OSF/1 Virtual Memory Improvements

David Black, Jeff Carter, George Feinberg,
Rod MacDonald, Shashi Mangalat, Eric Shienbrood,
Jim Van Sciver, Ping Wang

October, 1991

Abstract

The second release of the Open Software Foundation's operating system, OSF/1, is nearing completion. Among the features in this release are an increased focus on performance and robustness. This paper details the improvements made in these areas to the virtual memory subsystem. The improvements and the reasoning behind four selected changes are presented: deadlock removal in the VM code, eager allocation of backing store for kernel stacks, the addition of clustered page operations, and the addition of swapping.

1. Introduction

The first release of the Open Software Foundation's operating system offering, OSF/1, was made generally available in December of 1990. Since that initial release the OSF operating system group has been continuing its efforts to enhance the commercial viability of OSF/1 for the benefit of our members. Among the goals for this second release, termed OSF/1 R1.1, were performance and robustness improvements.

When considering subsystems for performance improvement, we wanted to choose only those areas which were deemed both critical and architecture neutral. The OSF/1 virtual memory subsystem met both of these qualifying criteria and, as a result, was the primary focus of the performance/robustness efforts in the R1.1 project.

OSF/1 is derived from Mach 2.5 technology. The Mach virtual memory subsystem has a number of desirable characteristics. Its architecture is clean and easily understood. The boundary between machine independent and machine dependent code is clearly demarcated with a well defined set of interfaces. Mach also exports an interface that allows user space applications to participate in memory management. Such an external memory manager (EMM) is responsible for managing secondary storage, supplying page data to the kernel on request, and accepting page write requests. External memory managers run in separate tasks, and communicate with the kernel using Mach IPC.

We examined the virtual memory subsystem in two ways. First, we stressed the system under increasing load and compared the OSF/1 system performance with another commercial version of Unix on the same hardware. Second, we ran small programs to stimulate specific virtual memory metrics, such as paging rates, and compared our results to the other system. OSF/1 R1.0 performed worse than the commercially available system and, in the process of load testing, a number of undesirable system failures were discovered.

This paper describes the modifications made to the virtual memory subsystem to correct these problems. The robustness changes were the removal of deadlocks when wiring pages and the preallocation of backing store for kernel stacks. Also, clustered paging was added to improve performance. The addition of swapping improves both system performance and robustness.

2. Deadlock Removal

The routine vm_map_pageable() is used throughout the kernel for wiring and unwiring memory. Three problems were found with the R1.0 version of this routine:

- it holds the map lock for the duration of the faults that wire down pages. This means that vm_fault is called with the map locked for read. Under certain conditions deadlock will result when the kernel tries to acquire the map lock for write.
- it ignores error codes from vm_fault. The system assumes that wiring down memory cannot fail. Unusual error conditions occur when it does.
- vm_map entry deletion does not synchronize correctly for wired entries. A wired entry may be unwired and deleted regardless of the wired state.

It became obvious that in order to avoid holding the map lock we needed to keep the current wired state of a map entry. A new flag, *in_transition*, was added to the vm_map_entry structure to indicate if an entry is being wired or unwired. During a wiring operation a thread will block if it encounters an in_transition entry. The blocked thread indicates that it wants the entry by setting the new flag *needs_wakeup*. The thread which set the in_transition bit will wakeup threads waiting on the entry once it completes the wiring.

If the vm_fault fails during a wiring operation all entries wired thus far are unwired and a failure status is returned to the caller. Failure to unwire an entry is a panic condition.

When unwiring an entry, either directly or by deleting an address range, wired entries are not unwired unless all the wirings are removed. The *wired_count* member is a counter indicating how many kernel wirings have been performed on the entry. An entry deletion always removes one kernel wiring and, if the wired count falls to zero, continues to unwire and deallocate the entry.

An additional wired reference count, *user_wired_count*, tracks user wirings. This count allows users to wire down memory. No comparable user interface currently exists yet to take advantage of the interface. In order to protect the kernel from loss of available physical memory, deleting user wired memory always unwires the memory, disregarding the number of user wirings. This policy guarantees that all user wired memory is unwired and deallocated when a user program exits.

The vm_map_wire and vm_map_unwire routines now take an additional argument to distinguish between kernel or user requests. A new macro vm_map_pageable_user() is provided to facilitate this. Currently only the plock system call uses vm_map_pageable_user.

By reducing the scope of the map lock during wiring/unwiring operation we achieve better parallelism and performance. The number of simultaneous wirings on a single map no longer depends on the map lock. Operation granularity is finer since synchronization needs to be done only for overlapping regions at the entry level rather than at the map level. On a multiprocessor this allows parallel wirings of the same map at different address ranges.

3. Eager Allocation of Backing Store for Kernel Stacks

In OSF/1 R1.0 and Mach 2.5 backing store allocation for a virtual page is deferred until the page needs to be paged out. This is a good performance strategy since many pages are created and destroyed without ever being paged out. However, in the case of kernel thread stacks, this policy can lead to disaster if a stack page must be paged out when there is no space available in the paging file. The pageout operation will fail but the operating system does not discover this failure until it makes a subsequent attempt to page in the kernel stack. The end result is an unrecoverable page fault when the kernel attempts to touch that

kernel stack, causing a system crash.

The solution is to allocate backing store for kernel stack space at the time the stack is created. Then either the thread creation fails gracefully because there was no more space in the paging file, or the kernel can be assured that all page faults on the thread's kernel stack will succeed.

Mach uses a client server model to perform paging instead of embedding paging in the kernel's virtual memory system. The three components of this model are the kernel's VM subsystem (client), a pager which can be either internal or external to the kernel (server), and a communications mechanism. Implementing backing store preallocation required changes to each of these three components. Allocate and free interfaces had to be added to both the client (kernel) and server (pager), and the kernel IPC mechanism needed extensions to support kernel initiated RPCs. Mach documentation usually refers to pagers as memory managers.

3.1 External Interfaces

There are two components to the interface between Mach and the pager that handles kernel created memory (e.g., kernel stacks). This pager is known as the *default pager*; this is the vnode pager in OSF/1. In addition to the standard interface used by all external pagers, this pager implements an extension to support paging of kernel created memory. The routines in this extension allow the default pager to accept responsibility for a memory object created by the kernel, and to correctly implement the paging of memory copied from another memory object that may not be managed by the default pager. The actual paging operations for kernel created memory use routines from both the standard interface and this extension.

Our implementation of eager allocation added two routines to this extension of the external memory management interface. These routines are only used in the paging of kernel created memory, such as kernel stacks.

```
kern_return_t
memory_object_data_allocate(mem_obj, mem_obj_control, offset, length)
        memory_object_t         mem_obj;
        memory_object_control_t mem_obj_control;
        vm_offset_t             offset;
        vm_size_t               length;

kern_return_t
memory_object_data_terminate(mem_obj, mem_obj_control, offset, length)
        memory_object_t         mem_obj;
        memory_object_control_t mem_obj_control;
        vm_offset_t             offset;
        vm_size_t               length;
```

The *memory_object_data_allocate()* interface asks the pager to allocate *length* bytes of backing store starting at *offset* in the specified memory object *mem_obj*. (The *mem_obj_control* argument allows the pager to identify which kernel made the request if more than one kernel is using the memory object.) Conversely, *memory_object_data_terminate()* asks the pager to deallocate previously allocated backing store. Unlike all other pager interfaces, memory_object_data_allocate is synchronous. This ensures that the backing store is allocated before the operation requiring it completes. This routine returns KERN_SUCCESS upon successful completion or KERN_RESOURCE_SHORTAGE if there is insufficient backing store. This allows thread creation to fail (synchronously) when backing store is not available for the new thread's kernel stack. The companion routine, memory_object_data_terminate(), is asynchronous. Attempts to deallocate non-existent backing store are ignored.

3.2 Internal Interfaces

Two similar routines have been added to the internal VM interfaces. These routines provide the kernel (client) side interface to backing store allocation/deallocation functions. These interfaces can be used for the allocation/deallocation of backing store for any subset of virtual space.

```
kern_return_t
vm_map_backing_alloc(map, vaddr, size)
        vm_map_t        map;
        vm_offset_t     vaddr;
        vm_size_t       size;

void
vm_map_backing_free(map, vaddr, size)
        vm_map_t        map;
        vm_offset_t     vaddr;
        vm_size_t       size;
```

The *vm_map_backing_alloc()* routine sends a request to the default pager to allocate backing store for the subset of *map* that starts at *vaddr* and has size *size*. If the backing store is successfully allocated or had been previously allocated, KERN_SUCCESS is returned. Otherwise KERN_RESOURCE_SHORTAGE is returned.

Conversely, *vm_map_backing_free()* sends a request to the default pager to deallocate backing store. This routine does not wait for a reply because the corresponding memory_object_data_terminate request is asynchronous and it is impossible for the free request to return an error. If the backing store had been previously allocated then the vm_map_backing_free() request would succeed. If the backing store is unallocated at the time of the request then the request is simply ignored.

3.3 Mach Kernel RPC

The implementation of backing store allocation required modifying the Mach IPC system to support kernel initiated RPCs. Mach IPC normally assumes that all messages sent to the kernel are operation invocations, and replaces the receive processing of these messages with a direct invocation of the corresponding operations. This makes it impossible for threads to wait in the kernel for messages. Instead, such threads synchronize with the invoked operations. A special trick was used for the exception facility; the kernel would pretend that it was a user task while sending an exception RPC. This was the only circumstance in which the kernel initiated an RPC.

Neither of these techniques is suitable for returning the result of backing storage allocation. Implementing an operation to return the result requires a separate data structure and logic that serve no other purpose. Borrowing the identity of the user task that is creating the thread (and its kernel stack) is wrong in principle (the kernel should be allocating backing store for kernel stacks), and fails in practice for the creation of kernel threads. Instead we decided to extend Mach IPC to support kernel initiated RPCs (kernel RPC).

The implementation of kernel RPC marks ports to detect reply messages to kernel RPCs. When a thread in the kernel initiates an RPC, the reply port's *kernel_reply_port* field is set to mark it as a reply port for a kernel RPC. The IPC system delivers messages to such ports normally (e.g., completing the RPC) instead of attempting an operation invocation. This change only affects kernel initiated RPCs; RPCs directed at the kernel still invoke kernel operations as before. The reply port for a kernel RPC in progress is recorded in the data structure of the thread that is waiting for the reply; this allows the RPC to be aborted if

necessary. The kernel caches and reuses these reply ports when possible. The completion of an RPC frees the reply port to a cache which is checked when a new RPC is initiated. Reuse of a port from this cache requires passing a security check to ensure that no user task has surreptitiously queued a message on or retained a send right to the port; the use of send once rights in Mach 3.0 IPC will obviate the need for this check.

We also used kernel RPC to greatly simplify and improve the implementation of Mach's exception facility. The impersonation of a user task by the kernel causes the exception facility to contain a significant amount of code that understands IPC internals. For example, the exception facility must explicitly translate port names from the kernel to the impersonated task. Using kernel RPC in the exception facility allowed all of the offending code to be removed, and simplified the exception facility; now it need only find the appropriate exception port and initiate the RPC. Using kernel RPC also removed the side effect of the exception facility allocating its reply ports in the user task's port space.

3.4 Kernel and Vnode Pager Changes

Changes were required beyond adding and using the interfaces to allocate and deallocate backing store. The first was to handle system initialization. As the system boots, kernel thread stacks are allocated before the vnode pager task is initialized and well before a paging file has been established. Thus there is a collection of kernel stacks which do not have preallocated backing store.

There are two possible solutions to this problem: statically preallocate a fixed number of startup stacks or dynamically allocate secondary storage for existing stacks once the vnode pager is available. The former solution was rejected because of the more difficult problem of selecting an appropriate number of preallocated stacks for an operating system that is intended to run across a very large range of machines and configurations.

Instead of a static solution we chose to "record" all stack allocation and free requests and "play back" these requests once the pager is ready to receive backing store requests. The vnode pager signals its readiness when swapon() calls into the kernel with stack_init_backing_store(). This routine initializes the kernel object and replays the allocation/deallocation request queue. Should an allocation request fail then the unplayed requests are left on the queue. A subsequent swapon() can then satisfy these requests. Otherwise, some of the startup stacks will still be left with no preallocated backing store. The existing stack cache lock is used to protect transitions on the record queue.

The second change was made for performance reasons. The operating system already maintains a cache of thread kernel stacks. Stacks get added to this cache as threads are created until the cache limit is reached. Therefore the allocation and deallocation of backing store is not made when the thread is created and destroyed but when a new kernel stack is acquired or released via kmem_alloc() and kmem_free(). This means that the cached kernel stacks retain their backing store preallocation even if they are not being used. A possible future performance optimization is to allocate multiple kernel stacks when the stack cache is empty to amortize the cost of the backing store allocation.

One routine, stack_free_backing(), is added to support deallocation of backing store when a kernel thread is made unswappable. This is done under the assumption that once made unswappable, kernel threads are never again made swappable. This is enforced by a call to panic() in thread_swappable(). Freeing the backing store eliminates allocation of backing store that is never used. The backing store for user threads is not freed when a thread is made unswappable.

4. Clustered Paging Design

One of our performance studies measured paging and disk I/O statistics for a system that was forced into

steady state paging behavior. OSF/1 demonstrated a larger number of paging operations than a comparable operating system under similar conditions. The major factor leading to this behavior was OSF/1's use of single page I/O operations for pagein and pageout in comparison to the multiple page operations used by the other operating system. These single page operations limit the I/O bandwidth available to clean pages and service page faults. Our solution to this problem was to "cluster" page operations, i.e., transfer more than one page at once in order to improve I/O path utilization. Implementing this required three major sets of changes:

- the backing store page allocation policy was changed so that contiguous memory object pages are stored contiguously on the disk. This *clustered allocation* allows adjacent logical pages to be transferred in a single disk operation.
- when a dirty page is selected for replacement then *clustered pageout* will also select adjacent pages in the memory object to be cleaned in the same pageout operation.
- when a page is faulted in then *clustered pagein* will read adjacent pages in the same page read operation.

4.1 Clustered Allocation and the Vnode Pager

We first describe how paging works in OSF/1 R1.0 to provide background for the R1.1 changes. The OSF/1 R1.0 vnode pager associates pages of a temporary object with separate pages of backing store. All of the backing store pages for a single object are allocated from a single paging file. Paging files may be either raw disk partitions or regular files residing in file systems. Paging to raw disk partitions is more efficient, but unused paging space on a raw partition cannot be used by other files. OSF/1 R1.0 supports multiple paging files but cannot change the association of an object with a paging file after it is established. A simple mechanism is available to influence the selection of a paging file: a paging file can be declared as preferred when the swapon command is issued to add it to the system. Preferred paging files will be used before space is allocated from non-preferred ones.

There are two steps involved in establishing an association between an object and its pages on backing store. The first step associates a data structure in the pager with the object in the VM system. The VM system initiates this step by sending a memory_object_create() message to the pager. In response, the vnode pager sets up a data structure (called a *vstruct*) to represent the object and assigns a paging file to contain its pages. No allocation of pages occurs at this time. The second step involves the actual pageout of pages. The VM system initiates this step by sending a memory_object_data_write() message to the pager in order to page out a page. At this time the vnode pager allocates a page from the previously selected paging file and the page is written to the backing store. These backing store pages remain allocated until the object is destroyed and its resources are reclaimed.

These paging mechanisms have been generalized in OSF/1 R1.1. Allocation of pages occurs in units of clusters instead of single pages to better match the multi-page pagein and pageout operations that the VM system uses in R1.1. The allocation of paging file pages has been generalized to allow pages for a single object to be obtained from multiple paging files, and the preference mechanism has been extended to improve control over the use of paging file space.

4.1.1 Paging Files

The R1.1 code allows a single VM object to be automatically backed by more than one paging file. This replaces R1.0's associations of memory objects and paging files with associations of page clusters and paging files. Creation of these cluster associations is deferred until pageout of the cluster occurs; this automatically spreads paging activity across the available paging files and adds another use of lazy evaluation to those already present in the Mach virtual memory system.

A priority scheme is used to select paging files instead of the previous binary (preferred/not preferred) scheme. This priority is set by the swapon call that initializes a paging file. Multiple priority levels for paging files are a better match to multiple levels of secondary storage performance. For example, a system could be configured to select backing storage from a ram disk, followed by a parallel disk array, single disk drives, and finally from across the network as a last resort. The paging file selection algorithm is invoked each time a page cluster needs to be allocated (as a result of pageout). This algorithm cycles through all paging files of the highest priority, in round robin order, spreading allocations across them. When there is no more space available at a particular priority level then the files at the next available level are used.

4.1.2 Clustered Allocation

One of our design goals was to limit dependencies on how the kernel implements clustered paging. In particular, our design exhibits the following flexibility:

- Pagein/out requests may exceed paging file cluster size.
- Pagein/out requests can start on any page boundary within a cluster.
- Pagein/out requests may span cluster boundaries.

The design does require that the cluster size be a power of two multiple of pages. This allows us to track paging file page allocation on a per-cluster instead of a per-page basis, reducing the amount of state and simplifying the algorithms. We don't expect this to be a significant limitation in practice.

The vstruct (vnode pager data structure that represents an object) is the primary source of information about the location of the object's pages on backing store. This structure contains the size of a cluster in pages and a map of allocated clusters. Due to the use of lazy evaluation, this map is usually sparse. Each entry in this map indicates the paging file from which the cluster was allocated, the offset of the cluster within the paging file and a bit mask of valid pages in the cluster. The bit mask is necessary to support pageout of partial clusters and eager allocation of backing storage (which results in clusters that contain no valid pages). Our current implementation packs this information into a single 32 bit field to minimize the space occupied by these data structures. The boundaries between the components of this field are variable; here is the layout we use by default:

#bits	5	1-8	19-26
field	paging file	bitmap	cluster offset

The paging file field is an offset into an array of paging file pointers. A field size of five bits limits the number of paging files to 32. The bitmap field must be sized to match the number of pages in a cluster; the current implementation allows cluster sizes ranging from one to eight pages. The remaining bits are used for the offset of the cluster within the paging file. The cluster offset in the paging file varies inversely with the bitmap size.

The flexibility and scalability of our implementation are enhanced by severely limiting the number of routines that have knowledge of this data structure. This makes it easy to adjust the boundaries among the fields (e.g., to allow a larger number of paging files). It is also straightforward to change to a new representation that uses more space (e.g., to scale to larger quantities of backing store), because the internal interfaces have been abstracted so that they do not depend on this representation.

Our current implementation uses a system-wide default for the cluster size for all paging files and objects in the system. This optimizes the use of space, but the design allows paging file cluster sizes and VM object cluster sizes to vary. The assignment of cluster size to a paging file occurs at paging file creation; it is an optional argument to the swapon command, and defaults to eight pages. In turn this default can be set by a system administrator or vendor to 1, 2, 4, or 8 pages. When paging from an actual file instead of a

paging file (e.g., to support mapped file functionality), the cluster size is set to match the filesystem block size, because this is likely to be the size of contiguous allocations in the filesystem. For example, if the filesystem block size is 8k, and the vm page size is 4k, then the cluster size will be set to two pages. This policy can be modified to depend on filesystem type.

4.2 Clustered Pagein

Clustered pagein retrieves a faulted page and some number of adjacent pages in a single read operation. Clustered pageins improve performance by taking advantage of locality of reference; there is a good chance that there will be page faults in the near future on pages close to the initially faulted page. Retrieving these pages in advance avoids the need to access disk when these faults occur. In the absence of locality of reference, the performance degradation is minor; the scarce resource in paging is disk operations, and clustering of pageins does not involve additional disk operations. The major source of performance improvement comes from amortizing expensive disk transfers over multiple pages.

No changes to the existing EMMI interfaces were required to support clustered pagein, as *memory_object_data_request()*, *memory_object_data_provided()*, *memory_object_data_unavailable()*, and *memory_object_data_error()* already support multiple pages. The implementation of these routines and related code was improved by incorporating Mach 3.0 techniques into OSF/1 R1.1. These techniques are known as "fictitious pages" and "page stealing." In addition, we retained and enhanced a performance improvement introduced in R1.0, "short circuiting."

4.2.1 Fictitious Pages

Fictitious pages are used in the fault path to improve physical memory utilization. OSF/1 R1.0 used Mach 2.5's technique of allocating physical pages in the fault path to block subsequent faults and reserve the memory that will eventually satisfy the fault (via a pagein, page copy, or zero fill, depending on the fault). The need to block faults reaching an object via different paths may require the allocation of two physical pages to satisfy a single page fault. To reduce this use of physical memory R1.1 employs Mach 3.0's technique of allocating fictitious pages as placeholders in the fault path. A fictitious page consists of just a vm_page data structure, without any associated physical memory. Using fictitious pages in the fault path defers allocation of physical memory until it is actually needed to satisfy the fault. This reduces the use of physical memory at the cost of a small increase in the complexity of the resident page management code in the virtual memory system.

4.2.2 Page Stealing

Page stealing logic transfers pages directly from the pager to the destination object without performing a copy. To use page stealing effectively, the pager must allocate its own buffers for retrieving data, and allow the data to be deallocated as a side effect of the pagein. If the pager wishes to retain the data in its address space after the pagein, then the pages cannot be transferred and must be copied. The actual data transfer swaps the pages for the fictitious pages serving as placeholders, and frees the fictitious pages. A new interface, *memory_object_data_supply()*, is used in place of *memory_object_data_provided()* to enable page stealing. Due to a limitation of the Mach 2.5 version of MiG used in OSF/1, memory_object_data_supply() always deallocates the paged-in buffer; memory_object_data_provided() is still available for pagers that wish to retain copies of paged-in data.

4.2.3 Short Circuiting

The short circuiting optimization allows faulting threads to directly execute the vnode pager's pagein code. This addresses a major problem with older versions of Mach (including that used as the base for R1.0), namely that the pagein path always makes a copy of the page being paged in. Short circuiting

eliminates this copy by paging in directly to the page reserved by the page fault logic. Short circuiting also eliminates the exchange of messages and context switches required to invoke the vnode pager, and enhances throughput by allowing non-vnode pager threads to execute the pagein code. Although page stealing eliminates the page copy (reducing the performance advantage of short circuiting), we chose to reimplement short circuiting in R1.1 to retain its other advantages.

Short circuiting is not used for pageout because the Mach pageout daemon is a single thread. By transferring responsibility for performing the pageout, a single pageout thread can keep multiple pager threads (and hence paging files) busy, achieving higher bandwidth than would be possible with a single thread. There is also an important robustness advantage to this transfer. Some paging files (e.g., those accessed over NFS) are prone to temporary inaccessibility. By delegating a different vnode pager thread to actually access the paging file, the pageout thread is insulated from this type of problem.

4.3 Clustered Pageout

Combining multiple pages in a single (clustered) pageout operation can save write operations to backing store. Clustering combines adjacent dirty pages with the original candidate page for pageout and writes the collection to backing store in a single operation. After this write, only the original candidate would normally be freed; the other pages retain their positions in the pageout queues. If these pages remain clean until they become candidates for page replacement, then they can be freed instead of being written; this saves write operations. As in the pagein case, the scarce resource is disk operations, so the added overhead of writing a cluster instead of a single page is minor even if the additional pages become dirty and require their own write operations.

It was much more difficult to implement clustered pageout than pagein. This was caused by the complexity of the Mach pageout path and by assumptions in both the pager and pageout daemon that pageout operations only involved single pages. A major contributor to the complexity is the asynchronous nature of pageout; this required additional state to track pageouts in progress for both the kernel and pager. We also found it necessary to introduce a new method for pageout of dirty pages, cleaning in place, and discovered that careful coding was required to ensure that the interactions between clustered paging and the page queues did not disrupt the approximate LRU algorithm used to select pageout candidates.

4.3.1 Cleaning in Place

It is necessary to first understand how OSF/1 R1.0 pageout works before explaining the need for cleaning in place. When the system needs to move pages to the free list the pageout daemon selects the oldest page from the inactive queue for replacement. If the page has been referenced while on the inactive queue then it is reactivated. If the page is not dirty, then it is immediately placed on the free list. Otherwise, the page must be cleaned (paged out) before it can be released.

Pageout is implemented by moving the page to a new VM object. This page is temporarily replaced in its original object by a busy fictitious page to prevent faults on this page from causing pageins before the pageout message is sent. The real page is moved into a new VM object, created for the purpose of this pageout, and sent to the pager. The fictitious page is then released. The new VM object is used primarily to protect Mach from recalcitrant user-written pagers. The system's default pager backs the new object, and can write the pages to its backing store if a user-written pager is unwilling to do so. This level of protection is not necessary for interactions with the default pager itself.

This replacement mechanism is not appropriate for clustered pageout. Making the entire cluster of pages inaccessible for the duration of the pageout has a number of undesirable effects. It increases the cost associated with these pageouts, and also effectively increases the system's page size. The major problem is that this risks selecting the wrong pages for pageout since other pages in the cluster could be much more active than the original candidate selected for replacement. Instead, we changed the implementation of

pageout in R1.1 to clean pages in place without moving them into a separate VM object. The adjacent pages in the cluster are left in the original VM object and remain accessible (the target page also remains in the VM object, but becomes inaccessible because in order to guarantee that it will be successfully cleaned and freed). All of these pages are marked as unmodified before the pageout is initiated; the alternative, waiting until pageout completion to do this, risks losing modifications made while the pageout was in progress.

Cleaning in place cannot be used with untrusted pagers because it eliminates the temporary object backed by the default pager. Our implementation does not perform clustered pageout to an untrusted pager, but instead uses the existing Mach single-page mechanisms to retain protection from recalcitrant pagers. An additional test has been added to the pageout code that allows the vnode pager to be considered trusted for both internal (default pager) and permanent (file) objects.

Two additional flags were added to the VM page structure to detect intermediate states in the new pageout process, the "cleaning" and "pageout" flags. The "cleaning" flag marks a page for which a pageout is in progress. This avoids initiating duplicate pageouts on these pages, and allows the operation that completes pageout to detect that it is manipulating the correct pages. A new call, *memory_object_data_write_completed()*, was created so that the pager could inform the VM subsystem of pageout completion, so that the "cleaning" flag can be cleared for the pages involved. The "pageout" flag identifies a target page for pageout (this is a page that has been selected for cleaning, and is therefore not on the pageout queues). The memory_object_data_write_completed() operation frees such pages if they are clean; otherwise it returns them to the pageout queues. If a non-target page (adjacent to the original target page in the cluster) reaches the end of the inactive queue before its pageout completes, its "pageout" flag is set to mark it as a target page. The cleaning flag is also used to synchronize cleaning in place with copy-on-write.

The pages in the cluster cannot be deallocated while they are in the process of being cleaned. To prevent this the pageout daemon takes a paging in progress (software) reference on the object's data structure for each page being cleaned. These references are released as part of the memory_object_data_write_completed() call. vm_object_deallocate() synchronizes with the release of these paging in progress references.

4.3.2 Effect on LRU Approximation

Mach uses the active and inactive queues to approximate an LRU page replacement algorithm. The pmap module is responsible for implementing a reference bit per physical page that can be cleared and checked. The pageout daemon clears this bit when placing a page on the inactive queue, and checks it when the page reaches the end of the queue. If the bit is set, the page is returned to the active queue instead of being paged out or freed. This reference bit should not be confused with the modify bit that is set only by writes to the page.

Mach's association of the reference bit with the physical page instead of the virtual to physical mapping (e.g., as in 4.2BSD) is a potential source of problems. Systems that associate the reference bit with the mapping can distinguish between accesses that use different mappings; this allows such systems to ignore references made by their pageout mechanisms by using a separate mapping. Investigation revealed that the vnode pager in OSF/1 R1.0 would always set the reference bit in the process of paging out the page. This is because that pager wires and unwires pages, and may actually reference the data (typically on machines that lack DMA hardware).

This behavior seriously distorts the LRU approximation. Consider the situation in which a group of pages is paged in and modified, then not used for a long period of time. When the first of these pages reaches the end of the inactive queue, it will be selected for replacement, and cleaned. The adjacent pages (which are near the end of the inactive queue) will also be selected to be cleaned as a cluster. At the completion of the cluster write, the target page is freed, but the other pages are reactivated (because the pager set their

reference bits). Rather than being replaced soon (since they are inactive) they are now the least eligible pages in the system for replacement. To prevent this behavior the vnode pager was modified to not wire the pages into its address space (this also shortens the pageout code path).

This has repercussions for those doing OSF/1 ports. Their ports must be examined to ensure that the act of performing a disk write from the memory region set up by the pager does not cause any fault on the pages or any change to the reference bits implemented by the pmap module. Otherwise clustered paging will distort the LRU page replacement algorithm.

5. Swapping

When the total demand for page frames, i.e. the system-wide working set, exceeds the total available resident memory space then the amount of useful work accomplished between page faults falls. This causes an increase in the amount of system resources dedicated to moving pages between backing store and primary memory, with a corresponding decrease in available CPU cycles to perform useful work. As the problem gets worse, it is even possible to fetch pages into main memory and then discard them prior to their being used even once. This awkward degradation of demand-paged virtual memory systems is called thrashing. A thrashing system is spending too large a fraction of its available resources on paging, and too small a portion on actual computation.

OSF/1 R1.0 makes no attempt to manage the demand for page frames, and is therefore subject to thrashing under even moderate loads. To correct this, we have implemented swapping: the detection of excessive page demand and the selection and suspension of tasks to reduce that demand. This section describes how the Mach based OSF/1 operating system was modified to add swapping. The discussion is divided into sections on swapping policy and mechanisms. Much of the motivation for this work arose from the swapping implementation in 4.3BSD Unix.

5.1 Swapping Policy

Swapping addresses the following problems in the 4.3BSD Unix system:

- The system page map becomes fragmented, preventing processes from allocating space for page tables.
- Processes are inactive for more than 20 seconds, but continue to consume page table resources.
- The paging rate is excessive, so the pageout daemon cannot clean pages fast enough to meet demand.

The first two problems do not exist in OSF/1, because the Mach VM system uses different data structures and allows resources (e.g., page tables) to be reclaimed. Fragmentation is prevented by the use of appropriate resource management techniques in the pmap module. (Note: it is possible to write a pmap module to create this problem but a well written pmap module will not fragment.) The consumption of page table resources by idle processes is prevented by the thread swapper. The thread swapper scans the all threads list looking for threads that have been idle for at least 10 seconds. Such threads have their kernel stack unwired, and the pmap module is invoked to reclaim the thread's page table resources.

The problem that was not addressed in R1.0 is excessive demand for physical memory. When the total working set of all active tasks exceeds the amount of memory available, the system needs to prevent some tasks from running. Two policy questions must be answered when swapping becomes necessary: which tasks should be swapped out and when should swapped tasks be swapped back in?

Our swapout decisions are based on the algorithm used in 4.3BSD Unix. That algorithm maintains resident set size and memory residence time information for each process. Of the four processes with the largest resident set size, the one that has been resident in memory the longest is chosen as the victim to be swapped out. We chose this algorithm based on its simplicity and the experience with its use in 4.3BSD

Unix.

The OSF/1 R1.1 pageout daemon initiates task swapping when the free page count remains low despite the pageout daemon's best efforts. Paging activity in the pageout daemon is controlled by two thresholds on the amount of free memory, a minimum (typically 1% of memory) and a target (typically 1.25% of memory). The pageout daemon is invoked when the amount of free memory drops below the minimum threshold, and remains active until the target threshold is reached. The higher target threshold improves the utilization of the pageout daemon by invoking it less often to do more work each time. Task swapping is invoked when the amount of free memory remains both below the minimum threshold for 5 seconds and below the target threshold for 30 seconds. These delay times were also taken from 4.3BSD Unix, but are easily modified.

The task swapper chooses a task to swap out by using the 4.3BSD Unix swapout algorithm. Of the largest tasks in the system, the one that has been resident the longest is selected for swap out. The current resident set size of the task is used as an estimate of the number of pages that will be freed by this operation. (A better estimate would be to use the non-shared resident set size, but this number is currently not maintained by the pmap modules.) Tasks swapped by this swapper are *involuntarily* swapped; they will not be permitted to run again for some period of time. A task can also be swapped out *voluntarily* when the thread swapper swaps all of the task's threads.

There are two ways a non-resident task can be swapped in: a voluntarily swapped task will be swapped in when any of its threads are awakened as a side effect of swapping in the thread. An involuntarily swapped task will become eligible for swapin when the memory shortage has been relieved sufficiently to allow the task to make progress (based on the task's resident set size at swapout time) or when the task has been non-resident for longer than an adjustable threshold. If swapping in a task to prevent starvation, the swapper may choose to swap out another task first. A list of all involuntarily swapped tasks is maintained to implement this time based swapin algorithm.

When implementing these swapping policies we made a strong effort to separate policy and mechanism. We recognize that there are a wide range of possible swapping policies and that system vendors will will need to adjust or replace this policy when tuning OSF/1 for their particular systems.

5.1.1 Additional Statistics

Some statistics needed to be added to the Mach kernel in order to support the policy decisions. These statistics had to be added to the thread, task, and Unix data structures.

The kernel already tracks a set of system-wide virtual memory statistics such as pageins, copy-on-write faults, and page reclaims. To supplement the existing statistics an events_info structure was added to the thread and task structures. The thread statistics are incremented when the kernel updates the global statistics. Per thread statistics are accumulated in the per task statistics at thread exit. In addition, the cumulative statistics for all terminated child processes of a (Unix) process is recorded. This data is stored directly in the u.u_cru structure in the U-area (utask). This information is now available to user space via the rusage structure and the task_info() and thread_info() Mach calls.

5.2 Swapping Mechanism

The swapping mechanisms implement swapping policy decisions. The swapout mechanisms are responsible for suspending the execution of victim tasks and reclaiming appropriate resources from them. The swapin mechanisms are responsible for allowing task execution to resume and resources to be restored. This section describes the implementation of these mechanisms in OSF/1 R1.1. We begin with a brief description of the virtual memory data structures that are involved.

5.2.1 Virtual Memory Data Structures

This is a brief summary of the Mach virtual memory (VM) data structures. The reader should already be familiar with Mach VM but this section is provided as a convenient reference. An understanding of the relationships between VM data structures is necessary in order to explain the swapping mechanisms.

Although the figure shows the most common relationships between VM components it is by no means complete. One could add to the complexity by including additional tasks pointing to the sharing map, a more complicated shadow object chain, and copy objects. However, this diagram provides enough support for the following discussion.

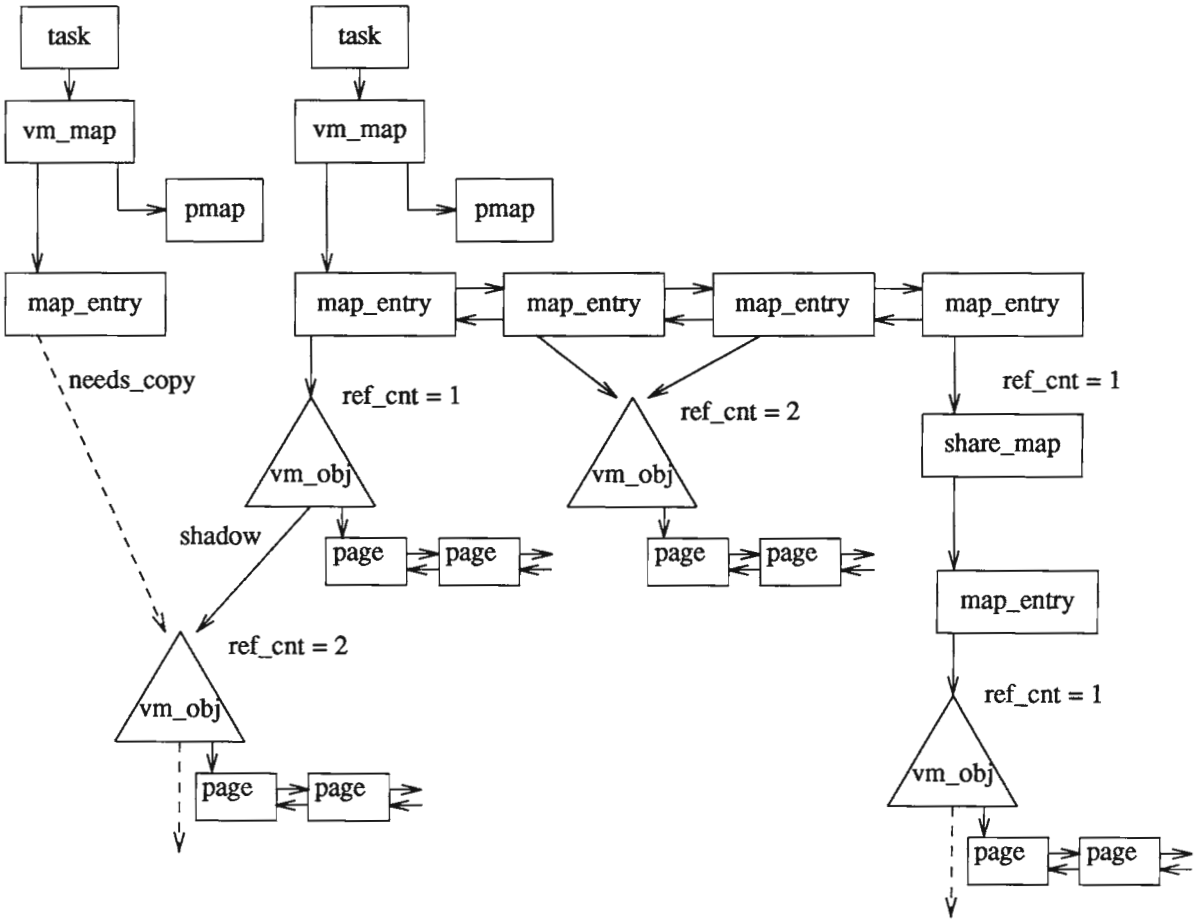

Figure 1. Relationship between VM data structures

A task's virtual address space is described by a *vm_map*. The vm_map has access to the physical mapping for that task and the linked list of map entries that describe each virtual memory region. Each *map_entry* points to either a *vm_object* or a *share_map*. The vm_object represents the set of pages that are mapped to the corresponding region. The vm_object keeps a linked list of the region's resident pages, a port that is used to communicate with a memory manager for non-resident pages, and a possible pointer to other vm_objects which describe additional pages within the region. A map_entry can also point to a *share_map* in the case where multiple tasks are sharing memory. The share_map points to its own linked list of map entries and their vm_objects.

Sharing of physical pages can occur in a variety of ways with these data structures. A sharing map causes a portion of address space to be shared among two or more tasks. An object can be independently mapped

into the address space of more than one task. Often the sharing is indirect due to the use of virtual copy optimizations; objects mapped into two different tasks may be copies of a single underlying (or shadowed) object. Multiple levels of such shadowing are possible (and occur in practice).

5.2.2 Reclaimed Resources

We chose to reclaim only a critical subset of the resources that could be reclaimed from a swapped task. Among these resources that could be reclaimed are: task state (including VM structures), thread state, thread stacks, Unix process state, task resident pages, and the task's page table (pmap). We chose to focus on thread stacks, the task's resident pages, and the task's pmap. This selection gave us the largest benefit for the initial swapping implementation.

One reason for not attempting to reclaim the smaller structures associated with a swapped task is that many of them consume a fraction of a page that is shared with other unrelated structures. Writing the contents of these structures to backing store and freeing them back to their respective allocator would provide only a marginal benefit because it is unlikely to free the entire page. Beyond this, many allocators in the kernel retain a pool of available pages or structures instead of aggressively releasing free memory to the page pool.

In OSF/1 the BSD style U-area has been split into two pieces, the utask and the uthread. The utask structure holds those task specific fields which were once part of the U-area and the uthread structure contains the thread specific fields. The uthread is stored in a thread's kernel stack so it will automatically be reclaimed when the thread's kernel stack is reclaimed. The utask structure and the logically connected per-process file table are very good candidates for future reclamation.

5.2.3 Resource Reclamation

When examining the VM structure relationships diagram one can identify three levels: the task, map, and vm_object levels. Changes had to be made to each of these levels in order to reclaim a task's resources.

5.2.3.1 Task Swapping

The changes at the task level consist of some minor additions to the task structure and the addition of three routines. We added a state variable, swap_state, to the task structure. This has six states that are analogous to thread swap states. A queue chain is used to link together all swapped tasks. A timestamp is added to record the last time the task was swapped in or out. When the task is swapped out the resident set size is saved. This size is later used as part of the swap in criteria.

The first of the three new routines, *task_swappable()*, is used to indicate that a given task cannot be swapped. This is used for tasks associated with the pageout path. Among the non-swappable tasks in R1.1 are the first task, the vnode pager tasks, the device pager, the exception handler, and the NFS client tasks.

Task swapping is implemented by a task swapping daemon. This is similar to the pageout daemon except that it swaps tasks instead of paging out pages. When this daemon finds an appropriate victim task to swap, it calls *task_swapout()*.

```
void
task_swapout(task_t task)
{
    record resident set size in task structure

    task_hold(task);        // suspend all threads
    task_dowait(task);      // wait for all threads to stop

    add task to list of non-resident tasks

    // start swapping threads
    thread = task->thread_list;
    while (thread)

        // mark thread swapped, swapping out.
        thread_swapout(thread);
        thread = thread->thread_list;
}
```

Figure 2. task_swapout pseudocode

There are three steps involved in swapping a task, suspending its threads, swapping them, and swapping the task's memory. Task_swapout first suspends all threads with the task_hold and task_dowait routines. (These routines were already present in R1.0.) These increment the suspend count for all the threads in the task and then wait for them to actually suspend. Task_swapout then swaps out all the threads. This unwires the threads' kernel stacks, making the eligible for pageout. Thread_swapout also tracks the number of swapped threads in a task; when the last thread is swapped, the pmap module is invoked to reclaim the task's page tables, and vm_map_res_deallocate() is called to swap out the task's resident pages. This routine is discussed in detail in the next section.

The third new routine is *task_swapin()*, the inverse of task_swapout(). To swap in a task, the task is marked not swapped and the suspend count on all threads is reduced via task_release(). Task_release() then makes all threads with a zero suspend count runnable. When the first thread becomes runnable the scheduler will make the task's map resident by calling vm_map_res_reference().

The swapin time is saved in the task structure so that the swapout policy can calculate the residence time (and ensure that a swapped in task remains swapped in for some minimum time period). Swapping in a task does not involve explicitly bringing pages from secondary store into main memory. The task will begin faulting in its working set as soon as it is made runnable. Clustered paging improves the efficiency of restarting a swapped task.

5.2.3.2 Map and Object Swapping

This section covers the swapping of the lower layers in the VM system, the map and object layers. We discuss these layers as a unit because similar mechanisms are used for both. In reading this discussion it is important to understand that the the map and object data structures themselves are not swapped. Rather these structures are traversed and modified to allow the contents of resident pages to be swapped out and the page tables to be reclaimed. Actually swapping out the map and object data structure is a topic for potential future work.

Traversing the complex relationships between the VM structures is the most difficult aspect of designing the OSF/1 swapping mechanisms. The design is greatly simplified by relying on the Mach techniques of reference counts and lazy evaluation.

A resident count field is added to both the vm_map and vm_object structures. Thie field parallels the reference count; every holder of a reference on the data structure (i.e., has incremented the reference count) may optionally hold a resident reference (i.e., has incremented the resident count). The map or object is considered swapped in only when the resident count is non-zero (i.e., 'swapped out' maps and vm_objects always have zero resident counts). The act of swapping a task's pages consists of decrementing the resident count on the task's VM map that corresponds to the reference held by the task data structure. When the resident count becomes zero, the algorithm proceeds to the next layer and repeats; each map and object structure in the next lower layer has its resident count decremented. This recursion terminates when a non-zero resident count (after decrementation) is found or when the bottom is reached. When a vm_object's resident count reaches zero, those pages are not in use by any active thread are deactivated. The table below summarizes the R1.1 routines which affect the vm_map and vm_object resident counts.

Map Swapping	res	ref	Object Swapping	res	ref
vm_map_create	1	1	vm_object_allocate	1	1
vm_map_res_deallocate	-	nc	vm_object_res_deallocate	-	nc
vm_map_deallocate	-	-	vm_object_deallocate	-	-
vm_map_res_reference	+	nc	vm_object_res_reference	+	nc
vm_map_reference	+	+	vm_object_reference	+	+
vm_map_swapin	+	+			

- == decrement 1 == assign one
+ == increment nc == no change

Figure 3. Resident and Reference Counts

The resident and reference counts match unless swapping has occurred. This is because the routines that change only one count but not the other (the *_res_reference and *_res_deallocate routines) are only called by swapping code. The normal operations that create, deallocate, and reference maps and objects make the same changes to both counts.

The swapping algorithm implements a conservative approach to swapping shared memory. Such memory is swapped when the last task sharing it is swapped. This behavior depends on the use of references by the Mach kernel. In a quiescent state (no operations in progress) a top-level (task) VM map will have exactly one reference, the reference held by the task data structure. All other sharing maps and objects will have one reference from each higher level object that points to them (i.e., shares their contents). The maintenance of resident counts in each data structure ensures that a resident page can only be forced out by swapping when all tasks into which it may be mapped are swapped. If any such task is not swapped, its VM map will have a non-zero resident count, and hence all sharing maps and objects below that task's VM map will also have non-zero resident counts. This bookkeeping for shared regions is the source of most of the complexity in the swapping implementation, but we felt that taking this approach to shared memory was important. For example, the C library is shared in OSF/1; it is important that the library remain swapped in while any task is using it.

A consequence of this is that any operation that takes or releases a map reference must be prepared to cause the corresponding swapin or swapout. The fact that these are blocking operations causes a difficulty with the code that manages vm_maps. Map references are often taken in circumstances where the code is not prepared to block; these circumstances are always ones in which the code already holds a reference on the map and is cloning the reference. This cloning of a reference can never cause a swapin. To identify such clonings and make this assumption explicit, we use different routines for this case and the case of taking a reference that can cause a swapin; both routines have the effect of incrementing both the resident and reference counts. We used vm_map_reference for the cloning case (swapin not possible) since this corresponds to most existing uses of that routine. For the cases in which taking a reference may cause a

swapin, the new *vm_map_swapin* routine is used. An important example of such a case is the reference taken by convert_port_to_map() when the invocation of a Mach VM operation crosses a task boundary (i.e., the invoking thread is not in the task affected by the operation). This reference ensures that a VM map is considered swapped in while an operation is in progress on it.

6. Conclusion

One of the more important conclusions of our work is that the Mach virtual memory subsystem is flexible enough to allow these large functional changes without requiring major overhaul of the underlying design. Also, we feel that these robustness and performance changes to the virtual memory subsystem have greatly improved the commercial viability of the OSF/1 operating system. The robustness changes, deadlock removal, and eager allocation of kernel stacks, have increased system availability and preliminary performance measurements of the clustered paging changes show that larger numbers of pages are being transferred with each I/O operation, as expected. Complete performance numbers were not available at the time this paper was written. We expect to have performance results at the time of the conference.

References

[1] David L. Black, Avadis Tevanian,Jr., David B. Golub, and Michael W. Young, "Locking and Reference Counting in the Mach Kernel", Proceedings of the 1991 International Conference on Parallel Processing, August 1991, pp. II-167 - II-173.

[2] S.J. Leffler, M.K. McKusick, M.J. Karels, J.S. Quarterman, "The Design and Implementation of the 4.3BSD Unix Operating System", Addison-Wesley, Reading, MA (1989).

[3] R. Rashid, A. Tevanian, Jr., M. Young, D. Golub, R. Baron, D. Black, W.J. Bolosky, and J. Chew, "Machine-Independent Virual Memory Management for Paged Uniprocessor and Multiprocessor Architectures", IEEE Transactions on Computers, Vol. 37 No. 8 (August 1988), pp. 896-908.

[4] Avadis Tevanian, Jr., "Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments: The Mach Approach", PhD thesis, CMU-CS-99-106, Dept. of Computer Science, Carnegie-Mellon University, December, 1987.

[5] Jim Van Sciver, Ping Wang, "OSF/1 (1.0.1s+) System Memory Usage", OSF/1 Release 1.1 Engineering Note, Open Software Foundation, May, 1991.

[6] Michael Wayne Young, "Exporting a User Interface to Memory Management from a Communication-Oriented Operating System", PhD thesis, CMU-CS-89-202, Dept. of Computer Science, Carnegie-Mellon University, November, 1989.

[7] M. Young, A. Tevanian, Jr., R. Rashid, D. Golub, J. Eppinger, J. Chew, W.J. Bolosky, D. Black, and R. Baron, "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System", Proceedings of the Eleventh ACM Symposium on Operating Systems Principles, Vol. 21, No. 5 (November 1987), pp. 63-77.

[8] "The Design of the OSF/1 Operating System", Book in Progress, Open Software Foundation.

[9] "Guide to OSF/1: A Technical Synopsis", O'Reilly & Assiciates, Inc., 1991.