most programmers have been exposed to and feel comfortable with. The multiprocessor and distributed environments are much more difficult to target applications for. A simple method for creating applications in these environments is to use communication based mechanisms, such as remote procedure call (RPC), to allow the distribution of an application[37]. While this method allows an application to be distributed within a loosely coupled distributed system (such as a LAN), it also works within tightly coupled multiprocessor systems. Unfortunately, a communication based mechanism such as RPC can cause unnecessary inefficiencies in certain applications, especially highly parallel, tightly coupled applications designed to run on tightly coupled UMA architectures. For such applications, use of shared memory can be essential to reasonable performance. Likewise, certain applications can be decomposed so as to run on NUMA or NORMA architectures efficiently, exploiting the various trade-offs that have been made in those hardware designs.

From a virtual memory perspective, dealing with the shared memory programming anomalies that arise in programming NUMA and NORMA systems is very difficult. The problem is especially acute in the case of NUMA architectures. In this class of architectures, there are potentially several good solutions, but each depends on particular hardware designs. In NORMA systems, the high remote communication costs indicate that it is best to consider a NORMA machine to be a very efficient distributed system. NUMA systems, with a much lower cost for remote communication, tempt the user into treating them as the less complicated UMA systems. Treating the processors in a NUMA as if they were part of a large distributed system without shared memory intuitively strikes system builders as a waste of computing power. Nevertheless, NUMA designs will not always run UMA applications efficiently.

For these reasons, the application designer may find it necessary to give up the portability associated with the Mach virtual memory system in order to tune an algorithm for a specific hardware configuration. The cost of the lack of portability versus the gain in performance is something that must be measured on a per-application and per-architecture basis.

Chapter 9

Related Work

This chapter examines other approaches to operating system support for virtual memory. Several virtual memory systems, both research and commercial, are described. Each is compared and contrasted with the Mach design.

9.1 Historical Overview

The first virtual memory system was the Atlas, developed at the University of Manchester in England in the early 1960's[47]. This was the first system that provided access to more memory than was physically available. In the mid-1960s, Multics attempted to use virtual memory to provide advanced functionality such as mapped files. Multics was a large, complicated system. As a result, researchers at Bell Labs began work on Unix, a much simpler system offering no virtual memory support.

Since that time, many virtual memory systems have been built. Most, such as IBM's MVS[4] and VM[32] operating systems and DEC's VMS[18] operating system offer little virtual memory functionality. Systems such as Accent[41], Pilot[42] and more recent versions of Unix attempt to provide more aggressive virtual memory support.

9.2 UNIX Systems

UNIX systems have generally provided little in the form of virtual memory support. Early versions of UNIX, running on PDP-11 hardware, provided support for *swapping*. The size of each process was bounded by the amount of memory in the machine (or the 64K addressing limit of the PDP-11, whichever was smaller). When the total amount of memory used by multiple processes exceeded the amount of memory available, the system swapped entire processes' address spaces onto secondary storage. When a swapped process was later run, it was first swapped back into memory, potentially causing other processes to be swapped out[43,6].

As the software environments in which UNIX was used began to mature, typical UNIX programs began to grow in size. Further, developments in hardware technology made it more attractive to support demand paging in UNIX, the first version of this being 4.0 BSD[5]. Most current versions of UNIX support demand paging. In particular, recent releases of System V UNIX, along with many BSD derived versions of UNIX, all support virtual memory.

UNIX systems have been undergoing changes in virtual memory designs and implementations. For example, the mapped file extensions to 4.2 BSD were proposed as early as 1982[28]. Since then, similar mapped file interfaces have been implemented by several vendors, both as part of 4.2 BSD UNIX (e.g., Sequent Dynix[19]) and as part of a System V modified to contain 4.2 BSD enhancements (e.g., IBM's AIX[52]). A shared memory facility not based on mapped files is available in AT&T's System V[6] and has also been adapted to a variety of 4.2 BSD based systems such as DEC's Ultrix[14].

9.2.1 Virtual Memory in 4.3 BSD

Virtual memory management in 4.3 BSD is very closely tied to the VAX architecture upon which it is implemented. Accordingly, for each process in the system there exists a set of page tables. The page tables are always kept complete and represent the kernel's knowledge of virtual memory.

VAX page tables consist of two regions, allowing address space growth in two directions (see Section 7.2.1). First, stacks can grow from address 0x80000000 to lower addresses by increasing the size of the P1 region. Second, the data segment of a UNIX process can expand upward toward address 0x4000000 by increasing the size of the P0 region (see figure 9.1).

For each region, there is a *base* and *length* register. These two registers determine the location (and size) of an array of page table entries (PTEs). Each PTE describes a page of virtual memory — 512 bytes on the VAX. When a page contains a valid mapping, the corresponding PTE points to the appropriate physical page. The protection bits are set to allow whatever protection is applicable (read-only for text, read/write for data and stack). The valid bit is turned on to enable hardware access to the memory. When a page does not contain a valid mapping, and it should be paged in on reference, the valid bit is turned off. In this case, the PTE contains the disk address corresponding to the data. When the hardware generates a page fault, the fault handler consults the PTE for the disk address and invokes an I/O operation to retrieve the data. In a similar fashion, memory that is to be zero filled on demand is specified by a PTE with the valid bit turned off. An additional bit in the PTE indicates to the fault handler that the page should be zero filled when a fault is detected.

The 4.3 virtual memory system is, for the most part, a bare bones virtual memory management system. There is no support for copy-on-write operations, general file mapping or other advanced

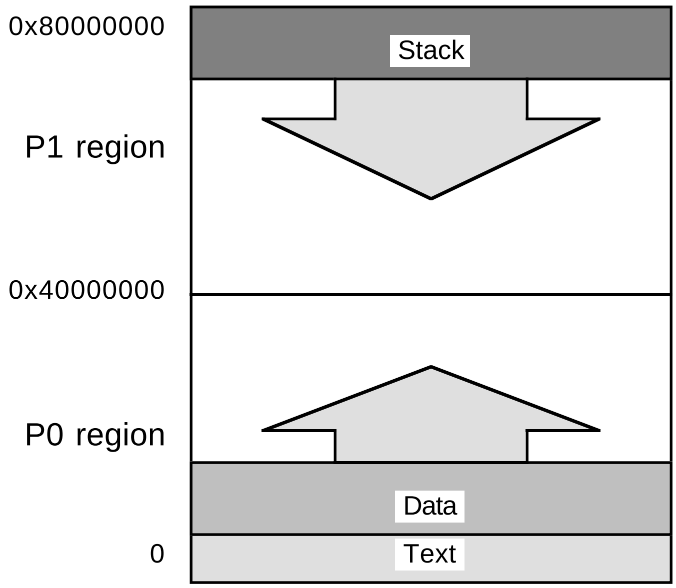

Figure 9.1: Vax address space.

functionality. However, in order to reduce some of the memory use in a system, 4.3 supports a notion of shared text segments (also supported in earlier UNIX systems). All invocations of the same program are allowed to share that program's text segment. This feature is implemented by a special *text table* which keeps track of the text segments in use. When a program is invoked and its text segment is already in memory, the page table entries corresponding to that text segment are used. If its text segment is not in memory, new page table entries are created.

4.3 provides support for both paging and swapping. It uses a clock based page replacement algorithm. One "clock hand" sweeps physical memory, turning off reference bits (since the VAX has no reference bits, they are simulated by invalidating the page). A second clock hand follows behind the first, looking for pages without a reference bit turned on. When such pages are found, they are paged-out. When the system paging rate becomes very high due to a heavy memory load, the system runs the *swapper* process. The swapper process forcibly swaps entire processes out of memory — an efficient mechanism for paging out large amounts of memory at once. The swapper is used to reduce the amount of page thrashing that may occur on a heavily loaded system.

Even though the 4.3 implementation is closely tied to the VAX architecture, it is often used as the basis for implementation on non-VAX hardware. However, vendors are increasingly recognizing many of the deficiencies in the BSD memory management system. The net result is that many vendors are replacing their 4.3 BSD Vax based virtual memory implementations with their own, new implementations. Tektronix, Sun and Gould are recent examples[48,23,2].

9.2.2 Virtual Memory in System V

The virtual memory implementation provided by System V is based on memory regions. These regions correspond to system objects that can be mapped into memory: text, data, stack or shared memory. Each region contains page tables and some amount of physical memory (corresponding to those pages in primary memory). Each process in the system has a region table which lists those regions that are mapped into the process.

System V implements copy-on-write by creating a new region. The page table entries of the original region are copied into the new region and all PTEs have a copy-on-write bit that is then set. When a write fault occurs on such a page, the page is copied and the PTE is updated to point to the new page. The copy-on-write bit is then turned off.

Since objects are represented as regions, and regions can be mapped into multiple address maps (region tables), System V has no need for specialized mechanisms such as the BSD text table. Instead, text sharing is accomplished by mapping the region corresponding to the text segment into multiple region tables. Shared memory is implemented in a similar fashion. Each shared memory segment in the system is represented by a region. When a process attaches to a shared memory segment, the region is added to the region table and page table entries are copied.

Like 4.3, System V provides both a pager and a swapper. In fact, the overall paging and swapping strategies of 4.3 and System V are similar. The System V pager is implemented by using a clock based algorithm; however, it provides for page aging — a feature not present in 4.3. Page aging is implemented by incrementing the age of each page passed over by the clock hand. When a page's age reaches some implementation-dependent constant it becomes eligible for page-out.

9.2.3 Comparing 4.3 BSD, System V and Mach Virtual Memory

Of the three virtual memory implementations (4.3, System V, Mach) 4.3 was the first. It is therefore not surprising that it is the least functional, both internally and externally. 4.3 provides only a minimal user interface to virtual memory (*fork*, *exec*, *brk*) and is closely tied to the VAX architecture.

The System V implementation is significantly more machine-independent than 4.3. It also provides many features not available in 4.3. System V provides a copy-on-write implementation of *fork*, obviating the need for the 4.3 *vfork* primitive. System V also provides access to shared memory.

Mach is still more machine-independent than System V, offering a formal separation between machine-independent and machine-dependent modules. All of the features provided by System V are provided by Mach using much more general mechanisms. Mach provides a much higher level of functionality than both 4.3 and System V in many respects. For example, Mach allows any operation to be performed on any page and for multiple operations to be combined. This results in use of virtual memory that is far less restrictive than the region manipulation provided by System V. The Mach implementation can easily support the System V virtual memory interface while the System V implementation cannot support the Mach interface.

Although System V does provide copy-on-write, it does not carry lazy evaluation as far as Mach. In particular, System V does not perform lazy evaluation of page tables. Since the System V implementation depends on full page tables, it is not appropriate for applications with sparsely used large address spaces.

Neither 4.3 nor System V addresses the issues of virtual memory in a distributed or parallel environment. The primitives of Mach were carefully chosen to avoid problems in such environments (such as burdening the kernel with maintaining network coherent shared memory). 4.3 successfully avoids many such problems due to the simple fact that it exports very little functionality. However, System V ignores the ramifications of its shared memory primitives in a network environment.

The trend toward large memory systems, a primary concern of Mach, creates many problems for 4.3 and System V. Perhaps one of the most serious shortcomings of both 4.3 and System V, related to large memory machines, is the mechanism used to allocate swap storage. Whereas Mach employs lazy evaluation to postpone allocation of any swap storage, 4.3 and System V allocate that storage when new processes (or regions) are created. This has serious ramifications for machines with large memories and little swap space. A machine with more physical memory than swap space cannot use all of its memory since there is no place for swap allocation to occur.

Both 4.3 and System V attempt to prevent page thrashing by forcibly swapping processes under heavy load. The Mach virtual memory system does not attempt to perform any such swapping. However, the Mach kernel implementation contains a swapper thread which tries to provide similar functionality. In particular, for each thread in the system there exists a kernel stack. For normal execution, the kernel stack must be wired down (cannot be paged). The Mach swapper, the *swapout* thread, periodically locates threads that have been idle for sometime and makes their corresponding kernel stacks pageable. When such a thread is run at a later time, it first has its kernel stack wired down by another kernel thread, the Mach *swapin thread*. The Mach swapper does not forcibly page out memory; hence, under periods of heavy load, Mach is more susceptible to thrashing. The Mach swapper is not a part of the virtual memory system. Instead, it is an application that uses some of the functionality provided by the virtual memory implementation.

9.3 VAX/VMS

The VAX/VMS Operating System, like 4.3 BSD UNIX, contains a virtual memory implementation that is closely tied to the VAX architecture[29]. The important data structures used to implement virtual memory in VAX/VMS are:

- *process specific data structures*: The process specific data structures contain the P0 and P1 page tables and working set information.
- *PFN database*: The PFN database contains information about physical pages in the system (physical page frames).
- *global page data structures*: Separate data structures maintain information about global pages such as a global page table.

Page tables account for all mapping information in VMS. As in 4.3, page table entries take on different formats depending on the data they represent. PTEs contain information representing valid pages, demand zero pages, pages in transition, global pages and pages in the paging file or image file.

Sharing is provided by allowing processes to map *global sections*, hence global pages, into their address space. Per-process PTEs are set to point to global information corresponding to global pages. This information is used at page fault time to generate a valid PTE, corresponding to the global memory, for the process.

Many of the operations performed by VMS are based on copy-on-reference. Each PTE contains a copy-on-reference bit. When this bit is set on a page and a fault occurs on that page, a new physical page is allocated. The data referenced by the old page is copied into the new page and the new page is placed into the PTE (with the copy-on-reference bit turned off).

VMS supports both paging and swapping. The page replacement algorithm is based on the working set model. Each process has a working set limit — the maximum number of pages that can be used by the process. When the working set limit is reached, pages are removed from the process' address space in a FIFO manner. This type of algorithm prevents a single, large process from consuming all of a machine's resources and starving other processes. The VMS swapper, like the swappers of 4.3 and System V, is used to prevent thrashing under high memory loads.

Most of the comparison of Mach and the two UNIX implementations applies to a comparison of Mach and VMS. In particular, VMS lacks the machine independence of Mach. Although VMS does provide copy-on-reference as a form of lazy evaluation, it does not take advantage of lazy evaluation techniques to the same degree as Mach. Finally, VMS does not address the issues of providing support for parallel and distributed applications. Although it is possible to use shared memory in VMS, like System V, the ramifications of its shared memory primitives in a distributed environment have been ignored.

9.4 Other Virtual Memory Systems

While many systems provide only simple primitives for allocating and deallocating virtual memory, some systems provide more advanced virtual memory functions. Systems such as Accent and Multics provide the ability to create segments within a virtual address space that correspond to files or other permanent data. Accent also provides powerful mechanisms for transferring large amounts of virtual memory between protected address spaces in an efficient manner in much the same way as Mach.

Although Accent and Multics provided advanced functionality, they lacked Mach's flexibility in providing different paging mechanisms. The segments provided by Accent and Multics were system features deeply entrenched in the virtual memory implementation. The same functionality in Mach is a simple extension of existing primitives, using well defined interfaces.

Obvious parallels can be drawn between Accent and Mach. However, Accent lacked the ability to support tightly-coupled parallel computation. Perhaps more importantly, the implementation of virtual memory in Accent was tied very closely to the Perq hardware[39] on which it was implemented. Unable to cope with the large variety of architectures, even those that were only uniprocessor based, Accent eventually ceased to be a viable operating system.

Multics, often cited as the "excuse" for the creation of UNIX, due to its complexity, was much like Mach and Accent in that it supported advanced use of virtual memory. Nevertheless, Multics, like Accent, could never gain wide acceptance as its design was hopelessly tied to specific hardware support in complex ways. When Multics was designed, the idea of a portable operating system was largely foreign to system designers. It is therefore not surprising that the system was so closely tied to a specific hardware base.

Several other systems are worth noting:

- *Aegis*: Aegis is a distributed system that runs on Apollo workstations[30]. Aegis is similar to Mach in that it has a notion of objects which can be mapped into a process' address space. Unlike Mach, Aegis does not support parallel processing. Aegis also differs from Mach in that its objects may exist anywhere in a distributed system — and no data coherency is provided by the kernel when different processors utilize the same object. Mach avoids this pitfall by imposing upon an external pager to define the sharing semantics.

- *Dynix*: Dynix is a multiprocessor version of UNIX that runs on the Sequent Balance and Symmetry series machines[19]. This version of UNIX supports parallel processing by providing shared memory via the proposed Berkeley 4.2 mechanism.

- *Pilot*: Pilot was an operating system designed for a personal computer[42]. It provided a single user, single language[35], single address space environment. The address space in Pilot was structured as a collection of *spaces*, each space corresponding to a contiguous range of pages. Backing storage for spaces was represented by files and served as the basic unit of swapping.

- *Sprite*: Sprite is a distributed system under development at the University of California at Berkeley[38]. Sprite provides virtual memory similar to UNIX. Sharing occurs by processes sharing entire address spaces. This type of sharing is roughly equivalent to the multiple thread per task mechanism provided by Mach. Sprite attempts to solve the problems of sharing and communication with the use of files instead of virtual memory, as is the case with Mach.

- *System/38*: System/38 is an operating system developed at IBM to run on System/38 machines[22]. This system represents an extreme in which virtual memory is used as a basis for a completely object oriented system. A System/38 installation has a single, global address space in which all processes run. All objects in the system are represented by virtual memory somewhere in the virtual address space. For adequate performance, System/38 depends heavily on its special purpose hardware.

- *Umax*: Umax is a multiprocessor version of UNIX that runs on the Encore Multimax[20]. This version of UNIX provides an inheritance mechanism for shared memory similar to Mach.

Some of these systems address issues of virtual memory in a distributed environment while others address these issues in a parallel environment. None address issues that arise in both distributed and parallel systems. Unlike Mach, the more advanced virtual memory systems are closely tied to the hardware upon which they are intended to be used.

Chapter 10

Summary and Conclusions

10.1 Summary

This thesis has described the design, implementation and evaluation of the Mach virtual memory system. The virtual memory system

- is easily portable among 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 many different hardware architectures, 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. 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, 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 virtually arbitrary mapping relations to be created. Shared memory and support for file mapping (with Unix emulation) have been provided.

Many functions performed by the virtual memory system have traditionally been performed only by the operating system kernel. The Mach virtual memory system is structured so as to allow many of the functions to be performed by user tasks.

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

10.2 Data Structure and Algorithm Summary

The data structures and algorithms used in the Mach virtual memory system were carefully chosen to address the following issues:

- *Architecture Independence*: The data structures reflect reasonable assumptions about how virtual memory is used - not how hardware represents it.
- *Address Space Management*: Large, sparse address spaces are efficiently represented.
- *Multiprocessor Support*: All data structures and algorithms are structured to allow parallel access. Algorithms guarantee that deadlock conditions are impossible.
- *Modular Interfaces*: All interfaces between the virtual memory system and other operating system components are well defined.
- *Lazy Evaluation*: Lazy evaluation techniques are used whenever possible to increase the performance of the system.

There are four basic data structures used in the implementation:

- *resident page table*: A table used to keep track of information about machine-independent pages.
- *address map*: A doubly linked list of address map entries. Each address map entry corresponds to a region of virtual memory. A hint is used to speed lookup operations.
- *memory object*: The basic unit of backing storage managed by a pager.
- *pmap*: The abstract type corresponding to the hardware's representation of an address map.

Key algorithms manipulate these data structures. Address map algorithms manipulate the list of address map entries, including splitting and merging entries. Object based algorithms maintain reference counts on objects and implement the object cache. The fault handler, the hub of the virtual memory system, verifies access to pages, transparently implements copy-on-write and invokes the pmap module for page validation.

10.3 Modular Design

The Mach virtual memory implementation interfaces with the rest of the kernel using well defined interfaces. In addition, subcomponents of the implementation have well defined interfaces. The two most important such interfaces are the pager interface and the pmap interface.

The pager interface provides a formal mechanism by which clients may implement pagein and pageout operations on memory. This manifests itself in the form of the inode pager, used by the system to map files as well as perform normal backing storage for pages which are paged out. A complete generalization of this interface is the external pager interface, described in [53].

The pmap interface (routines implemented by the pmap module) is used to manipulate an architecture's MMU. The clean split between the machine-independent memory management modules and the pmap module has resulted in a system that is easily portable to paged architectures.

10.4 Future Impact

Future virtual memory research will continue with external pagers as well as sophisticated page replacement algorithms. One can also expect future research to study the feasibility of running Mach virtual memory on future hardware architectures, especially loosely coupled multiprocessors. Clearly, implementation of the proposed NUMA layer is a logical choice for future research.

Other research, only loosely associated with this thesis, is likely to include putting Mach virtual memory to use. For example, many existing applications assume a restricted model of virtual memory. Most Unix utilities assume they have only a small, limited address space available for use, and use temporary files as a replacement for virtual memory. One should expect extensive research into the effects of rewriting these applications to use the Mach virtual memory model.

Another area in which this thesis will have impact is in the design of hardware for virtual memory systems. In particular, there are two logical paths for hardware and software designers to follow pertaining to the support of virtual memory. The first path, already followed by some, is the simplest possible MMU — a TLB only. The second path could be considered the exact opposite of the TLB-only path: a complex MMU that implements the pmap module directly. Whichever path is taken by hardware designers, it is hoped that lessons are learned from the experience gained during this thesis research.

10.5 Conclusion

This dissertation has demonstrated that it is possible to build an architecture-independent virtual memory system. The system can provide support for multiprocessor architectures and advanced functionality such as shared memory and mapped files. Further, all of these features can be provided without a negative impact on performance. Choosing data structures and algorithms that match the expected use of virtual memory (rather than those that match virtual memory hardware) can yield a system that performs better than systems targeted at specific architectures.

Appendix A

Mach Virtual Memory Operations

This appendix describes the virtual memory operations that are exported to the user. These primitives collectively form the Mach Virtual Memory Interface. The primitives are divided into four separate functional groups: address space manipulation, protection, inheritance and miscellaneous.

A.1 Address Space Manipulation

A.1.1 Allocate virtual memory: vm_allocate

```
kern_return_t vm_allocate(target_task, address, size, anywhere)
        task_t           target_task;
        vm_address_t     *address;            /* in/out */
        vm_size_t        size;
        boolean_t        anywhere;
```

Vm_allocate allocates a region of virtual memory, placing it in the specified task's address space. If the *anywhere* flag is TRUE, then the kernel determines where to allocate the memory and returns the address in *address*. If the *anywhere* flag is FALSE, the kernel attempts to allocate memory at the address passed in by the user. The physical memory is not actually allocated until the new virtual memory is referenced. By default, the kernel rounds all addresses down to the nearest page boundary and all memory sizes up to the nearest page size. The global variable *vm_page_size* contains the page size. *Task_self_* returns the value of the current task port which should be used as the *target_task* argument in order to allocate memory in the caller's address space. For languages other than C, these values can be obtained by the calls *vm_statistics* and *task_self*. Initially, the pages of allocated memory will be protected to allow all forms of access, and will be inherited in child tasks as a copy. Subsequent calls to *vm_protect* and *vm_inherit* may be used to change these properties. The allocated region is always zero-filled.

A.1.2 Deallocate virtual memory: vm_deallocate

```
kern_return_t vm_deallocate(target_task, address, size)
        task_t          target_task;
        vm_address_t    address;
        vm_size_t       size;
```

Vm_deallocate relinquishes access to a region of a task's address space, causing further access to that memory to fail. This address range will be available for reallocation. Note that because of the rounding to virtual page boundaries, more than *size* bytes may be deallocated. *Vm_page_size* or *vm_statistics* can be used to discover the virtual page size.

This call may be used to deallocate memory passed to a task in a message (via out of line data). In this case, the rounding should cause no trouble, since the region of memory is allocated as a set of pages.

The *vm_deallocate* call affects only the task specified by the *target_task*. Other tasks which may have access to this memory may continue to reference it.

A.2 Protection

A.2.1 Protect virtual memory: vm_protect

```
kern_return_t vm_protect(target_task, address, size,
                         set_maximum, new_protection)
        task_t          target_task;
        vm_address_t    address;
        vm_size_t       size;
        boolean_t       set_maximum;
        vm_prot_t       new_protection;
```

Vm_protect changes the protection of some pages of allocated memory in a task's address space. Memory specified by the *address* and it size is affected. If the *set_maximum* flag is TRUE, then the maximum protection value is changed, if the flag is FALSE, the current protection value is changed. If the maximum protection is reduced below the current protection, both will be changed to reflect the new maximum. Protection values, specified by the *new_protection* parameter can be any combination of *VM_PROT_READ*, *VM_PROT_WRITE* and *VM_PROT_EXECUTE*. When memory is first allocated it has all protection bits enabled. The exact interpretation of a protection value is machine-dependent. For example, machines on which Mach is currently implemented only provide three levels of memory protection: no access, read and execute access, and read, execute and write access. On this class of architectures, *VM_PROT_WRITE* allows read, execute and write access, *VM_PROT_READ* or *VM_PROT_EXECUTE* allows read and execute access, but not write access.

A.3 Inheritance

A.3.1 Specifying inheritance: vm_inherit

```
kern_return_t vm_inherit(target_task, address, size,
                                        new_inheritance)
        task_t           target_task;
        vm_address_t     address;
        vm_size_t        size;
        vm_inherit_t     new_inheritance;
```

Vm_inherit specifies how a region of a task's address space is to be passed to child tasks at the time of task creation. The region to be affected is specified by the *address* and *size* parameters, which take on a new inheritance value of *new_inheritance*. The new inheritance may be either *VM_INHERIT_SHARE*, which indicates shared memory, *VM_INHERIT_COPY*, which indicates copied memory (copy-on-write) or *VM_INHERIT_NONE*, which indicates no transfer of memory.

A.4 Miscellaneous

A.4.1 Reading virtual memory: vm_read

```
kern_return_t vm_read(target_task, address, size, data,
                                     data_count)
        task_t           target_task;
        vm_address_t     address;
        vm_size_t        size;
        vm_address_t     *data;               /* out */
        vm_size_t        *data_count;       /* out */
```

Vm_read allows one task's virtual memory to be read by another task. Data specified by *address* and *size* in the *target_task* are read. The result is returned in newly allocated memory, specified by *data* and *data_count*. Since this memory is newly allocated, the caller should use *vm_deallocate* to free the memory when it is done with the data.

A.4.2 Writing virtual memory: vm_write

```
kern_return_t vm_write(target_task, address, data, data_count)
        task_t          target_task;
        vm_address_t    address;
        vm_address_t    data;
        vm_size_t       data_count;
```

Vm_write allows a task's virtual memory to be written by another task. Specifically, data in the caller's address space, specified by *data* and *data_count* is written into the *target_task* at *address*.

A.4.3 Copying virtual memory: vm_copy

```
kern_return_t vm_copy(target_task, source_address, count,
                                   dest_address)
        task_t           target_task;
        vm_address_t     source_address;
        vm_size_t        count;
        vm_address_t     dest_address;
```

Vm_copy causes the source memory range (starting at *source_address*) to be copied to the destination address (*dest_address*). The size of the range to be copied is specified by *count*. The destination address range must already be allocated and writable; the source range must be readable.

A.4.4 Region information: vm_region

```
kern_return_t  vm_region(target_task, start, size,
protection, max_protection,
inheritance, shared,
object_name, object_offset)
        task_t          target_task;
        vm_offset_t     *start;                 /* in/out */
        vm_size_t       *size;                  /* out */
        vm_prot_t       *protection;            /* out */
```

```
        vm_prot_t       *max_protection;        /* out */
        vm_inherit_t    *inheritance;           /* out */
        boolean_t       *shared;                /* out */
        port_t          *object_name;           /* out */
        vm_offset_t     *offset_in_object;
```

Vm_region returns a description of the specified region of the target task's virtual address space. The actual start and size of a region with the same properties (and containing *start*) are returned in *start* and *size*. If the specified start does not correspond to a region of allocated memory, the region returned corresponds to the next region in the address space with a higher starting address. This feature may be used to scan an address map, determining region information for the entire map. The properties associated with that region are returned in *protection*, *max_protection*, *inheritance*, and *shared*. Information specific to the object mapped is returned in *object_name* and *object_offset*.

A.4.5 Querying statistics information: vm_statistics

```
struct vm_statistics {
        long    pagesize;           /* page size in bytes */
        long    free_count;         /* # of pages free */
        long    active_count;       /* # of pages active */
        long    inactive_count;     /* # of pages inactive */
        long    wire_count;         /* # of pages wired down */
        long    zero_fill_count;    /* # of zero fill pages */
        long    reactivations;      /* # of pages reactivated */
        long    pageins;            /* # of pageins */
        long    pageouts;           /* # of pageouts */
        long    faults;             /* # of faults */
        long    cow_faults;         /* # of copy-on-writes */
        long    lookups;            /* object cache lookups */
        long    hits;               /* object cache hits */
};

typedef struct vm_statistics        vm_statistics_data_t;

kern_return_t       vm_statistics(target_task, vm_stats)
        task_t                      target_task;
        vm_statistics_data_t        *vm_stats;      /* out */
```

Vm_statistics returns the statistics about the kernel's use of virtual memory since the kernel was booted. The page size can also be found as a global variable *vm_page_size* which is set at task initialization and remains constant for the life of the task.

Appendix B

The Pmap Interface

This appendix describes the interface to the pmap module. The pmap module is the machine-dependent portion of the virtual memory implementation. It is responsible for the maintenance of any hardware related data structures and controls access to the MMU.

B.1 Pmap Handle Maintenance

B.1.1 Create new pmap: pmap_create

```
pmap_t pmap_create(size)
        vm_size_t    size;
```

Create and return a pmap. If the *size* specified for the map is zero, the map is an actual physical map and may be referenced by the hardware. If the *size* specified is non-zero, the map will be used in software only and is bounded by that size.

B.1.2 Gain reference to a pmap: pmap_reference

```
pmap_reference(pmap)
        pmap_t       pmap;
```

Add a reference to the specified pmap. This is used to indicate that the pmap is in use by multiple machine-independent maps and will prevent early deallocation of the pmap.

B.1.3 Destroy reference to a pmap: pmap_destroy

```
pmap_destroy(pmap)
        pmap_t       pmap;
```

Remove a reference from the specified pmap. If the reference count goes to zero, the pmap is destroyed and any resources it may have used are freed. Pmap_destroy is only called on a pmap with no active references.

B.2 Range Operations

B.2.1 Enter mappings: pmap_enter

```
pmap_enter(pmap, v, p, prot, wired)
        pmap_t                pmap;
        vm_offset_t           v;
        register vm_offset_t  pa;
        vm_prot_t             prot;
        boolean_t             wired;
```

Insert the given physical page (*p*) at the specified virtual address (*v*) in the specified physical map with the protection (*prot*) requested. If specified by *wired*, the page will be wired down meaning no machine-independent faults may occur on the mapping.

B.2.2 Remove mappings: pmap_remove

```
pmap_remove(pmap, s, e)
        pmap_t       pmap;
        vm_offset_t  s, e;
```

Remove the specified range of addresses from the specified map. The start and end addresses (*s* and *e*) are guaranteed to be rounded to page boundaries.

B.2.3 Change protections: pmap_protect

```
pmap_protect(pmap, s, e, prot)
        pmap_t       pmap;
        vm_offset_t  s, e;
        vm_prot_t    prot;
```

Set the protection on the specified range of this map. The start and end are specified by *s* and *e*. The desired protection is specified by *prot*. Like *pmap_remove*, the start and end are guaranteed to be rounded to page boundaries.

B.3 Global Operations

B.3.1 Invalidate: pmap_remove_all

```
pmap_remove_all(phys)
        vm_offset_t    phys;
```

Remove the specified physical page from all maps in which it resides. This routine also reflects modify bits back to the machine-independent data structures.

B.3.2 Copy-on-write: pmap_copy_on_write

```
pmap_copy_on_write(phys)
        vm_offset_t    phys;
```

Remove write privileges from all physical maps for the physical page specified by *phys*.

B.4 Physical Memory

B.4.1 Zero a physical page: pmap_zero_page

```
pmap_zero_page(phys)
        vm_offset_t    phys;
```

Zero the specified physical page.

B.4.2 Copy a physical page: pmap_copy_page

```
pmap_copy_page(src, dest)
        vm_offset_t    src, dest;
```

Make physical page *dest* be a copy of physical page *src*.

B.5 Use Information

B.5.1 Use map: pmap_activate

```
pmap_activate(pmap, th, cpu)
        pmap_t      pmap;
        thread_t    th;
        int         cpu;
```

Signifies that the specified pmap and thread are now active on the specified cpu.

B.5.2 No longer use map: pmap_deactivate

```
pmap_deactivate(pmap, th, cpu)
        pmap_t      pmap;
        thread_t    th;
        int         cpu;
```

Signifies that the specified pmap and thread are no longer active on the specified cpu.

B.6 Miscellaneous

B.6.1 Synchronize: pmap_update

```
pmap_update()
```

Make sure MMU data structures are up to date with information passed to pmap module. This is used to guarantee that operations being delayed by the pmap module are performed when necessary.

B.6.2 Copy: pmap_copy

```
pmap_copy(dst_pmap, src_pmap, dst_addr, len, src_addr)
        pmap_t          dst_pmap;
        pmap_t          src_pmap;
        vm_offset_t     dst_addr;
        vm_size_t       len;
        vm_offset_t     src_addr;
```

The range specified by *src_addr* and *len* has been virtual copied from the source pmap (*src_pmap*) to the range specified by *dst_addr* and *len* in the destination pmap (*dst_pmap*). This routine is only advisory and need not do anything (the same information is provided by other routines such as *pmap_copy_on_write* and *pmap_protect)*.

B.6.3 Set pageability: pmap_pageable

```
pmap_pageable(pmap, start, end, pageable)
        pmap_t          pmap;
        vm_offset_t     start;
        vm_offset_t     end;
        boolean_t       pageable;
```

Make the pages specified by *start* and *end* pageable (or not) as specified by *pageable*. Like *pmap_copy*, this routine is only advisory and need not do anything.

Appendix C

The Lock Primitives

This appendix describes the locking primitives used to guarantee proper multiprocessor functionality of the virtual memory implementation. There are two types of locks used: *simple* locks and *complex* locks. Simple locks are generally faster than complex locks but do not offer as much functionality.

C.1 Simple Locks

C.1.1 Initialize: simple_lock_init

```
simple_lock_init(l)
        simple_lock_t    l;
```

Initialize the lock *l* to an unlocked state.

C.1.2 Lock/Unlock: simple_lock, simple_unlock, and simple_unlock_try

```
simple_lock(l)
        simple_lock_t    l;
```

Gain the lock specified by *l*. The processor spins until the lock is acquired. Once holding the lock, no other processor may acquire the lock.

```
simple_unlock(l)
        simple_lock_t    l;
```

Release the lock specified by *l*. Other processors waiting for the lock may now acquire it.

```
boolean_t simple_lock_try(l)
        simple_lock_t    l;
```

Attempt to gain the lock specified by *l*, but do not spin if the lock is in use by another processor. The return value specifies whether or not the lock was acquired.

C.2 Complex Locks

C.2.1 Initialize: lock_init

```
lock_init(l, can_sleep)
        lock_t     l;
        boolean_t  can_sleep;
```

Initialize the lock *l*, setting it to an unlocked state. The *can_sleep* flag specifies whether or not a sleep operation should be invoked if an attempt to acquire the lock is made when another processor holds the lock.

C.2.2 Set sleepable: lock_sleepable

```
lock_sleepable(l, can_sleep)
        lock_t     l;
        boolean_t  can_sleep;
```

Set whether or not the lock *l* should use sleep/wakeup operations for synchronization (rather than spin loops) based on *can_sleep*.

C.2.3 Set recursion: lock_recursive

```
lock_recursive(l, can_recurse)
        lock_t     l;
        boolean_t  can_recurse;
```

Set (or clear) the flag that determines whether or not a lock may be acquired in a recursive fashion. A recursive lock may be acquired many times by the same thread.

C.2.4 Lock/Unlock: lock_write and lock_done

```
lock_write(l)
       lock_t      l;
```

Acquire a write lock on lock *l*. Once acquired, no other thread may acquire the lock. If another thread already has the lock, the thread may sleep, depending on the previous calls to *lock_init* and *lock_sleepable*.

```
lock_done(l)
       lock_t      l;
```

Relinquish access to lock *l*.

C.2.5 Read locks: lock_read and lock_read_to_write

```
lock_read(l)
       lock_t      l;
```

Gain a read lock on lock *l*. Read locks may be held by any number of threads simultaneously. As is the case with *lock_write*, this call may sleep in order to wait for another thread to relinquish the lock.

```
boolean_t lock_read_to_write(l)
       lock_t      l;
```

Upgrades a read lock to a write lock. The caller must already hold a read lock. If another reader has already requested an upgrade to a write lock, the request fails. In this case, TRUE is returned and no lock is held.

Appendix D

The Zone Primitives

This appendix describes the zone primitives. The zone primitives are used for fast dynamic allocation of internal kernel data structures.

D.1 Zone Operations

D.1.1 Initialize a zone: zinit

```
zone_t zinit(size, max, alloc, pageable, name)
        vm_size_t    size;
        vm_size_t    max;
        vm_size_t    alloc;
        boolean_t    pageable;
        char         *name;
```

Create and initialize a new zone. The size of each element in the zone is specified by *size*. The maximum size of the zone is *max*, and when more memory is needed to fill the zone, allocations occur *alloc* bytes at a time. The *pageable* flag determines whether or not the memory can be pageable. The *name* parameter provides a textual name that can be used for debugging purposes.

D.1.2 Force memory into a zone: zcram

```
zcram(zone, newmem, size)
        zone_t       zone;
        vm_offset_t  newmem;
        vm_size_t    size;
```

Puts the memory specified by *newmem* and *size* into the specified zone. This is useful to prevent recursive dependencies of the zone package and virtual memory system at boot time.

D.1.3 Allocate zone elements: zalloc and zget

```
vm_offset_t zalloc(zone)
        zone_t       zone;

vm_offset_t zget(zone)
        zone_t       zone;
```

Returns an element from the specified zone. In the normal case, this is a simple list operation. If the zone is empty, more expensive memory allocations take place. The *zalloc* operation may block if it is necessary to wait for the paging system to free pages. The *zget* operation will never block, however, it may not return an element if the list is temporarily free.

D.1.4 Free zone elements: zfree

```
vm_offset_t zfree(zone, elem)
        zone_t       zone;
        vm_offset_t  elem;
```

Places the specified element (*elem*) onto the free list for the specified zone (*zone*).

Bibliography

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: a new kernel foundation for UNIX development. In *Proceedings of Summer Usenix*, July 1986.

[2] Pervaze Akhtar. A replacement for Berkeley memory management. In *Proceedings of Summer Usenix*, July 1987.

[3] James Q. Arnold. Shared libraries on UNIX System V. In *Proceedings of Summer Usenix*, June 1986.

[4] M. A. Auslander, D. C. Larkin, and A. L. Scherr. The evolution of the MVS operating system. *IBM Journal of Research and Development*, 25(5):471–482, 1981.

[5] O. Babaoglu and W. Joy. Converting a swap-based system to do paging in an architecture lacking page-referenced bits. In *Proc. 8th Symposium on Operating Systems Principles*, pages 78–86, ACM, December 1981.

[6] Maurice J. Bach. *The Design of the Unix Operating System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[7] BBN Laboratories. *Butterfly Parallel Processor Overview*. Technical Report, BBN Computer Company, Cambridge, MA, June 1985.

[8] R. Bisiani and A. Forin. Architectural support for multilanguage parallel programming on heterogenous systems. In *Proceedings of the 2nd Symposium on Architectural Support for Programming Languages and Operating Systems*, October 1987.

[9] F.J. Corbato. *A Paging Experiment with the Multics System*. Project MAC Report MAC-M-384, MIT, May 1968.

[10] P.J. Denning. Virtual memory. *Computing Surveys*, 2(3):153–189, September 1970.

[11] P.J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, May 1968.

[12] P.J. Denning. Working sets past and present. *IEEE Transactions on Software Engineering*, 6(1):64–84, January 1980.

[13] P.J. Denning and S.C. Schwartz. Properties of the working set model. *Communications of the ACM*, 15(3):191–198, March 1972.

[14] *Ultrix Programmer's Manual*. Digital Equipment Corporation, 1986.

[15] *KA630-AA CPU Module User's Guide*. Digital Equipment Corporation, 1986.

[16] *VAX Architecture Reference Manual*. Digital Equipment Corporation, 1983.

[17] *VAX Hardware Reference Manual*. Digital Equipment Corporation, 1983.

[18] *VAX/VMS System Services Reference Manual*. Digital Equipment Corporation, 1985. AA-Z501B-TE.

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

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

[21] R. Fitzgerald and R. F. Rashid. The integratation of virtual memory management and interprocess communication in Accent. *ACM Transactions on Computer Systems*, 4(2), May 1986.

[22] R.E. French, R.W. Collins, and L.W. Loen. System/38 machine storage management. *IBM System/38 Technical Developments, IBM General Systems Division*, pages 63–66, 1978.

[23] Robert A. Gingell, Joseph P. Moran, and William A. Shannon. Virtual memory architecture in SunOS. In *Proceedings of Summer Usenix*, July 1987.

[24] *80386 Hardware Reference Manual*. Intel, Inc., 1986.

[25] *Introduction to the 80386*. Intel, Inc., April 1986.

[26] A. K. Jones, R. J. Chansler, I. E. Durham, K. Schwans, and S. Vegdahl. StarOS, a multiprocessor operating system for the support of task forces. In *Proceedings of the 7th Symposium on Operating System Principles*, pages 117–129, ACM, December 1979.

[27] Michael B. Jones, Richard F. Rashid, and Mary R. Thompson. Matchmaker: an interface specification language for distributed processing. In *Proceedings of the Twelfth Annual Symposium on Principles of Programming Languages*, pages 225–235, ACM, January 1985.

[28] W. Joy et. al. *4.2BSD System Manual*. Technical Report , Computer Systems Research Group, Computer Science Division, University of California, Berkeley, July 1983.

[29] Lawrence J. Kenah, Ruith Goldenberg, and Simon F. Bate. *VMS Internals and Data Structures: Version 4.4*. Digital Press, Maynard, Massachusetts, 1987.

[30] 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 Journal on Selected Areas in Communications*, SAC-1(5):842–857, November 1983.

[31] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. In *5th Symposium on Principles of Distributed Computing*, 1986.

[32] S. E. Madnick and J. J. Donovan. Virtual machine/370 (vm/370). *Operating Systems*, pages 549–563, 1974.

[33] *MC68030 Technical Summary*. Motorola, Inc., 1986.

[34] David B. McDonald, Scott E. Fahlman, and Alfred Z. Spector. *An Efficient Common Lisp for the IBM RT PC*. Technical Report, Carnegie Mellon University, July 1987.

[35] J. G. Mitchell, W. Maybury, and R. Sweet. *Mesa Language Manual*. Xerox Palo Alto Research Center, 1979.

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

[37] Bruce Jay Nelson. *Remote Procedure Call*. PhD thesis, Carnegie Mellon University, May 1981.

[38] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite network file system. In *Symposium on Operating Systems Principles*, November 1987.

[39] *Perq System Overview*. Perq Systems Corporation, Pittsburgh, Pennsylvania, March 1984.

[40] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *Proceedings of the 2nd Symposium on Architectural Support for Programming Languages and Operating Systems*, October 1987.

[41] R. F. Rashid and G. Robertson. Accent: a communication oriented network operating system kernel. In *Proc. 8th Symposium on Operating Systems Principles*, pages 64–75, December 1981.

[42] David D. Redell, Yogen K. Dalal, Thomas R. Horsley, Hugh C. Lauer, William C. Lynch, Paul R. McJones, Hal G. Murray, and Stephen C. Purcell. Pilot: an operating system for a personal computer. *Communications of the ACM*, 23(2):81–92, February 1980.

[43] D.M. Ritchie and K. Thompson. The Unix time-sharing system. *CACM*, 17(7):365–375, July 1974.

[44] Robert D. Sansom. *Building a Secure Distributed Computer System*. PhD thesis, Carnegie Mellon University, November 1987.

[45] Alfred Z. Spector, Dan Duchamp, Jeffrey L. Eppinger, Sherri G. Menees, and Dean S. Thompson. The Camelot interface specification. September 1986. Camelot Working Memo 2.

[46] Alfred Z. Spector and Kathryn R. Swedlow, editors. *The Guide to the Camelot Distributed Transaction Facility: Release 1*, Carnegie Mellon University, Pittsburgh, Pennsylvania, February 1987.

[47] F. H. Sumner, G. Haley, and E. C. Y. Chen. The central control unit of the Atlas computer. In *Proceedings of the IFIP Congress*, pages 657–662, 1962.

[48] Edward W. Sznyter, Patrick Clancy, and James Crossland. A new virtual-memory implementation for Unix. In *Proceedings of Summer Usenix*, July 1986.

[49] A. Tevanian, R. Rashid, D. Golub, D. Black, E. Cooper, and M. Young. Mach threads and the UNIX kernel: the battle for control. In *Proceedings of Summer Usenix*, June 1987.

[50] 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 *Proceedings of Summer Usenix*, June 1987.

[51] Avadis. Tevanian, Jr. Mach: a new basis for future UNIX development. In *Proceedings of Autumn EUUG*, September 1986.

[52] F. Waters, editor. *IBM RT Personal Computer Technology*, International Business Machines Corporation, 1986.

[53] 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 *Symposium on Operating Systems Principles*, November 1987.