Real Memory Mach

Philippe Bernadat and David Black
Open Software Foundation

Real Memory Mach [(1)]

Philippe Bernadat[(2)]
David L. Black[(3)]

Abstract

Mach was designed with the assumption that the underlying hardware includes a page-based memory management unit, enabling the use of a large, sparse virtual address space. Therefore the Mach micro-kernel in its current state cannot be used on segmented machines such as Cray vector supercomputers or real memory machines such as transputers. This paper describes the results of initial work to adapt the Mach micro-kernel to such architectures, including the architectural changes and results obtained from a prototype system.

1 Introduction

Mach's virtual memory system is based on the abstraction of a memory object[7]. A memory object represents a single source of memory (e.g., an executable file), and task address spaces are assembled by mapping portions of memory objects into the address space. For example, a traditional process consists of the text and data portions of one object plus additional objects for uninitialized (bss) memory and the stack. Mach's communication (IPC) system allows portions of memory objects to be included in messages for efficient transmission of large amounts of data[6]. Communication and interaction among tasks is also facilitated by the ability to share memory objects among tasks. The implementation of Mach's virtual memory (VM) system, including memory objects, makes extensive use of page-based functionality to support techniques such as lazy evaluation and copy on write. As a consequence, the existing Mach micro-kernel cannot be easily adapted to machines that do not support page-based memory management; these include machines that only support segments, and machines with no memory management support whatsoever.

Memory management hardware that supports pages allows virtual addresses to be mapped to arbitrary physical addresses on a page by page basis. Protection (read/write) and mapping validity can also be set on a page by page basis; a memory access for which a valid mapping does not exist or that exceeds the protections allowed by its mapping causes the processor to take a trap. The operating system is then responsible for providing a valid translation with sufficient protection, or signalling an exception to the application. Operating system implementations of virtual memory functionality (including Mach's) depend on the resulting ability to modify virtual to physical memory mappings and protections without the application's knowledge.

Page based memory management support is not universally available. Although virtually all workstations and minicomputers support this functionality, other systems do not. For some high performance processors, the impact of translation logic on hardware cycle time may be unacceptable (e.g., Cray systems[8]). Another reason is that the impact of translation on processor complexity may be unacceptable (e.g., transputers[3]). These hardware driven design decisions are often complemented by system usage patterns that place less demand on memory management functionality (e.g., dedicated use, swapping instead of paging); the resulting system software can be closely tied to its hardware base. We refer to such hardware as *real memory systems*.

---

1. This work was supported in part by a grant from Cray Research, Inc.
2. Philippe Bernadat, OSF Research Institute, Grenoble, France, bernadat@gr.osf.org
3. David L. Black, OSF Research Institute, Cambridge, MA, USA, dlb@osf.org

Software that does not depend on large sparse virtual address spaces may not need sophisticated memory management support. Much of the Unix interface does not require the existence of virtual memory. Implementations are another matter; both OSF/1[4] and OSF/1 MK (the Mach 3 based single server version of OSF/1[1][2]) are designed for platforms that support page-based memory management. OSF/1 MK should be more amenable to execution on hardware without virtual memory support because its server uses Mach kernel interface (as opposed to accessing internal kernel routines) and has little dependence on the existence of virtual memory. Our results bear this out; only a few minor configuration changes were needed to adapt the OSF/1 MK server to our prototype implementation of Mach for segmented hardware without virtual memory support.

This paper is organized as follows:

- Segmented memory model
- Objectives and Approach
- Mach VM & IPC architecture description
- Micro kernel architectural changes
- Server modifications
- Prototype Status
- Performance
- Future work

2 Segmented Memory Model

An abstract segment model forms the basis of our work to adapt Mach's memory management implementation for execution on hardware that lacks virtual memory support. Depending on the specific architecture, these segments can be of fixed or variable size, and may or may not have alignment constraints. We use the term *virtual segment* to refer to a segment in a task's address space, and the term *physical segment* to refer to a contiguous region of physical memory that may be accessed via a virtual segment. These notions of segment replace the corresponding notions of virtual and physical pages in communication between the machine independent and machine dependent (pmap) portions of the memory management code. The pmap module for the resulting real memory system is responsible for mapping, unmapping, and protecting segments instead of individual pages. The correspondence with hardware features is obvious for segmented hardware, but this model is also applicable to hardware that has no memory management support due to its (desirable) property of managing physical memory in contiguous chunks.

The model delineates virtual segments by their segment number and length; physical segments are correspondingly identified by a base physical address and length. We assume that the segment number of a virtual segment is implicit in addresses supplied to Mach's kernel VM primitives (e.g., vm_allocate()). Use of the most significant bits to encode the segment number is a common technique. This model is also applicable to hardware that specifies the segment number in a different fashion for actual memory accesses; in this case it is necessary to define a composite address for use in kernel primitives that includes the segment number (encoding in the high bits is one possible technique). Examples of hardware for which this is necessary include that in which the segment number is implicit in the type of access (e.g., instruction vs. date) or is explicitly specified in the instruction (e.g., by selection of segment register).

We have made minimal conservative assumptions about the capabilities of the hardware for this work. No assumption is made about whether accesses to invalid data can be continued or restarted (they are fatal on some segmented systems). As a result, we have had to remove all lazy evaluation from Mach's memory management code. Hardware that can restart or continue such accesses may be able to take advantage of techniques such as load on reference, or copy on write, on a segment basis. Similarly, we make no assumptions about the hardware's ability to enforce read, write and execute protections. In fact, we take advantage of this to limit segment usage. For example, on hardware that cannot enforce protections, all protections are equivalent and a new segment need never be created for protection reasons.

An overall goal of this work has been to consider virtual segments as a scarce resource, and manage them accordingly. For example, Cray Research machines have exactly two segments, a text segment and a data segment. Since these systems implement stacks as software abstractions in the data segment, this is equivalent to three segments on a machine that supports stacks in hardware. Our prototype implementation has not been optimized for architectures where a task could map an arbitrary number of segments with arbitrary protections.

3 Objectives and Approach

The goal of this work has been to modify Mach's memory management system to execute on real memory hardware with minimal impact on the existing kernel interface. In order to address the fundamental architecture and design issues without getting distracted by the details involved in a complete port to a new system, we have built a prototype system that emulates segments using a conventional memory management unit. The prototype implementation was intended to address the memory management architecture issues and kernel interface changes. Additional work on the prototype would be necessary to define formal segment layers and machine dependent interfaces. The architecture described in this paper occasionally corresponds to the ideal goal rather than the actual prototype; we have been careful to make this distinction clear when it arises.

The required kernel interface modifications are minor, and usually consist of subsetting to cope with the limited capabilities of real memory hardware. For example, external memory managers are still supported (e.g., to obtain contents of executed and mapped files), but the kernel will return an error if a manager attempts to change the protection of an individual page. Another example of a hardware limit is that every explicitly mapped object (e.g., a mapped file) consumes one virtual segment; attempts to use more segments than the hardware supports are also rejected by the kernel.

Our prototype kernel has a number of limitations. We have not addressed swapping of segments or management of swap space. We have also made no attempt to run our modified Mach kernel on actual segmented hardware. This kernel also does not support multiprocessors, although there are no fundamental design obstacles in extending it to do so. This study has only considered variable sized segments without alignment constraints. The ideal design would split segment management into machine independent and machine dependent portions to avoid the appearance of segment constraints in machine independent code.

The prototype has reached the point where a port to segmented hardware would be reasonable to undertake (i.e., there are no fundamental architectural or design issues standing in the way of such a port). We have been pleasantly surprised by the small amount of work required to run the OSF/1 MK server on the prototype kernel, and in turn have gained the advantage of not needing to develop a specialized server to test the prototype kernel or run standard benchmarks.

4 Mach Internal Virtual Memory Architecture

In order to understand the required architectural changes, we consider the internal layers of the Mach VM architecture (see Figure 1).

4.1 Machine independent layer

Tasks contain threads, a virtual address space and an ipc space for port rights. The virtual address space is described by the ordered collection of map entries (vm_map_entry) in a virtual memory map (vm_map). Each map entry defines the mapping of a portion of a memory object (vm_object) to a range of the address space, including protection and inheritance attributes. Each memory object is a collection of untyped data organized into pages (vm_page). Pages within an object are identified by their offset from the start of the object, hence a map entry can be considered as a mapping from a portion of the task's address space to a portion of the object's offset space. The kernel manages physical memory as a cache of memory object data; the actual data

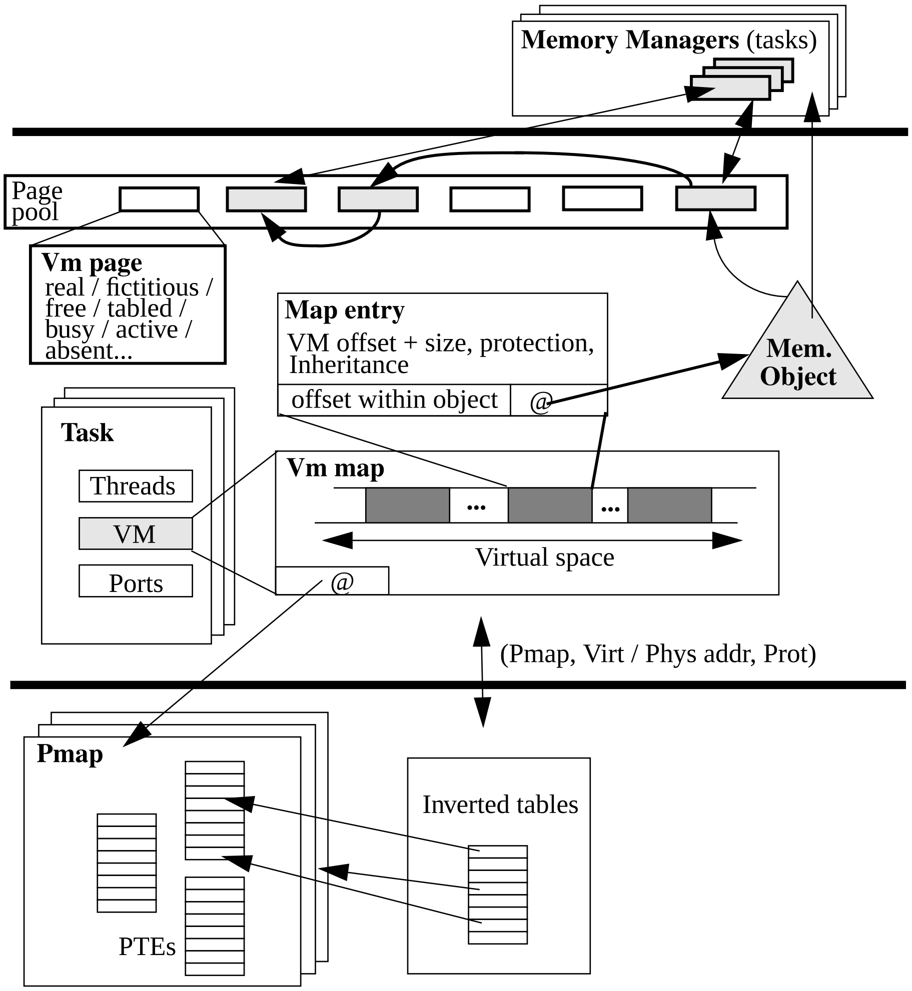

figure 1: Mach VM Architecture

is managed by an external memory manager or the kernel's default memory manager.

Memory managers are tasks like any other. The kernel requests memory object pages from the related memory manager at page fault time, and sends the data back to the memory manager on request from the manager or at pageout time. Memory object pages correspond to physical pages when the data is in memory. Fictitious pages (no corresponding physical page) are used when the data is not present, but has been requested from the pager. Anonymous memory, created as the result of a vm_allocate() call or copy operation, is managed by the default memory manager. This manager also acts as the pager of last resort when external managers do not free pages as requested by the kernel. The default memory manager's memory is wired down as it cannot page itself.

Memory objects can be shared by multiple tasks or mapped more than once within the same task. When a task sends data to another task, or when a task inherits memory from another task without sharing it, the memory is (logically) copied into a new object. To improve performance, the VM system avoids copying the object's data by using shadow objects. A shadow object contains only the pages modified since the copy, with the original object providing the unmodified pages. Pages in the shadow object are said to shadow (i.e., cast a shadow on) the corresponding pages in the original object that they replace; a light shown from above would illuminate only those pages visible to the task.

4.2 Machine dependent VM layer

All the information required to find the memory page corresponding to a virtual address is in the machine independent layer; no machine-dependent data structures are used. Once this information is located, it is necessary to instruct the hardware that a virtual address in this task must be mapped to a specific physical page with appropriate protections. This is achieved via a simple interface to the physical map (pmap) layer. The pmap layer is responsible for managing the memory management unit, including installation, modification, and removal of virtual to physical translations. The pmap layer must also be able to tell the machine independent layer if a given physical page is mapped or if it has been modified. As a consequence, pmap modules for hardware that employs a tree-structured or linear page table must also manage an inverted list to enable it to locate all virtual mappings of a physical page.

4.3 IPC

Mach's IPC system is relevant to our work because it uses the VM system when large amounts of data are sent in a message. Mach IPC messages are typed, providing information to the kernel that allows it to determine if the message contains either of the following:

- Port rights: Messages are the only way to transfer these kernel managed entities among tasks.
- Out of line memory (passed by reference).

When sending data via messages, a task can either send it *in-line* or *out-of-line* (ool). All data sent in a message has copy semantics; the receiver appears to get its own copy. For in-line data, this is implemented by copying the data into the message body at send time and copying it from the message body at receive time. Out-of-line data is passed by reference, and the kernel employs lazy-evaluation and copy-on-write techniques to minimize the amount of data that must be copied to implement copy semantics.

The VM layer provides an abstraction called a map copy that the IPC layer can manipulate. A map copy is a temporary repository for data being sent via a message. It is created when the message is sent and destroyed when the message is received. The map copy abstraction is a logical copy of the virtual region passed in the message at the time the message was sent.

There are three different possible representations of a memory region in a map copy:

- A vm object
- A list of pages
- A list of vm_map_entries

The object representation is used for pageout. When the kernel sends a page to a pager, the page is moved to a new object to allow it to be backed by the default pager, thus protecting the kernel from a recalcitrant pager that refuses to perform pageouts. The new object contains a single page, and it is more efficient than using map entries for this case.

The list of pages is used for pagein and device operations. It allows the kernel to move pages directly from the original object if the sender deallocates the data at send time. For most pageins and device reads, the sent data is deallocated once sent by the pager or device handler.

The map_entry type is the most common. When a thread sends a region of virtual memory in a message, the kernel copies the map entries and mark the object(s) as copy on write. To facilitate the implementation, the VM layer splits and clips together vm_map entries to match the region. At receive time these map entries are moved into the receiver's map, providing the receiver with copies (shadows) of the original objects (see Figure 2).

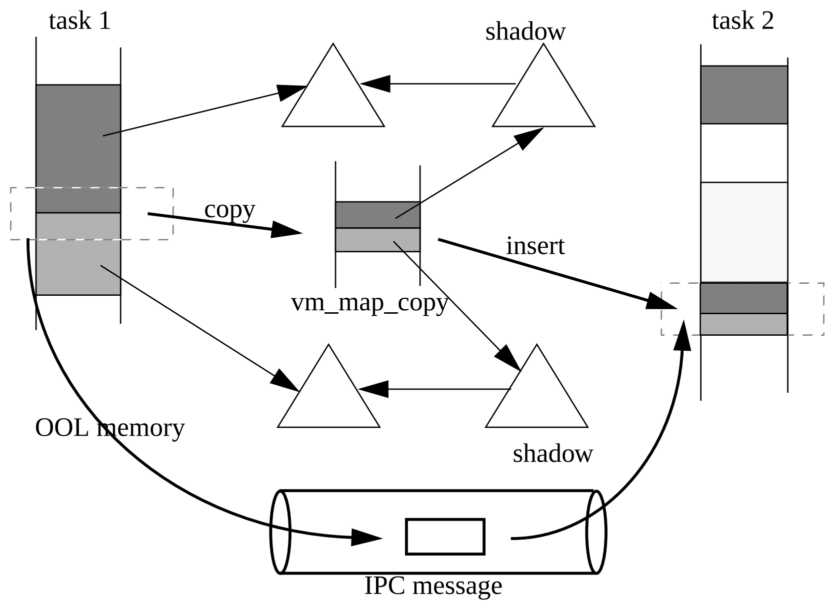

figure 2: Sending OOL Memory Through Mach IPC

5 Architectural changes

This section describes the architectural changes made to Mach's virtual memory management system to support real memory hardware.

5.1 Object / Segment / Page relationship

An object contains a linked queue of referenced (present or being requested) pages. For the real memory system, the pages of an object must be contiguous and in existence (it is not possible to use lazy evaluation to fault them in). Thus an object represents a single physical segment. A virtual segment in a task's address space is a different concept; it may be mapped to an entire physical segment, or only a portion thereof. In turn, physical segments may be mapped multiple times into the address spaces of multiple tasks.

The vm_page entity persists in the real memory design, as it is the unit of communication with external pagers and the IPC system. Every page of an object exists (unless the object is swapped), cannot be fictitious (i.e., it always corresponds to a physical page) and must be initialized before any access. The pages do not need to be linked into a queue as they are contiguous. The page abstraction is not used by and is not visible to machine dependent code.

Mach's original VM system makes very little use of object sizes. Sizes of externally managed objects are adjusted in response to page faults that appear to extend the object. This is unreasonable for a real memory system. Even if hardware permits restarting an access beyond the end of a segment, this would require expanding that segment for each new page. If the next sequential page is not free in physical memory, reorganization and compaction of physical memory are required to accommodate the newly enlarged segment. The real memory system initializes the object size at object creation time to avoid these inefficiencies.

Contiguous physical pages (i.e., a physical segment) must be allocated and initialized at memory allocation or mapping time. Because access faults may not be restartable, the real memory system cannot rely on the kernel to detect initial references to pages. Hence, memory allocation causes the resulting memory to be zero filled immediately. Similarly, memory mapping causes the mapped pages to be requested from the object's memory manager immediately. The algorithm and code responsible for requesting pages from the memory manager or zero filling pages are basically unchanged.

Lazy evaluation also cannot be used to optimize memory copying (e.g., copy on write) because the real memory system cannot assume that write references can be detected and responded to. As a consequence, shadow objects cannot be used. Copying an object involves creation of a new regular object and a new segment to hold the copy; then relevant memory is physically copied. When a task is created, the objects corresponding to memory inherited as copy are copied by the kernel instead of being shared copy-on-write.

5.2 Map entry / segment relationship

Virtual and physical segments do not correspond one to one. A physical segment can be mapped multiple times at different addresses within the same or different tasks. Consider a program with text and data. The memory object (a file) corresponds to a single physical segment in memory. But the text and data have different protections, thus requiring two virtual segments (Figure 3a). Two virtual segments are also required if text and data are contiguous in the file but separated in the task's address space (Figure 3b). On machines with a single segment per task, the text and data virtual addresses must correspond to the text and data offsets within the memory object, and there can be no distinction in protection at the hardware level (Figure 3c).

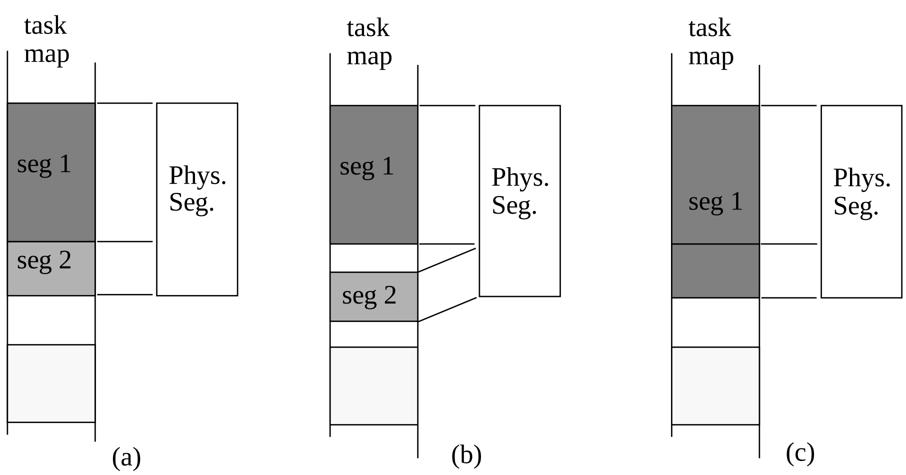

figure 3: Virtual and Physical Segments

In all three cases there are two map entries, even if they correspond to a single object and physical segment. The machine independent layer marks each entry with the appropriate protections, even if these protections are implemented identically by the hardware.

5.3 Anonymous memory management

The real memory system attempts to avoid creating new segments (both virtual and physical) when anonymous (zero fill) memory is allocated. This is in contrast to mapping an externally managed object which requires creation of the segments. Mach's original VM architecture creates a new object in the following cases:

- The virtual memory region is not contiguous to an existing one (user specified virtual address)
- Required protection/inheritance differ from the one of the existing object
- Object reference count is greater than one.

These conditions can be made less restrictive in the real memory system by employing lazy deallocation and overly eager allocation. This results in accessible valid memory at locations that would be invalid in the original system, but this is a small price to pay for the resulting saving of virtual segments.

If the virtual address at which a user wishes to allocate new memory is not contiguous with the last virtual address of an existing anonymous memory region, the same memory object can still be used in some cases (Figure 4). The middle pages will be allocated in the physical segment but not used. Accesses to these pages will not be trapped even though the memory region is invalid from the memory system’s point of view. The corresponding segment area could be uninitialized, but for security reasons, they should be zero filled (to be implemented in our prototype). The memory management system must prevent mapping of new objects inside this invalid memory region.

In the case where memory is allocated without specifying a fixed address for the new memory, the real memory system tries to merge the new region with an existing segment. The predefined read, write and execute protections of the new and existing region may not match. As an example the OSF/1 server sets the data region protection to r/w, but the default r/w/x protection is used when the kernel allocates memory within a task (e.g., for message passing). This prevents coalescing the newly allocated memory region with the data region. On most hardware, these 2 protections are identical. Hence, the real memory system introduces a machine dependent function to compare protections based on whether they are implemented identically (e.g., our prototype assumes that r/w/x is equivalent to r/w). In this case, a single physical and virtual segment suffice to implement the resulting protection.

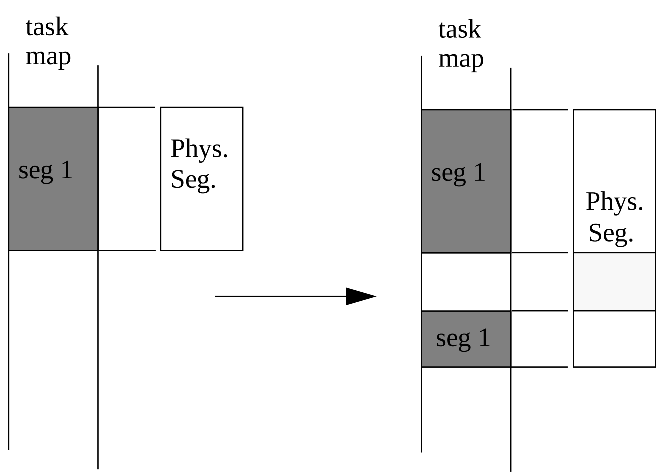

figure 4: Anonymous Memory Allocation

Different concerns apply if the existing region to be extended has an object with a reference count larger than one. This indicates that the object is mapped into multiple tasks or may also be mapped by multiple regions of the same task. Regions are often split when a task deallocates memory in the middle of an existing memory region. In this case it is possible to coalesce newly allocated memory with this object even if the object’s reference count is larger than one. Shadow objects no longer cause this situation, as they aren’t used in the real memory system. In order to determine if an object is actually shared, the physical (object) segment data structure contains two fields indicating the task in which the segment is mapped and where. Reserved values are used to indicate that the object is mapped by more than one task or at more than one virtual address in the same task. These fields are also used by the segment management layer.

At deallocation time, if a task specifies a memory region in the middle of an existing object, the VM layer registers it and modifies the map entries, but does not change the task’s segments. The deallocated region remains accessible to the application and may be reused to satisfy a subsequent memory allocation request.

5.4 IPC

The real memory implementation changes the use of map copy structures by the IPC system. The map entry lists of a map copy always refer to regular objects. Object copies (of the part sent in the message) are performed at message send time. On the other hand, at receive time, the kernel allocates anonymous memory in the receiver's memory space and copies the data to minimize segment usage; directly mapping the copied object(s) into the receiver's address space would require one or more additional virtual segments. The kernel often splits map entries to facilitate the creation of these map copies. This appeared to pose a problem in minimizing the number of virtual segments, so code was added to the prototype to be aggressive about merging map entries. This code is not necessary because one virtual segment can encompass multiple vm map entries, but yields performance benefits by reducing the number of map entries. Our prototype does not support the page list version of map copies, in order to avoid problems if the pages in the page list are discontiguous. This is not strictly necessary, as the pages in a page list usually come from a single segment (and hence are contiguous), and could be copied if this was not the case.

5.5 Segment management

The segment management layer is new; it replaces and changes the functionality of the resident page management layer. The segment management layer implements segment allocation, deallocation and growth. Segment allocation requires fitting the desired segment into an available space in memory (in contrast to page allocation, which simply removes a page from a queue). Care must be taken to minimize fragmentation of the system's physical memory. Inefficient management of segment growth can drastically affect performance. Segment growth must be contiguous to the original segment, and so may require moving the original segment or its neighbors to make room. These segment moves are expensive and should be minimized. The pageout queue management code has been deleted because pageout is not a meaningful concept in a real memory system; entire segments must be swapped out instead (our prototype does not implement this). Objects can be still be marked as cacheable; this causes their physical segments to be cached for reuse, as in the original system.

Figure 5 shows the data structures used for segment management. Memory is divided into some number of contiguous segments; the number and their sizes vary dynamically as the system executes (for clarity we assume that the system's physical memory is contiguous, but this is not a requirement). Each segment is either busy (in use) or free (available for use). An allocation operation looks for a free segment of at least the requested size; this segment is split if it is larger than needed. All segments are organized into a doubly linked list by physical address to facilitate segment merging. Free segments are hashed by size and linked into buckets; adjoining free segments are always merged into a single segment. Our prototype uses a hash function that assigns segment sizes between adjacent powers of 2 to the same bucket (i.e., bucket n contains segments whose size satisfies $2^n+1 \le size \le 2^{n+1}$). The first segment of the appropriate bucket is selected by an allocation operation. If the segment must be split, the remaining portion is placed at the end of the free list for its bucket; this increases the likelihood that this portion will be available if the allocated segment should need to expand in the future.

Each segment is represented by the following data structure:

```
struct seg {
      struct segment       *next_seg; *prev_seg;
      struct segment       *next_hash, *prev_hash;
      vm_offset_t          page_addr;
      vm_size_t            page_count;
      struct vm_page       *page;
      struct vm_object     *object;
```

```
        struct vm_map          *map;
        vm_offset_t            vaddr;
}
```

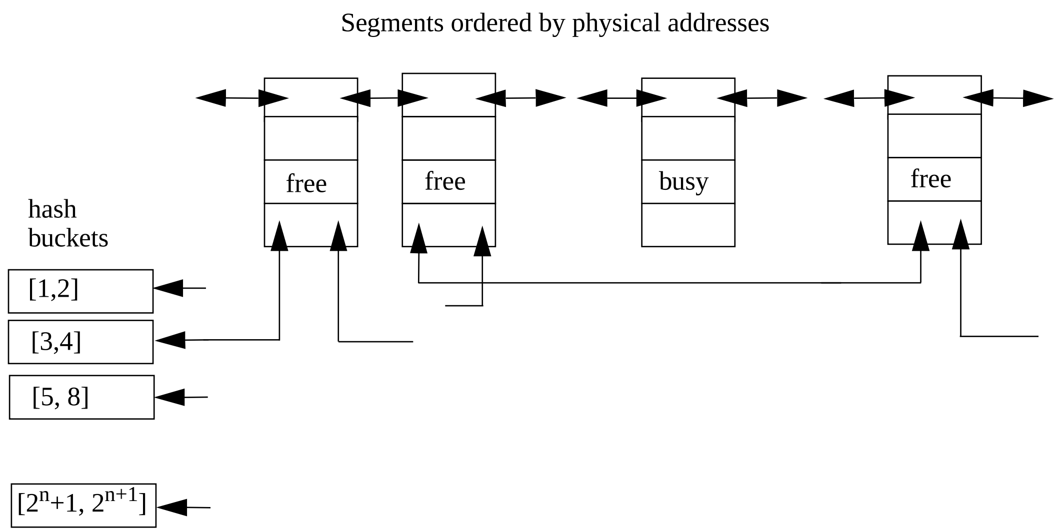

figure 5: Segment Management Data Structures

The vm_page field is used for communication with pagers and managing out-of-line memory sent via IPC; in order to save space in the segment structure, the page_addr argument could be obtained from the vm_page structure. The last three fields in the structure support segment growth. When moving a segment, the code checks that its vm object is not busy (e.g., pagein, pageout or device I/O in progress), and that the segment is not mapped multiple times. Once the segment is moved, it is remapped into the task that is using it. Additional locking constraints would be required for a multiprocessor implementation.

5.6 Management of the kernel task's space

We assume that the kernel runs in physical mode with the kernel's address space being the address space of physical memory. Multiple segments are not needed to protect the kernel from itself, and not all architectures of interest make multiple segments available for privileged mode execution. Running the kernel with physical addresses simplifies virtual space management because the physical segment addresses and the kernel virtual segment addresses match. This also makes it possible to move data directly from user-mode tasks to the kernel when desired because the physical address of the memory must be otherwise unused in the kernel's address space. Our prototype does not implement physical mode execution for the kernel; instead, we assume no limitations on the number of kernel virtual segments to simulate running in physical memory (segments are used for user-mode tasks).

A couple of minor modifications to the kernel's management of its address space were necessary. The kernel defines a special memory object, the kernel object, which is a repository for most of the data structures of the kernel that are not pageable. This object's size is initially zero, and is grown as needed. It is necessary to give this object a fixed size so that its segment can be managed like other physical segments. The prototype system does not implement automatic growth of the kernel object, although this is possible. The kernel may also define submaps that reserve portions of the kernel's address space. The primary use of these maps is to avoid deadlocks and related problems involved in paging and fault handling. Since our real memory system does not take page faults and swaps instead of pages, submaps were removed. In addition, reservation of address space in this fashion would reduce the kernel's available address space and prevent it from accessing all possible segments by their physical addresses.

5.7 Pmaps

We did not address the issues involved in modifying this layer for real memory hardware because our prototype continues to run in virtual mode. Instead of invoking the pmap module with physical page addresses, the final system would provide segment addresses and sizes (e.g., to pmap_enter() and pmap_remove()). The segment number must be part of the virtual segment address. The distinction between the kernel and other tasks in current pmap modules must be maintained because the pmap_enter() and pmap_remove() calls do not need to change any mapping information for the (physically addressed) kernel in a complete real memory system.

The prototype contains no support for enforcing hardware restrictions on segment size and alignment. Additional machine-dependent data structures would be needed along with an interface to access them when searching for available address space in a vm_map. This internal interface would have little impact on the relevant algorithms. A similar modification was used by Wheeler and Bershad to optimize mapping alignments for support of virtual caches [9].

6 Server modifications

Use of virtual memory in the OSF/1 MK system is a potential obstacle to moving the server and commands to a real memory system. The OSF/1 server is a large user mode task whose memory is subdivided into many regions. At the time of this work, the server allocated over 30 megabytes (MB) of virtual address space to itself using between 50 and 100 map entries. A straightforward transformation of this to our segmented model would imply between 50 and 100 segments, which is clearly excessive. Between the emulator[2] and the use of shared libraries, OSF/1 commands required at least 2MB of virtual space and used between 12 and 20 map entries. The usage of segments and virtual space must be reduced to match the hardware constraints of a real memory system. Our primary goal was to enable execution on systems with hardware support for three segments; text, data, and stack (this corresponds to two segments on systems that simulate the stack in software because the stack is then part of the data segment). This section describes the minimal modifications to the system that were necessary to achieve both this goal and a sizeable reduction in virtual memory usage; we were pleasantly surprised by the ease with which this was accomplished.

The first step was to reduce the server's configuration by:

- Reducing the number of users from 16 to 2 (thus changing NPROC from 168 to 36)
- Removing the streams subsystem, and the AFS and system V filesystems.
- Limiting the memory allocated to mbufs to 256 K instead of 1 Meg.
- Reducing the maximum buffer cache size to 256 K.

The server's DEFAULT configuration is built with a shared memory area between the emulator and the server. Each task (emulator) shares three regions with the server: two for partially mapping the u_area, (one is read only) and a one to exchange data between the emulator and the server. This improves performance by reducing the number of RPCs and the amount of memory passed in messages. In a real memory system, this would require 3 additional segments per task. We removed this memory sharing option from the configuration.

There were a number of other configuration changes. We eliminated the configuration option that allows the server to map in a kernel maintained time value for faster access, saving the virtual segment that would be required to implement this. We changed the default stack size for user programs to 64KB from 2MB to reduce memory usage. Shared libraries consume virtual segments on a real memory system (at least one per library); we used statically bound commands and libraries instead to avoid this usage.

These changes allow us to achieve our segment reduction goals. The resulting server has three virtual segments (text, data, and stack). The text and data are the same physical segment, but different virtual segments for protection reasons. The stack segment is used only in initially starting the server; the cthreads package within the server allocates its own stacks from the data segment. Commands and other applications now have six segments: text, data, stack and 3 segments for the emulator (text/data/bss). A version of the server that does not use an emulator is currently being implemented[5]; with such a server, each user task would only require 3 segments. The current version of this server maps arbitrary regions of memory from its tasks into its address space to optimize the performance of moving data to and from them (copyin(), copyout()). This server also has a configuration option that uses kernel calls (vm_read() and vm_write()) to perform this function; a real memory version of this server would be configured in this way.

Our prototype has reduced the server's use of virtual address space to 4MB. The OSF/1 server can create up to 50 cthreads. In single user mode, there are already 30 cthreads. Each cthread is created with a stack size of 64K, accounting for almost half (about 2MB) of the virtual space usage. This stack size could be reduced to 32K saving another megabyte if desired. Further reductions are complicated by the use of buffers allocated from the stack to send and receive messages whose size may be up to 8KB. Recent changes to MiG have partly alleviated this restriction, but are not incorporated into our prototype.

7 Prototype Status

Our prototype kernel executes on an IBM-compatible PC with 16 Mbytes of memory, a 300 MB disk and an ethernet connection. The kernel code is based on NORMA_MK11, but we have made no attempt to preserve NORMA functionality. In the memory management area, 11 files were modified and 2 new files were created. Modifications were also required to 13 other files in various areas of the kernel. Note that this prototype still executes in virtual mode, so there is very little change to i386-specific code. The vast majority of in-kernel interfaces are unchanged. Only the interfaces between the vm_object layer and the resident memory layer have been changed to deal with segments. Submaps have been removed. The kernel memory allocation interfaces are unchanged, but the vm layer will always provide non-pageable memory.

The OSF/1 server described above comes up multiuser on the prototype kernel, and the resulting system is accessible via the network. The only programs that fail are the ones requiring more than 64 K of stack (e.g., rsh) or requiring large virtual space (e.g., gcc: 16 Mbytes). Since the prototype does not implement swapping, it stops when physical memory is exhausted.

8 Performance

It is important to note that the prototype is neither complete nor optimized. The performance impacts from completing and optimizing it may not be negligible. The pmap management code is still page based and still manages an inverted page table. The kernel task runs in virtual mode and invokes pmap subroutines. In addition, the cost of memory translation is still present. In various places we did not have the time to optimize the code, especially when paging in or reading pages from devices, where we could move segments from the sender to the kernel at copyin time.

All of the measurements reported here were done on a BULL BM600 PC:

- 25 Mhz i386
- 16 Mbytes memory
- 300 Mbytes ESDI drive
- WD8003 Ethernet board

A page copy (4096 bytes) takes approximately 1 millisecond.

The OSF/1 server was running in multiuser mode, with statically bound commands and libraries.
The kernel and server configurations of interest are:

STD+WS:	Standard Mach micro kernel (NORMA_MK13)
STD+WS-vm	Real memory Mach microkernel(NORMA_MK11 based)
DEFAULT	Standard OSF/1 Server (SVR99)
SMALL	Reduced configuration server (Small buffer cache, no mapped u_area, no mapped time).

We address the following topics:

- RPCs
- VM
- disk and network I/O
- OSF/1 system call path
- Unix commands
- OSF/1 fork/exec

Attempts to measure program builds failed because the gcc compiler relies on a large virtual space; it unconditionally allocates 16 Mbytes of memory, which is more than the total available memory. The I/O performance figures must be analysed with care, as most of the elapsed time is "wait/io" time, and the extra page copies implied by the real memory kernel are not significant by comparison. However, in the case of a time sharing machines, the extra page copies would have a visible impact on performance. A user load benchmark like AIMIII would exhibit this effect.

8.1 RPCs

We used the standard *machipc test* program. This program is made up of one server and one client. They register and establish communication using the name server, in our case *snames*. Machipc uses the mach_msg interface with the MACH_SEND_MSG and the MACH_RCV_MSG options. With the out of line (ool) option, the buffers are not read or written by either the server or the client.

Table 1: RPC performance (μsec/RPC)

Configuration	Null RPC	4 KB RPC	8 KB RPC	16 KB RPC
STD+WS / SMALL	310	730	730	730
STD+WS-vm / SMALL	290	2640	4320	7930
Relative performance (%)	1.07	0.28	0.17	0.09

Since the data is never accessed by the user programs, the out of line pages are never copied by the system with the standard kernel. The real memory system must copy the pages twice; the cost of these copies (1ms per page) accounts for most of the performance differences between the systems in these cases.

8.2 VM

We used 2 programs. The first one is a program that was written to measure the cost of zero-fill page faults (uninitialized page accessed for first time) and lazy evaluation faults (resident but not mapped page). The program allocates 1 megabyte of memory and accesses every word (4 bytes) of it twice.This gives us the cost of a zero-fill page fault and the cost of an already mapped access. Then the program forks and accesses every word again, which gives us the cost of an unmapped access.

Table 2: VM Performance: Memory access - (µsec/page)

Configuration	zero fill	Lazy fault	Elapsed
STD+WS / SMALL	508	196	8984
STD+WS-vm / SMALL	0	0	9765
Difference	-508	-196	781

For the real memory kernel, although the program never takes page faults, the elapsed time is larger because the pages must be copied as part of the fork operation. The kernel with virtual memory support shares these pages copy-on-write instead.

The second program measures the cost of memory allocation and access time. It is run in two modes. The first mode allocates all the memory at once and then accesses each page. The second mode allocates one page at a time and accesses it.

Table 3: VM Performance: malloc (µsec/page)

Configuration	1 malloc + n touch	n malloc + n touch
STD+WS / SMALL	566	2109
STD+WS-vm / SMALL	390	1835
Relative Performance	1.45	1.15

The performance difference for the first mode is primarily due to the fault cost (~ 190 µsecs). The performance improvement for the real memory system in the second mode is somewhat surprising. The underlying segment must expand at every call, which may require segment copies and/or moves at various points during execution. The frequency and cost of these operations could be higher on a real memory system if physical memory is badly fragmented.

8.3 Disk I/O

These measurements utilize the OSF/1 file system. The program creates an 8 Mbyte file and then copies it to /dev/null.

Table 4: DIsk I/O Performance (Kbytes/sec)

Configuration	Create	Copy to /dev/null
STD+WS / DEFAULT	450	394
STD+WS / SMALL	391	320
STD+WS-vm / SMALL	335	301
Relative Performance	0.74 / 0.85	0.76 / 0.94

The performance is compared to two server configurations. With the DEFAULT configuration, the emulator uses shared memory to pass data to the server. With the SMALL configuration, data is passed in messages. In the real memory case this implies 2 extra page copies

8.4 Ethernet I/O

We used ftp with an HP/700 workstation to send a large file to /dev/null on the HP/700 and receive a large file from /dev/null on the HP/700.

Table 5: Ethernet I/O Performance (Kbytes/sec)

Configuration	Send	Receive
STD+WS / SMALL	110	200
STD+WS-vm / SMALL	100	180
Relative Performance	0.91	0.9

The file involved did not fit into the OSF/1 server's buffer cache.

8.5 System call path

We measured the getpid() and getuid() OSF/1 system calls.

Table 6: System Call path performance ((syscalls/sec)

Configuration	getpid()	getuid()
STD+WS / DEFAULT	13888	1538
STD+WS / SMALL	1562	1538
STD+WS-vm / SMALL	1666	1639
Relative Performance	0.12 / 1.07	1.06 / 1.06

There should be no difference in performance between the two systems with SMALL servers. Some of this difference may be due to variation involved in using the shell's time command instead of the gettimeofday and getrusage system calls. getpid() is considerably faster under the DEFAULT server because it takes advantage of the shared u_area to avoid an RPC to the server.

8.6 Unix commands

We measured typical Unix shell commands:

```
$ find /usr/include/sys -type f -exec grep -i copyright {} \;
$ find /usr/include/sys -type f -print
$ grep -i copyright /usr/include/sys/*
```

The related directory contained 115 files. The command outputs were redirected to /dev/null to avoid measuring console I/O performance.

Table 7: find/grep performance (elapsed seconds)

Command	STD+WS / SMALL	STD+WS-vm / SMALL	Relative performance
find + grep	37.4	46.8	0.80
find	0.7	0.9	0.77
grep	24.3	24.5	0.99

The difference between the first and third lines is that grep is executed 115 times in the former. The find command time is negligible. This seems to indicate that the fork/exec mechanism is leading to this performance drop. The next measurement confirms this.

8.7 Fork/Exec/Wait

The test program loops on forking/execing and waiting for exec to return. The executed program is a null program: `main() {}`

Table 8: fork/exec/wait (µsec/call)

Configuration	µsec/call
STD+WS / SMALL	49700
STD+WS-vm / SMALL	95100
Relative Performance	0.52

The program is 52 Kbytes in size and the null program is 22 Kbytes. In the real memory case, both the main and the null program are copied from the original cached object ((52+22)/4 = 18.5 ms). At fork time, both the text and the data are copied ((52+64)/4 = 29 ms). These times account for most of the performance difference.

9 Future work

The work described in this paper constitutes an initial prototype that demonstrates the feasibility of Mach as an operating system base for real memory hardware systems. There is still a fair amount of work to be done to create a version of Mach that runs on a real memory platform. The pmap interface needs to be redesigned to match the segmented memory model. This should result in a considerably simpler pmap module. The kernel needs to be adapted to execute in physical mode (versus the virtual mode of the current prototype). No change should be needed to our prototype's management of the kernel's (physical) address space to accomplish this. Segment swapping technology needs to be added; there is nothing new to be invented here, and a number of operating system implementations from which this technology could be adapted. In addition, there is a reasonable amount of code cleanup and error checking to be added, as would be expected with the first functional version of a prototype system. Additional extensions are needed to support multiprocessors and hardware-imposed segment alignment restrictions.

The prototype makes no attempt to optimize handling of out of line memory. Every virtual copy in the VM system is replaced by a physical copy in the real memory system. This results in a number of needless memory copies (e.g., a paged in page may be copied up to three times). In some cases, these are a matter of straightforward optimization work, but in other cases, kernel interface changes are needed. For example, a 'third-party' I/O capability in device_read() and device_write() that indicated the ultimate source or destination of the data in another task would avoid copies to and from the server's address space.

Some minor changes to the OSF/1 server are needed because certain virtual memory operations become quite expensive in a real memory system. The OSF/1 server knows that the kernel optimizes virtual memory operations. In traditional Unix implementations, the text portion of a task is often shared among the tasks running the same program. When a task needs to modify the text (e.g., for debugging), a private copy of the text is made. OSF/1 takes advantage of the fact that mapping an object as a copy of an underlying shared object is relatively cheap due to the use of copy on write techniques. But in the real memory case, mapping an object as a copy implies a physical copy of the object, with unfortunate (and unnecessary) performance impact.

10 Conclusion

The overall objective of this work has been to investigate the possibility of adapting the Mach micro-kernel to real memory hardware. We have shown that it is possible to do this without major modifications to the kernel interface. This result seems natural, since paged virtual memory functionality is a superset of segmented memory functionality. In the recent past, researchers improved operating systems by using page-based virtual memory, and the present exercise has consisted of removing this improvement. Our performance measurements show that the absence of virtual memory is not always a handicap, and may be an advantage in some cases. This work has demonstrated the adaptability of the Mach kernel to yet another interesting class of hardware architectures.

11 References

[1] D. Black, D. Golub, D. Julin, R. Rashid, R. Draves, R. Dean, A. Forin, J. Barrera, H. Tokuda, G. Malan, and D. Bohman, Microkernel Operating System Architecture and Mach, *Proceedings of the Usenix Workshop on Micro-kernels and Other Kernel Architectures*, Seattle, WA, April 1992, pp. 11-30.

[2] D. Golub, R. Dean, A. Forin, and R. Rashid. Unix as an Application Program. *Proceedings of the 1990 Summer Usenix Technical Conference*., Anaheim, CA, June 1990, pp 87-96.

[3] INMOS Limited, The T9000 Transputer Products Overview Manual, First Edition 1991.

[4] Open Software Foundation, OSF/1 Operating System Programmer's Reference, Prentice-Hall, Englewood Cliffs, NJ, 1991.

[5] S. Patience, Redirecting System Calls in Mach 3.0: An Alternative to the Emulator, *Proceedings of the Third Usenix Mach Symposium*, Santa Fe, NM, April 1993, in this proceedings.

[6] R. Rashid, Threads of a New System, *Unix Review*, Volume 4, August 1986.

[7] R. Rashid, A. Tevanian, Jr., M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew,. Architecture-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures, *IEEE Transactions on Computers*, Volume 37, Number 8 (August, 1988), pp. 896-908.

[8] R. Russell, The CRAY-1 Computer System, *Communications of the ACM*, Volume 21, Number 1 (January 1978), pp.63-72

[9] B. Wheeler and B. Bershad, Consistency Management for Virtually Indexed Caches, *Proceedings of the Fifth International Conference on Archtectural Support for Programming Languages and Operating Systems*, Boston, MA, October 1992, pp. 124-136.