Moving the Default Memory Manager out of the Mach Kernel

David B. Golub
Richard P. Draves

School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, Pennsylvania 15213

(412) 268-7667

Internet: dbg@cs.cmu.edu, rpd@cs.cmu.edu

1. Abstract

We have implemented a default memory manager for the Mach 3.0 kernel that resides entirely in user space. The default memory manager uses a small set of kernel privileges to lock itself into memory, preventing deadlocks against other Mach system services. An extension to the Mach boot sequence loads both the kernel and user program images at system startup time. The resulting system allows the default memory manager to be built and run in a standard user-level environment, but still operates with the high reliability required by the Mach kernel.

The default memory manager is bundled with another component of the Mach 3.0 system: the bootstrap service. This service starts the initial set of system servers that make up a complete operating system based on the Mach 3.0 kernel. Since the real file system may be one of these servers, the bootstrap service needs its own copy of a subset of the file system. This is shared with the default memory manager. Placing these two components outside the kernel allows them to be easily reconfigured with different file systems.

2. Introduction

We have implemented a version of the Mach 3.0 operating system in which all paging traffic to backing storage has been moved outside of the operating system kernel. This is a significant departure not only from previous versions of Mach 3.0, but also from more traditional operating systems, in which memory management for at least temporary memory has been an integral part of the kernel. This work has involved solving a number of technical challenges. The privileges required to manage temporary memory without deadlocking with the rest of the system have traditionally been only available to services embedded within the operating system kernel. We also had concerns about the time and space required to handle paging requests outside the kernel. However, the resulting system runs as reliably as an operating system with conventional temporary memory management. With a reasonable amount of memory, the system runs as fast as previous versions of Mach 3.0.

This research was sponsored in part by The Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title ‘‘Research on Parallel Computing’’, ARPA Order No. 7330, issued by DARPA/CMO under Contract MDS972-90-C-0035 and in part by the Open Software Foundation (OSF). Draves was supported by a fellowship from the Fannie and John Hertz Foundation.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, OSF, the Fannie and John Hertz Foundation, or the U.S. government.

The default memory manager is an essential component of the Mach 3.0 operating system. It provides backing storage for temporary memory objects. Since its clients may include other components of the system, it cannot depend upon them for any vital services, such as backing storage allocation, without risking deadlock. Traditionally, such vital components of an operating system have been integral parts of the kernel, often using services unavailable to other subsystems. Earlier versions of Mach 3.0 followed suit: the default memory manager was built into the kernel.

Over the past year, we have re-implemented the default memory manager as a task residing entirely in user space. The default memory manager needs only a few kernel primitives to obtain the privileges it needs: wiring down its own code and data, ensuring that data given to it is locked in memory, and priority in allocating new memory. These privileges are part of the normal Mach kernel interface, and potentially available to any user task. All other communication between the kernel and the default memory manager is via the kernel interface used by all other Mach memory managers.

The default memory manager is bundled with the Mach bootstrap service, which loads the initial set of servers. The bootstrap service and the default memory manager live in the same task and share a subset of the file system code to access existing files. Moving this task into user space has required rewriting the Mach kernel boot sequence so that two program images, the kernel and the bootstrap/pager task, can be loaded at system boot time. The kernel and the bootstrap/pager task are built independently. A new program, *makeboot*, combines the two into a bootable system image.

The Mach kernel can now be built without the extra support required for embedded server tasks that were present in earlier versions. There are no more naming conflicts in the kernel between internal kernel routines and user-level interfaces to them. The bootstrap service and default memory manager, similarly, can be built as normal user tasks. The resulting system is as robust as a standard Mach 3.0 system with the default memory manager inside the kernel. Performance on machines with reasonable amounts of memory is essentially identical.

3. In-Kernel Default Memory Management

A manager for temporary virtual memory must operate under some very adverse conditions. In conventional operating systems, the temporary memory manager must be able to allocate storage when no other component of the system can do so. In Mach, the temporary memory manager is, by default, the memory manager of last resort for all other memory managers. This role of *default* memory manager forces it to avoid most standard system services. These constraints have kept the temporary memory manager intimately tied to the kernel in most operating systems, including earlier versions of Mach.

Operating systems have traditionally provided a variety of memory management services implemented within the operating system kernel. [1] [5] They include:

- Allocating temporary memory for user programs and system services.
- Mapping executable files and data files into a task’s address space.
- Mapping IO devices such as graphics buffers.
- Sharing files or temporary memory between tasks.

The external memory management interface present in Mach 2.5 and Mach 3.0 has allowed most of these services to be provided by servers outside the kernel. However, temporary memory management has still resided within the kernel.

Temporary memory management has traditionally been implemented in the kernel because it must make use of other system services at times when they would normally be unavailable or risky to use. Temporary memory usually does not have pre-allocated backing storage. The temporary memory manager must allocate the backing storage when it is needed. Since pages are written to backing storage when the system is short of main memory, the routines that allocate backing storage must be able to operate in such conditions. Furthermore, allocating backing storage often requires allocating memory within the memory manager, further increasing the strain on main memory.

In a Mach 3.0 system, the temporary memory manager acquires an even more difficult role: that of default or last-resort memory manager for the entire system [6]. Memory management services may be provided by any task in the system, not only by trusted system components. Therefore memory managers cannot be assumed to be timely or reliable. A memory manager may take an arbitrary amount of time to write a page to its backing storage. It may also malfunction (accidentally or maliciously) and not write the page at all. During the time the page is in transit (paged out to its memory manager, but not yet written), it still consumes main memory. If the operating system is short of memory, this in-transit page must be paged out. Since the page is viewed as temporary memory while it is in transit, the temporary memory manager thus assumes responsibility for it.

The temporary memory manager therefore must not depend on other servers in the system, since these servers may depend on the temporary memory manager themselves. This requirement has, so far, kept the temporary memory manager within the Mach kernel.

4. Constraints on the Default Memory Manager

The default memory manager faces several stringent implementation constraints, because of its crucial role in the Mach system. It cannot depend on any system service that in turn could use it to provide temporary memory. It must also be able to function when the system is short of memory.

- The default memory manager must be resident - there is no other service to page its code and data into memory.
- All pages moving between the kernel and the default memory manager, and between the default memory manager and its backing storage service, must remain resident.
- The default memory manager cannot block to wait for more physical memory, since freeing physical memory requires that its contents be paged out (through the default memory manager).
- If the default memory manager is grouped with another service, the threads that handle default memory manager functions must be distinguished from threads that do not. When the system is short of memory, the unprivileged threads (those not handling default memory manager functions) may block waiting for physical memory. If the default memory manager's threads can block waiting for the unprivileged threads, a deadlock will result.
- The default memory manager cannot block to wait for the file system to allocate temporary disk storage, since the file system itself may be pageable. This requires that temporary paging space be allocated in advance, or that the file system be as privileged as the default memory manager.

To meet these requirements, the Mach 2.5 default memory manager was implemented as a task bound tightly within the kernel. It used several services that were unavailable to other components of the system.

- The default memory manager was allowed to allocate kernel memory even when the system was short of memory. Threads within the default memory manager were marked as *vm_privileged* threads, and were allowed to allocate memory when other threads would block.

- Threads within the default memory manager were marked as unswappable. The kernel could unlock and page out the entire kernel stack of a long-waiting thread when short of memory. Doing this to a default memory manager thread would, of course, deadlock the system.
- The default memory manager used the existing BSD 4.2 file system code to manage its disk storage and allocate new disk space for paging. The code was written to run in a standard kernel environment, which assumed that all of the code and data was resident, and the kernel stack of any thread executing the file system routines was locked into memory.

However, all paging traffic between the kernel and the default memory manager took place through the external memory management interface - an interface designed to be used with servers residing outside the kernel.

The original Mach default memory manager therefore used a mixture of user and kernel services. It moved pages to and from the kernel using the external memory management interface, as if it were a user task. It accessed the kernel memory and disk allocation routines through its privileged position inside the kernel. This structure, preserved even in the first Mach 3.0 implementations, required the kernel to provide the equivalent of a user-mode interface internally. It spoiled the clean separation between the Mach kernel and user-mode functions.

5. Moving Out of the Kernel

To move the default memory manager out of the kernel's address space, the privileged kernel functions had to be replaced by equivalent functions accessible from user programs. Fortunately, the default memory manager already used the standard external memory management interface to handle paging operations, using external data types (ports) to represent memory objects, rather than handles to the kernel's internal data structures.

Only a few kernel operations were accessed directly: allocating disk storage, synchronizing threads, allocating locked-down memory, and preventing threads from being swapped out. These operations were replaced with equivalent user-mode routines, or were added to the kernel interface where missing.

5.1. Kernel Interface Extensions

The kernel interface has been extended with two calls originally proposed [2] for supporting real-time Mach:

- ```
  kern_return_t vm_wire(
          priv_host_t     host_port,
          task_t          task,
          vm_address_t    start,
          vm_size_t       size,
          vm_prot_t       access)
  ```

  Locks ("wires") the specified range of virtual addresses for the task into memory. Accesses denoted by *access* cannot fault. Memory is unlocked by specifying no wired access (*VM_PROT_NONE*).

- ```
  kern_return_t thread_wire(
          priv_host_t     host_port,
          thread_t        thread)
  ```

  Permanently acquires a kernel stack for the thread, so that the thread can always run. Allows the thread's requests for physical memory to always succeed.

Both of these calls are only accessible to tasks with send rights to the privileged host port. There is currently no other resource control on memory allocated by these calls.

An existing kernel call identifies the default memory manager to the kernel:

```
kern_return_t vm_set_default_memory_manager(
        priv_host_t     host_port,
        mach_port_t     *default_manager)
```

Returns the old value of the default memory manager service port. If the supplied value for the default manager port is not MACH_PORT_NULL, the kernel will use this port as the default memory manager service port.

The kernel uses the default memory manager service port to request the default memory manager to handle paging for temporary objects. It also arranges that memory sent to the task with receive rights for this port remains locked down in physical memory: see section 5.5.

5.2. File System Access

The largest kernel function used directly by the Mach 2.5 default memory manager was the file system code. Since the file system code was resident, the default memory manager could take advantage of it to allocate new paging space. However, this required that the all of the file system code be resident, and that any thread that accessed the file system also be resident while it was executing that code.

To preserve the flexibility of being able to allocate new paging space would have required us to include the entire BSD file system code in the default memory manager. We decided instead to separate the file system and the memory manager, and use other mechanisms to allocate paging storage.

The default memory manager bypasses the disk allocation problem by using a pre-specified list of disk blocks. At system startup time, the bootstrap sequence looks for a paging file with a well known name: */mach_servers/paging_file*. It then reads the index information in the file to obtain the disk blocks occupied by the file, and passes them to the default memory manager. The paging file must, therefore, be pre-allocated with all of the blocks needed for paging.

5.3. Synchronization and Locking

The default memory manager uses the CThreads package [3] to provide locking and synchronization between different threads. The CThreads package provides two synchronization primitives: mutual exclusion locks, for controlling exclusive access to data structures, and condition variables, used by multiple threads to wait for events and signal them. Mutual exclusion locks were used to replace the simple spin locks used inside the kernel. Explicit calls to the kernel sleep and wakeup primitives were replaced with condition variables. The CThreads package does not provide multiple reader/single writer locks, but since these were used only in one routine in the file system, they were easily replaced with exclusive locking.

5.4. Allocating Locked-Down Memory

We replaced the standard C library memory allocator (*malloc* and *free*) with our own routines. These call the kernel routine *vm_allocate* to allocate full pages. This is replaced by a routine that allocates memory from the kernel (using *vm_map*), and locks it into memory using the new *vm_wire* call. The *thread_wire* call allows the threads in the default memory manager to always allocate memory.

5.5. Locking Memory sent to the Default Memory Manager

The default memory manager receives memory in messages from only a few sources in the kernel and the device service. These have been modified to know when they are sending memory to the default memory manager, and to lock it in memory.

- The pageout daemon locks pages being paged-out from temporary memory objects. These memory objects are created by the kernel to hold temporary memory, or to shadow memory that is passed copy-on-write from one task to another. If a page in one of these objects is sent to the default memory manager, it is locked down.
- The device service locks memory being read from backing storage devices (typically the disk) to the default memory manager. The *device_read* routine looks at the task holding receive rights for the destination of its reply message. If that task is also the receiver for the default memory manager's service port, the memory is locked down.

5.6. Locking Memory sent to Backing Storage

The default memory manager writes data to disk via the kernel device service, using the *device_write* call. This routine must not allow the data to be paged out, and must not block to wait for other paged-out data to be paged in. To avoid blocking, the *device_write* request message is processed in the context of the calling thread, not by a separate device service task. The message-send operation does not return until the memory to be written is queued for the disk.

If the device service were a separate task, it would need two classes of service threads. One set would handle write requests from the default memory manager, containing locked-down memory; the other set would handle requests from all other clients, containing pageable memory. If one pool of service threads handled both pageable and locked-down memory, a write request from the default memory manager could be blocked because pageable memory in a previous write request was paged out. This would result in deadlock, since the device service would need the default memory manager to retrieve the paged-out memory.

6. User-mode Environment for the Default Memory Manager

The default memory manager is built and operates as part of the Mach 3.0 bootstrap service task. This task is the first to run in the system once the kernel is initialized. In addition to the default memory manager, it contains the bootstrap loader for user-mode servers. Since both of these components share a common file system, they were moved out of the kernel as a unit.

6.1. Structure of the Bootstrap Service Task

The bootstrap loader loads and executes the first server task: the BSD server for the BSD single-server system, or the configuration manager for the multi-server operating system. It makes use of a bootstrap file system module to find the server file and read it into a new user task. It passes the privileged host port and the device server port to the initial server, thus giving the initial server access to all system privileged operations.

The default memory manager runs after the bootstrap service. It uses the bootstrap file system to find the paging file, and to read and write the file as paging operations take place. It exports an interface to allow other tasks to use it as a shared memory service; however, it provides no network consistency management, so the objects it exports cannot be shared across multiple machines.

The bootstrap file system that both of these components use provides a small subset of the operations available on a BSD 4.2 or 4.3 file system. It can look up files by device and name, and read and write existing data blocks in files. It cannot allocate new blocks to files, or create new files. To do so would conflict with the real file system. The interface to the bootstrap file system is well defined; the code can readily be replaced by one that understands a different disk format.

6.2. Loading Two Boot Images

Since the default memory manager and bootstrap service have been separated from the kernel, there are now two executable images (kernel and bootstrap) to be loaded when the system is run. We decided not to modify the existing initial bootstrap loaders to load both images. Most of them are proprietary to the individual manufacturers; in at least one case, we do not even have access to the source code. Instead, we expanded the kernel boot file to contain both images.

A new program called *makeboot* builds an executable out of both boot images (kernel and bootstrap) so that the complete image of the default memory manager and bootstrap task is in the kernel data segment when the kernel is loaded. The kernel then moves the bootstrap image to a newly created user task and starts a user-mode thread in it.

The boot file looks like a normal bootable image, but has no uninitialized storage (BSS) or symbol table. The size of the combined text and data segments is set to include the entire file. At the start of the kernel uninitialized data (*BSS*) is a *struct boot_info* describing the sizes of the rest of the file:

```
struct boot_info {
    vm_size_t   kern_sym_size;   /* size of kernel symbol table */
    vm_size_t   boot_image_size; /* size of bootstrap image */
    vm_size_t   load_info_size;  /* size of load information for
                                    bootstrap image */
};
```

Following this are three new sections:

- The kernel symbol table. This may include information pulled from the normal executable file header to make the symbol table self-describing.
- The boot image. This is identical to the bootstrap's executable file, except that its symbol table has also been made self-describing.
- Loader information for the boot image. This is in a compiler-independent format, to reduce the amount of machine-dependent code needed in the kernel.

At startup, the Mach boot file is loaded into contiguous physical memory. The kernel must, as its first action, move the symbol table, the bootstrap image, and the load information out of its BSS section. In addition, it must align the bootstrap image on a page boundary, so that the bootstrap image can be mapped to the bootstrap task rather than being copied. All of this may have to be done before virtual memory mapping is set up, since the kernel typically uses the first few pages of physical memory after the BSS for building page tables.

The kernel proceeds with its normal initialization. After starting all the internal threads, it creates a task and thread to run the bootstrap code. It then creates a memory object mapping the resident pages holding the bootstrap code. It maps this memory object into the first task as it would a normal executable file. It then allocates a stack for the task, passes it a small set of arguments, and starts it running in user space.

7. Results

Moving the default memory manager outside the kernel has both benefits and drawbacks. The build environment for both the kernel and the default pager (and bootstrap task) is considerably simplified, since user and kernel environments are not mixed. The system operates almost as fast as previous versions of Mach 3.0 that contained the default memory manager within the kernel. However, it does occupies more physical memory than previous versions. Using a pre-allocated paging file is adequate for development systems, but not for production use.

7.1. Building the System

Building the Mach 3.0 kernel and default memory manager is much easier. Both the kernel and the default memory manager are now self-contained. A new bootstrap service can be added to the Mach 3.0 boot file without rebuilding the kernel.

The kernel no longer has to support a user environment for otherwise self-contained servers. There is no longer a naming conflict between kernel interface routines called by servers and the kernel functions themselves. The default memory manager no longer has to distinguish global kernel names for ports from its own local names for them; this was a source of confusion in the Mach 2.5 default memory manager.

The default pager and bootstrap service can be built as a standard user task, using the Mach kernel primitives and device support routines, and the standard CThreads library. Where we have re-implemented routines from the standard Mach libraries, we have done so for performance or space reasons, not for new functions.

The kernel and the bootstrap service are built independently and combined with the *makeboot* program. If someone wants to experiment with a new default memory manager, or change the bootstrap file system to accommodate different disk formats, it is not necessary to rebuild the kernel. Only the boot file needs to be rebuilt.

7.2. Performance

Performance figures, comparing the MK63 release of Mach 3.0 with the out-of-kernel default memory manager, are shown in table 7-1. All measurements were run on a 25Mhz 386 processor with 8 Megabytes of memory and an ISA bus.

When operating with a reasonable amount of main memory, moving the default memory manager outside of the kernel has only a small effect on overall system performance. A memory-intensive paging benchmark, paging 7 megabytes of virtual memory against 6.5 megabytes of available physical memory, runs less than 1% slower with the default memory manager out of the kernel. We expect that a balanced workload, not dominated by paging operations, will show no difference.

We expected that overall system performance would be unchanged by having the default memory manager outside of the kernel. When the system is paging heavily, disk operations would dominate the

Memory Size, Megabytes		Elapsed Time, Seconds		
Maximum Memory	*Available Memory*	*MK63*	*Out-of-Kernel*	*Difference, Percent*
8	6.5	97.0	97.6	0.6
4	2.5	124.2	131.4	5.8
3	1.5	131.6	138.9	5.5

Table 7-1: Performance of out-of-kernel system vs. MK63

cost anyway. Otherwise, the only difference would be the cost of crossing the user/kernel boundary. The previous implementation of the default memory manager was already structured as a separate task, communicating with the kernel and the disk driver via IPC: this already forced messages to be copied and data to be remapped from the default memory manager's address space to the kernel's address space. The only added cost would be the actual kernel-to-user mode switch. These expectations correspond to what we have measured.

As the available memory decreases, the performance drops, but slowly. With 2.5 megabytes of memory, the paging test runs about 6% slower. The extra memory taken by the out-of-kernel bootstrap task does not account for the slowdown: adjusting the available memory to compensate does not change the performance figures.

One cause of the slowdown is the difference between the kernel and the CThread locking facilities. Mutual exclusion locks in the kernel compile into null routines when the kernel is built for a uniprocessor; since kernel threads are not preemptible, locks are not needed when the thread cannot block. Locking is always needed, however, when running in user space, since user-mode threads are preemptible. When the default memory manager was first moved outside the kernel, it grabbed and released a mutual exclusion lock nine times while reading one page from the disk. Reducing the number of lock/unlock pairs to four reduced the performance degradation from 10% to 6%. We expect that further tuning of the code will regain nearly all of the lost time.

7.3. Memory Usage

The sizes for the kernel and bootstrap image are compared with the size of an MK63 kernel in table 7-2. The extra memory used by the out-of-kernel default pager system is broken down in table 7-3. The loaded image occupies more space than it does in the file because the text and data segments and the symbol tables are rounded to page boundaries (4096 bytes).

File	*Text*	*Data*	*BSS*	*Symbols*
MK63	315360	33096	54364	69524
Kernel	294880	32224	52992	65386
Bootstrap	40928	2676	3616	19345

Table 7-2: Executable Image Sizes

Splitting the default memory manager out of the kernel does increase the system's locked-down memory usage. The default memory manager's code and data is still locked in memory. The C Threads library is

Size in KBytes	*Used By*
48	Code and Data for Bootstrap
24	Symbol Table for Bootstrap
-24	(Code and Data Removed from Kernel)
-4	(Symbol Table Removed from Kernel)
12	Page Table for Bootstrap Task
40	CThread Stacks for Each CThread
4	Extra Waiting Stack allocated by CThreads
12	Temporary Storage Managed by Memory Allocator
112	Total

Table 7-3: Extra Memory Used by Bootstrap Task

larger than the equivalent kernel locking primitives, and is not shared with other functions as the kernel primitives are. Each thread in the default memory manager has a user-mode stack as well as a kernel-mode stack. Since the threads must be wired, their kernel stacks cannot be discarded or handed-off to another thread.

Careful work on the default memory manager could reduce the extra memory usage, but not eliminate it entirely. The user-mode stacks could be reduced to 4 K bytes each, since only the bootstrap server loading code needs as much as 8 K bytes of stack. It would also be possible to discard the code pages occupied only by the bootstrap service loader routines once they have been used. In a production system, the kernel and bootstrap loader would be configured without run-time debugging information: this would save 20 K bytes from the bootstrap task alone.

7.4. Backing Storage Allocation

Using a pre-allocated paging file is currently the least satisfactory aspect of the default memory manager design. Mach 2.5, since it integrated the file system with the default memory manager, could use a variety of paging space allocation strategies: reserved disk partitions, pre-allocated paging files, and expandable paging files. In Mach 3.0, the file system is expected to be separated from the default memory manager. The two tasks must cooperate to supply more disk blocks for paging; otherwise they could both allocate the same blocks from the file system, destroying it. There is currently no way to give more paging space to the default memory manager; one could be added.

8. Further Work

Since the bootstrap service is now independent of the kernel environment, we can experiment with adding additional functions. The default memory manager interface could be extended, as previously mentioned, to request additional paging storage from the file system and to return paging storage that is no longer needed. The bootstrap loader could be used to load all of the servers for the multi-server BSD emulation, not just its configuration service. The bootstrap task is also the logical place to put any out-of-kernel disk drivers [4]that need to be shared between the default memory manager and the file system. To accommodate the space taken by these functions, we would need to restructure the bootstrap service so that code used only at startup can be freed.

For the adventurous, the default memory manager could even be made into a loadable server, separate from the bootstrap service. The default memory manager views backing storage as a list of disk blocks, not as files. Therefore it does not actually need to share the file system code with the bootstrap service. The default memory manager could be loaded and run just as any other system server; at some later time (for example, when *swapon* is executed), a file system server could provide it with a list of disk blocks to use.

9. Conclusions

We have shown that moving all memory management outside the Mach 3.0 kernel is not only possible, but beneficial. The resulting system runs as reliably and almost as fast as earlier versions of Mach 3.0. It is considerably easier to build and can more flexibly be reconfigured. These benefits are obtained at the cost of only a small amount of extra memory.

References

[1] L. A. Belady, R. P. Parmelee, and C. A. Scalzi.
The IBM History of Memory Management Technology.
IBM Journal of Research and Development 25(5):491-503, September, 1981.

[2] David L. Black.
Mach Interface Proposals - Priorities, Handoff, Wiring.
Internal Mach Project Memo - 13 August 1989.

[3] Eric C. Cooper and Richard P. Draves.
C Threads.
Technical Report, Department of Computer Science, Carnegie Mellon University, July, 1987.

[4] Alessandro Forin, David Golub, and Brian Bershad.
An I/O System for Mach 3.0.
In *Proceedings of the Second Mach Symposium*. The UseNIX Association, November, 1991.

[5] E. L. Organick.
The Multics System: An Examination of its Structure.
MIT Press, Cambridge, Mass., 1972.

[6] Michael W. Young.
Exporting a User Interface to Memory Management from a Communication-Oriented Operating System.
PhD thesis, Carnegie Mellon University, November, 1989.

Table of Contents

List of Tables