User-Level Physical Memory Management for Mach

Stuart Sechrest
Yoonho Park

Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, Michigan

Abstract

We have developed an extended version of Mach 3.0 that allows physical memory managers to run as user-level processes, and which allows the memory requirements of these managers to be balanced. Physical memory is a resource for which there are a number of potential competitors whose diverse uses of physical memory may require diverse management policies. Flexibility in physical memory management policy is important to database managers, multimedia file systems and persistent object storage managers. Our architecture allows the control of physical memory page frames to be assigned to processes running outside the kernel, with page frame records shared between the kernel and these managers. Control of page frames can be reassigned among physical memory managers by a balance manager.

1 Motivation

Physical memory is a resource for which there are a number of potential competitors. RAM is used not only as as backing store for processes' virtual address spaces, but for caches maintained by file systems, persistent object stores, and database systems, as well. The diverse uses of physical memory may require diverse management policies. Flexibility in physical memory management has been a concern of database researchers [11, 1] for a number of years. We believe it will become a matter of increasing concern in multimedia file systems and persistent object storage. As Stonebraker [12] points out, buffer managers may be able to take advantage of semantic knowledge about the data to make more effective use of physical memory. An image data manager can, for example, retain images at reduced resolution, rather than flushing them entirely.

We believe that a physical memory management system must balance the requirements of these competitors for memory, while allowing effective management policies to be implemented. Mach 3.0 does not address this need. We have therefore developed an extended Mach 3.0 system that allows multiple physical memory

managers to run as user-level processes, and allows the amounts of physical memory assigned to these managers to be balanced dynamically.

Mach 3.0 maintains a list of free pages selected by a version of the global clock page-replacement algorithm. This algorithm seeks to locate and free the least recently referenced (global LRU) pages. This algorithm is widely used, but may make inappropriate selections for particular types of data. It has been suggested, for example, that for joins in relational database queries, one may want to replace the *most* recently used page within a relation [1]. The division of physical memory into pools can be used to protect one process from the paging of another, as in the VAX/VMS operating system [4]. Our intention, however, is to use pool boundaries to protect *subsystems* from one another and alter these boundaries through explicit balancing operations. This is particularly important when the cost of storing data to and retrieving data from permanent storage can differ significantly.

2 Related Work

Current systems generally have limited interfaces allowing a user to affect page replacement choices. Mach 3.0 provides a call that allows certain pages to be pinned in memory. SunOS 4.1 provides the call *madvise()*, allowing applications to advise the kernel on the likely pattern of accesses in a region of the application's address space [7]. The application can state that access will be random or sequential, or that certain pages should be held and others released. The kernel can take these claims into consideration while implementing page replacement.

The QuickSilver operating system [3] placed important physical memory management responsibilities outside the kernel [14]. The QuickSilver kernel is extremely small. Management of physical memory is therefore entrusted to a server process. The page tables are shared between the kernel's and the server's address spaces. To implement page replacement the server invokes a kernel call which locates the (approximately) least recently used pages using a clock algorithm. Thus, only one page replacement policy is supported.

PREMO [6], an extension of Mach 2.5, allows a user-level program to implement its own page replacement policy by extending the external pager interface. The external pager is consulted when one of its pages is to be placed on the free list, and is allowed to substitute an alternative page. While this approach allows the implementation of alternative replacement policies within a pool of pages controlled by an external pager, the selection of the pool from which the victim is to be drawn is still based on a global LRU strategy. The PREMO approach does not take into account the differences between pools and may allow some pools to be unfairly victimized. Sprite [8] began to address the problem of balancing competing claims on physical memory by dividing memory between backing store for virtual

address spaces and file system buffer cache, but dynamically adjusting this division. Performance was improved, provided that the virtual memory system was given preference over the file system. The Sprite approach, however, was limited to two pools, and it did not allow the implementation of new replacement policies in any simple way.

3 Architecture

3.1 Physical memory management in Mach 3.0

In Mach 3.0 the majority of the physical memory of the system forms a single paging pool used as a cache for memory objects [5]. Page frames are represented within the kernel at two levels of abstraction. Machine-dependent details of the host memory architecture are hidden beneath an idealized *pmap* interface. Machine-independent records for page frames are kept in the resident page table. The kernel code responsible for handling page faults and for managing physical memory manipulate the resident page table information and make calls through the pmap interface.

The system virtual memory cache is managed by the kernel page-out daemon using an approximate LRU algorithm. With the exception of the *vm_pageable()* call, a privileged call that pins pages in memory, user-level programs have no control over this algorithm. The system page-out daemon implements a global clock algorithm. The available page frames are placed on an active, an inactive, or a free list. The page-out daemon seeks to maintain the list of free page frames, by moving page frames that have not been recently used from the active to the inactive list and from the inactive to the free list. In doing so it relies on calls through the pmap interface to access hardware maintained access information.

In addition to ordinary paging, page frames can be removed from the free list for use as storage for kernel structures [9]. These pages are freed by placing them on the active list, where the page-out daemon will eventually determine that they can be freed.

3.2 User-level PODs

We have developed an extended version of Mach 3.0 that divides the physical memory into a number of pools, managed separately by processes called PODs (for Page-Out Daemons). Because we want to explore physical memory management policies for new areas such as multimedia file and database management systems, we want to simplify the creation and configuration of systems implementing diverse physical memory management policies. PODs therefore run as (trusted) user-level programs, sharing access to kernel data structures (see Figure 1). As noted above,

the resident page table in the Mach 3.0 kernel is threaded with lists maintained by the kernel's page-out daemon. We allow the page table to be mapped into the address spaces of user-level PODs. PODs can also run as part of the kernel, although kernel PODs are not required. A *balance manager* running in the kernel shifts page frames between the pools maintained by the various PODs.

Page faults on memory objects are handled by the kernel. Every memory object is assigned to a POD, and its page faults will be satisfied from this POD's page frame pool. It is the responsibility of the POD to maintain a list of free page frames within this pool. Figure 2 illustrates the relationship of memory cache objects, POD records and the resident page table. The pool of page frames associated with a POD need not (and in general will not) be contiguous. Instead the records for the frames in a POD's pool appear on one of two lists whose heads are in the POD record. The *in-use list* contains page frames known by the POD to be in use, while the *free list* contains the remainder of the pool's frames.

A page frame on the free list is either *free* or *allocated.* (Figure 3 shows the state transition diagram for a page frame.) The kernel allocates pages from a POD's free list. The new state of these page frames is recorded in the resident page table. These pages, however, are not moved automatically to the in-use list. For short-lived objects, these page frames may quickly be freed again. Their state recorded in the resident page table will revert to free.

The state of a page frame on the in-use list is either *in-use* or *must-free.* When the POD is awakened, allocated page frames are removed from the free list and placed on a transfer list. This list is passed (by reference) to the POD, which incorporates them on its in-use list. Their recorded state is changed to in-use. The kernel may deallocate in-use page frames if, for example, an object terminates. In this case they are left on the in-use list, but their state is changed to must-free. Must-free pages and page frames deallocated by the POD are removed from the in-use by the POD and placed on a transfer list. This "laundry list" is passed (by reference) to the kernel.

The POD is awoken by the kernel both periodically and when its free-list falls below an agreed upon low-water mark. In the extreme case, the POD can set the low-water mark to empty and, thus, arrange to be called for every page frame assignment. Ordinarily, however, page frame assignments can be made from the free list without requiring a call to the POD. It is the POD's responsibility to ensure that the free-list size exceeds the low-water mark. When invoked the POD returns to the kernel a list of page frames to flush along with any must-free page frames. The kernel must remove the flushed pages from the memory objects to which they belong and, possibly, invoke external pagers to save modifications. The policies and data structures used to determine which page frames to flush are entirely the concern of the POD.

The architecture outlined here has certain assumptions that should be stated ex-

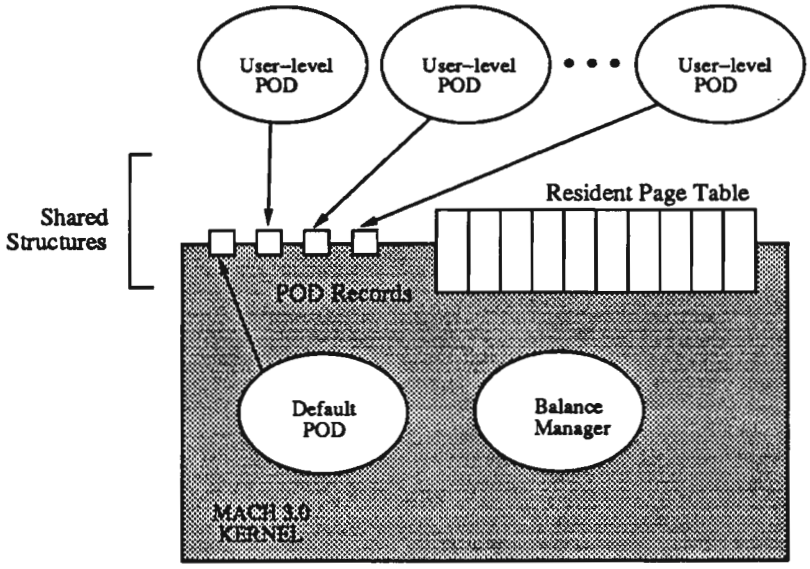

Figure 1: User-level physical memory managers (PODs) share access to kernel data structures

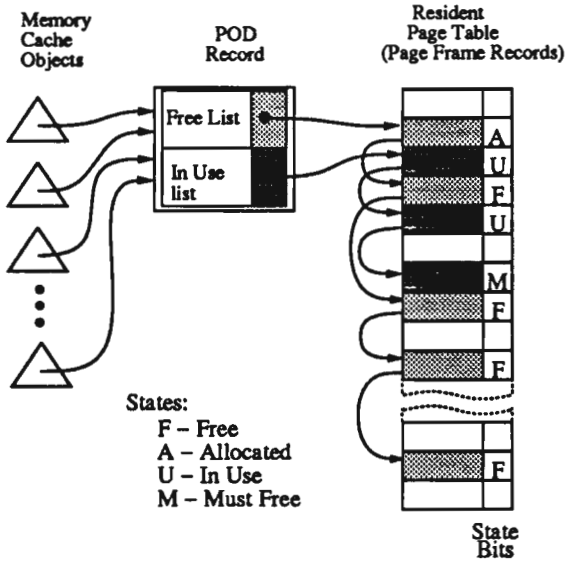

Figure 2: Memory architecture for user-level physical memory management

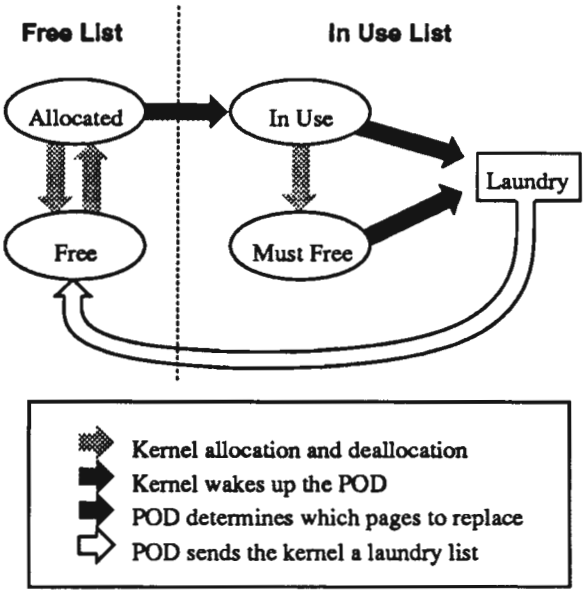

Figure 3: State diagram for page frame records

plicitly. First, since page records must be mapped into the kernel and into possibly several tasks, virtual address aliasing must be allowed by the machine architecture. While virtual address aliasing poses problems in some of today's caching architectures, we expect that future cache designs will address this problem [2, 13]. Second, because the POD only has access to the software-maintained machine-independent page records, and not the hardware-supported machine-dependent page tables, the POD will not have rapid access to the latest values of the modified and referenced bits. A kernel thread periodically updates the page-records. The absence of entirely up-to-date information is not expected to seriously impact the ability of PODs to manage memory effectively. Pages are flushed by the kernel, which does have access to the pmap interface, so recently modified pages will not be mistaken for clean pages. Third, the PODs share access to sensitive system data. They must therefore be cooperative and correctly written programs. Any ill-behaved POD could easily disrupt the system. This assumption of cooperation and correctness, however, must be made for the servers providing the infrastructure of any micro-kernel based system.

3.3 Sharing information between the kernel and PODs

In our architecture the resident page table and POD records are all mapped into the address space of every POD. Since PODs can both read and write sensitive data structures, it follows that PODs must be components of trusted systems, such as file and database management systems, rather than simple user applications. The delegation of responsibility to trusted processes is consistent with the microkernel

philosophy. It is not possible, however, to protect the kernel from the PODs or the PODS from one another.

The PODs operate on separate page frame lists and, when correctly programmed, do not access the same page frame records. Each POD, however, shares access to page frame records with the kernel and care must be taken that accesses are properly synchronized. Page frames are therefore divided between the free and in-use lists. The kernel does not access the in-use list. The POD does not access the free list. Transfers from and to the free list are accomplished by the creation of a temporary list whose head is passed to the POD or to the kernel via RPC. The POD accesses only the records of those frames on the in-use list. The kernel alone changes the state of a page frame on either list.

To implement a page replacement algorithm, a POD examines page frames on the in-use list. A thread in the kernel periodically updates the modified and referenced bits for these page records using information received through the pmap interface. For large page frame pools this information can be somewhat stale without significantly effecting the effectiveness of the approximate LRU algorithm. The kernel may also deallocate page frames on the in-use list (changing their state to must-free). The kernel does not alter the pointers of the in-use list.

We have made a few small additions to the Mach kernel interface to support user-level physical memory management. A POD registers with the kernel through the *phys_mem_request()* call. This call specifies the POD's minimum number of pages, the free-list low-water mark, and the periodic wakeup interval. The POD registers objects to be backed by its page frame pool by calling *phys_mem_register_object()*. The kernel wakes up PODs through the *phys_mem_pod_wakeup()* call. A POD can request that the kernel flush a list of pages through the *phys_mem_flush_queue()* call. The kernel removes each page from the object to which it is attached. If the page has been modified, the kernel invokes the pager through the current *memory_object_data_write()* interface.

3.4 Balancing demands for physical pages

The assignment of control over particular page frames to a POD is not permanent. Page frames can be reassigned to meet changing patterns of use. New PODs can also be created dynamically. The assignment of page frames to PODs is the responsibility of a ***balance manager***. Each POD reflects its "satisfaction" with its current memory allotment through a parameter provided to the balance manager through the POD record. Each POD has a declared minimum page requirement (the POD cannot be started if this requirement cannot be met). Subject to these constraints, the balance manager can shift free page frames from the free-list of one POD to that of another. The computation of satisfaction parameters is specific to managers and their proper computation is a subject for future research. Likewise, the balancing

of their requests will have to be examined in future work. Our present mechanism simply provides the framework. In our initial design, the balance manager attempts to maximize a weighted sum of these satisfaction parameters.

4 Implementation and Performance

In our initial implementation we have created user-level PODs implementing simple page replacement strategies. While our primary purpose is to create physical memory managers for file and database management systems rather than for virtual memory systems, we can run our system using a user-level POD to manage backing store for virtual memory. In this case we change the default POD from a kernel thread to one running at the user level.

Our implementation required some modifications of existing kernel data structures and their use, as well as the addition of some new structures. The principal modifications include

- The kernel and PODs need to share two sets of data structures, the resident page table and POD records (as shown in Figure 1). To allow this sharing, an object containing both the resident page table and all POD records is created during physical memory initialization. This object is mapped into a POD's address space as part of its registration through *phys_mem_request()*. For simplicity we decided to keep the object's size static. This means that the number of POD records and the size of the resident page table remains fixed.

- In the original Mach kernel, a page frame can be found in one of four places: the free list, the active list, the inactive list, or a zone. In our kernel, a page frame belongs to a POD's free list, a POD's in-use list, or to a zone. The change in page frame states (and possible page moves) made it necessary to change the page frame data structure and a number of kernel routines.

- When waiting for a page frame to be initialized by a pager during a page fault, a *fictitious* page frame record is used as a place-holder pointed to by a memory cache object. A fictitious page frame record does not point to any physical page frame. In the original kernel, the page frame pointers of a real and a fictitious page frame records are swapped once a new page has been initialized, essentially exchanging the roles of the two records. For simplicity we decided to map only real page frame records into the PODs' address spaces. Fictitious and real page frame records therefore cannot change roles. Instead the record pointed to by the memory cache object is changed.

The principal additions to the kernel were the POD data structure and maps allowing the POD to be found.

- The POD records mentioned above contain the following information: POD's task, POD's port, free list pointer, free count, free list lock, transfer list pointer, in-use list pointer, minimum free list size, free list low water mark, restart interval, and time to next restart. In our current implementation, the 'restart interval' and 'time to next restart' are not used.

- When a pager registers an object with the kernel, the kernel creates an object–POD mapping (which is hashed according to the object). When the object is actually mapped into an address space and created, the kernel fills the object's POD field from this mapping. Then, when the kernel needs to service a page fault for the object, the kernel is able to allocate a page from the appropriate POD's pool.

Our initial implementation runs on a Sun 3/60 with 12 Mbytes of memory. For our system, the cost of faulting on a zero-filled page supplied by an external pager is about 3.2 ms. Consulting with the POD to find a free page at the time of the fault adds an over head of approximately 14%. This is on the order of the cost of consulting the external pager, and accords well with McNamee and Armstrong's 10% overhead for consulting the external pager to obtain a free page [6]. In our case, however, this overhead is a worst-case figure. Since the POD attempts to maintain a free-list available to the kernel, it need not be consulted synchronously on every page fault.

If a POD maintains a free list by freeing bursts of pages when invoked, most page faults can be handled without invoking the POD synchronously. Consequently, the cost of consulting a user-level POD rather than the current kernel page-out daemon has little effect on the time taken to run a significant application. A series of compiles while remaking the kernel, for example, consulted the POD between 35 and 39 times over five runs, each taking nearly three minutes of wall clock time and approximately 102 seconds of combined system and user time. Thus the POD is consulted only once every four to five seconds of real time and less than once every two to three seconds of compute time. Not surprisingly, the differences in times between several runs using the kernel page-out daemon and several runs with a user-level POD were negligible. Frequent accesses to the POD can be artificially induced by creating processes that access large amounts of memory randomly. In this case the critical factor is disk I/O activity, rather than context switches between kernel and POD.

5 Conclusions

Systems in the future will have to handle a more diverse workload than they do today. This workload will include managing very large data objects such as images, audio streams, and video streams, as well as an increasing number of sophisticated

object managers and their underlying data management systems. All of these systems will rely on sophisticated use of caching for performance. It is therefore important for operating systems to provide to incorporate mechanisms allowing diverse memory management policies to be implemented, and to provide mechanisms for mediating contention for memory among these subsystems.

Our design exposes important data structures to selected user-level processes. This allows subsystems to take a direct role in managing physical memory assigned to them. Our design provides an interface through which physical memory managers can monitor the use of the page frames assigned to them and determine which pages to flush. In other work, we are evaluating the demands placed on file system buffers when handling photographic image files [10] to test the usefulness of this interface. Our design allows memory to be shifted among pools at the command of a balance manager within the kernel. We are investigating appropriate balancing policies.

References

[1] Hong-Tai Chou and David J. DeWitt. An evaluation of buffer management strategies for relational database systems. In *Proceedings of VLDB '85*, pages 127–141, Stockholm, Sweden, 1985.

[2] James R. Goodman. Coherency for multiprocessor virtual address caches. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, California, October 1987.

[3] Roger Haskin, Yoni Malachi, Wayne Sawdon, and Gregory Chan. Recovery management in QuickSilver. *ACM Transactions on Computer Systems*, 6(1):82–108, February 1988.

[4] H. Levy and P. Lipman. Virtual memory management in the VAX/VMS operating system. *IEEE Computer*, 22(3):35–41, March 1982.

[5] Keith Loepere. *MACH 3 Kernel Principles.* Open Software Foundation and Carnegie Mellon University, 1991.

[6] Dylan McNamee and Katherine Armstrong. Extending the Mach external pager interface to accommodate user-level page replacement policies. In *Proceedings of the USENIX Association Mach Workshop*, pages 17–29, Burlington, Vermont (USA), October 1990. USENIX Association.

[7] Sun Microsystems. *SunOS 4.1 Programmer's Manual.* Sun Microsystems, Inc, Mountainview, California, 1990.

[8] Michael N. Nelson. Virtual memory vs. the file system. Research Report 90/4, Digital Western Research Laboratory, March 1990.

[9] James Van Sciver and Richard F. Rashid. Zone garbage collection. In *Proceedings of the USENIX Association Mach Workshop*, pages 1–15, Burlington, Vermont (USA), October 1990. USENIX Association.

[10] Stuart Sechrest, Khaled Charif, and Wu-Chi Feng. File block costs of zooming and panning in JPEG compressed images. Technical Report CSE-TR-98-91, University of Michigan, 1991.

[11] Michael Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, July 1981.

[12] Michael Stonebraker. Managing persistent objects in a multi-level store. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, pages 2–11, Denver, Colorado (USA), June 1991. ACM, New York (USA).

[13] Wen-Hann Wang, Jean-Loup Baer, and Henry M. Levy. Organization and performance of a two-level virtual-real cache hierarchy. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 140–148, Jerusalem, Israel, June 1989.

[14] James Wyllie. Personal communication, October 1991.