Extending The Mach External Pager Interface To Accommodate User-Level Page Replacement Policies

Dylan McNamee and Katherine Armstrong

Department of Computer Science and Engineering
University of Washington
Seattle, Washington 98195

Abstract

The Mach external pager interface allows applications to supply their own routines for moving pages to and from second-level store. Mach doesn't allow applications to choose their own page replacement policy, however. Some applications have access patterns that may make least recently used page replacement inappropriate. In this paper, we describe an extension to the external pager interface that allows the programmer to specify the page replacement policy as well as the backing storage for a region of virtual memory.

1 Introduction

An operating system attempts to be all things to all users. Because of this, sometimes compromises have to be made; performance may be sacrificed for generality, or modularity for performance. Virtual memory page replacement schemes are an example of such a tradeoff. While the LRU page replacement policy rarely misbehaves grossly, it rarely provides optimal performance for any application. For some groups of applications we may suspect that a different page replacement algorithm would outperform LRU. Unfortunately, existing operating systems have a single, fixed page replacement scheme. Thus these applications either had to suffer the consequences, or had to make use of some special operations, such as "wire" and "flush", that may be provided by the operating system to alter LRU's actions.

The Mach operating system provides a user with certain control over the paging of an application. There is an external pager interface that allows applications to supply their own routines for moving pages to and from second-level store. Mach doesn't allow applications to choose their own page replacement policy, however. The Mach kernel chooses the page to be replaced, using an approximation to global LRU, and presents it for pageout to the pager associated with that page. The ability to specify backing store makes Mach's virtual memory system very flexible. Having gone as far as it has, it is a natural extension for Mach to give external pagers control over the page replacement policy for the pages under their control, since in many cases applications should know more about the patterns of access to their pages than the kernel. That is the accomplishment of the work described in this paper.

The next section motivates the extensions to Mach that will provide this facility. Section 3 summarizes Mach's virtual memory system. Section 4 discusses the various design decisions we made in the design of a user level page replacement system. Section 5 presents the structure of a system that uses our extensions. Section 6 presents a summary of the implementation status and preliminary performance results.

2 Motivation

Early on in the history of virtual memory research, various page replacement policies were suggested, with least recently used (LRU) being the most commonly discussed (and implemented) policy for choosing pages to remove from physical memory. Since then, most page replacement policies implemented by operating systems have been variants of LRU.

Understanding how to structure and use memory efficiently in parallel and distributed environments is still very much a topic of research. Features such as coresident task forces [Ousterhout 84], data replication, and coherence for distributed shared memory could be facilitated by extensions to the virtual memory system. Further, the growing gap between processor speeds and the speed of memory and I/O makes the choice of page replacement policy an important issue even in a uniprocessor environment. Although the trend toward larger memories will reduce page faults to some degree, those that still occur are becoming proportionately more expensive to service; therefore a paging policy that could further reduce page faults is potentially beneficial. A platform that allows different paging policies to be easily implemented and evaluated would be helpful in light of the demands of these new developments.

Through the external memory management interface, Mach already lets the programmer choose the backing store for a memory object[1] by selecting an external pager. Thus external pagers can be used to implement distributed backing store and shared virtual pages. This paper presents an extension to this interface that allows the programmer to implement the page replacement policy for a memory object as well.

A garbage collector is an example of a specific application that could benefit from the ability to implement its own page replacement policy. Rather than use the default, LRU, an external pager could maintain a prioritized list of pages to page out, with high priority pages being the pages it wishes to keep cached. The pages of text and data belonging to the garbage collector itself would have high priority, and the pages of data it has finished collecting would have lower priority. When asked to remove pages from physical memory, the pager would suggest garbage collected pages before its own private data, even though some of the private pages could be older than recently collected pages. This policy is certainly preferable to LRU, which would have kept the garbage collected pages in longer, since they were more recently touched.

Another application area that could make good use of the ability to choose its own page replacement strategy is databases. Stonebraker [Stonebraker 81] has observed that database systems access pages in a manner that makes LRU replacement inappropriate. Locality, a concept upon which the LRU policy is based, tends not to be as strong in the access patterns of database applications [Kearns & DeFazio 89]. An implication of this observation is that recency of access may not be the appropriate reference property upon which to base a page replacement scheme for database applications. Using user-level page replacement, an external pager could implement a strategy more appropriate for database accesses.

As a final motivating example we discuss Camelot [Eppinger 89], a transaction facility built on top of Mach using Mach's existing external pager interface. Transaction managers need to ensure that changes to virtual memory are also recorded on permanent storage. Since *hot pages*[2] will never be least recently used, the LRU scheme will never flush them on its own. In order to prevent the number of log entries from growing without bound, Camelot must explicitly check for hot pages and periodically force Mach to flush

[1] A memory object is an abstract object representing a collection of data bytes on which several operations (e.g. read, write) are defined [Young et al. 87]. Memory objects are discussed more fully in the next section.

[2] A *hot page* is a page that is used (either written or read) frequently. A dirty hot page has a log of all updates to it since its last write. Paging out a hot page to disk erases the log for that page [Eppinger 89].

them. With our extension to the Mach external memory management interface, Camelot could define an external pager that performed this action. Along with providing more modular system design, performance could be improved if a page replacement policy could be found that outperforms LRU in this setting. This example suggests that as experience grows, programmers could develop libraries of external pagers, each suited for certain types of applications or memory objects. As an example, an external pager for transaction systems could pageout dirty pages before clean pages, and unreferenced clean pages before referenced clean pages.

3 Mach Virtual Memory

We chose to use the Mach operating system for our implementation because the Mach abstraction of memory objects made the implementation easier than it would have been under most other operating systems. This section describes the memory object abstraction, and how it can be used to write external pagers[3].

The channels of communication between the Mach kernel, an application and an external pager are shown in Figure 1.

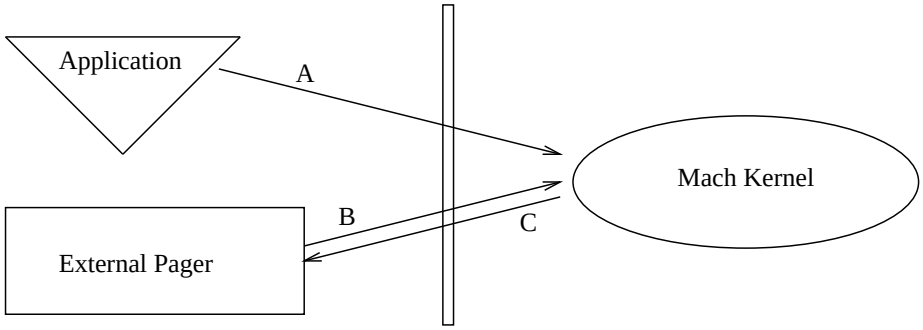

Figure 1: Communication Channels

A detailing of the relevant calls between kernel and user level is given in the Appendix. Each call is classified (using the letters in Figure 1) by the communication channel along which it passes. The application and its external pager may have a line of communication, if the pager's paging policy is adaptive, for example, but we have left this out of the diagram.

A memory object is an abstraction for a region of virtual memory in Mach. Physical memory is used to cache the contents of memory objects. Applications can map one or more memory objects into their address spaces, allowing them to access the data associated with those memory objects. If a task attempts to access data that is not currently cached, the kernel sends a message to the external pager responsible for that region of memory, requesting that the data be retrieved from backing store. When the kernel removes dirty pages from memory, it makes requests of the external pagers responsible for those pages to write their contents to the appropriate backing store.

[3] The terms external memory manager, external pager, and memory object all refer to a server implementing the external memory management interface [Young 89].

The `vm_allocate` call allocates a region of virtual memory backed by the default pager. `Vm_map` maps an external memory object into a task's address space. When a page fault occurs on a page backed by an external pager the kernel calls `memory_object_data_request`, which the pager is responsible for implementing. Once the pager has retrieved the data, it returns a pointer to the data to the kernel by calling `memory_object_data_provided`.

When a dirty page managed by an external memory manager needs to be removed from physical memory, the kernel calls `memory_object_data_write`. An external pager is expected to store the data it receives in a `memory_object_data_write` call and free the associated page so that it may be reused.

Communication between memory objects and the Mach kernel is through Mach RPC. RPC stubs are generated using the Mach Interface Generator (MIG) [Draves et al. 89]. This means that a memory object may be transparently connected to a remote kernel. Any number of tasks may map a particular memory object. Two tasks on the same machine may map the same portion of a memory object in order to share virtual memory. The same cached pages of physical memory may then be accessed by both tasks.

4 Design Decisions

The following subsections present the decisions we faced when designing an extended interface between the Mach kernel and external pagers. The theme of these tradeoffs was usually generality and flexibility versus performance. In the following subsections, an external memory manager that implements the extended interface presented in this paper will be called a Page REplacing Memory Object, or premo pager.

4.1 Direction of Information Flow

To allow an external pager to make page replacement decisions, it must have access to information about the pages it is managing. Many page replacement policies require page reference information to maintain the pages in next to replace order. Other information a pager may require are a page's dirty/clean status and whether it is active or inactive.[4] Since this information is currently available only to the kernel (via the pmaps), it was necessary to add a line of communication between the kernel and the pager. The direction of information flow could either be from the pager to the kernel or vice versa. This choice involves a tradeoff between performance and flexibility.

In the first alternative (information flows from pager to kernel), for each of its regions of virtual memory, the external pager selects from a list of kernel-implemented page replacement algorithms and communicates its choice to the kernel. After the initial communication of the policy choice, the kernel can maintain a list of the memory object's pages in the order prescribed by the page replacement policy without any further communication with the pager. Assuming a good implementation of each policy, this option has the greatest performance potential. The main problem with this approach is that it lacks flexibility. An application that requires a page replacement policy not currently supported by the kernel has to settle for a less efficient policy. The pager has no more information about its pages than it did before, which would prevent it from

[4] In Mach, a page is "active" when a reference to it exists in some task's *pmap*. A *pmap* is Mach's machine independent view of the information the hardware uses to perform virtual to physical address translation. Inactive pages (pages with no current references) may be reactivated if they are referenced, or are eventually sent to their pager to be written to backing store and freed.

making adaptive paging decisions. Pagers would have some control over policy, but none over mechanism. Clearly, this is not a good choice if experimentation with new replacement policies is a goal.

The alternative is to have the kernel keep the pager informed about page usage, allowing the pager to implement its own policy. This increases the amount and frequency of data flow between the kernel and the pager, which would decrease performance relative to the same (static) policy implemented within the kernel. On the other hand, a premo pager can implement a dynamically adaptive policy[5] only if the user-level external pager manages the replacement order of its pages. In addition, putting both policy and mechanism at the user level is more in keeping with the philosophy of the existing Mach external pager interface. Further, it allows applications to implement whatever page replacement policy seems natural, rather than have to settle for a "canned" policy provided by the kernel. For all of these reasons, the direction of information flow in our implementation is from the kernel to the memory object.

4.2 Communication Methodology

In dealing with the question of how to transfer information about pages from the kernel to the pager, simplicity of design and performance were the main criteria. We thought about allowing external pagers to have access to the page usage data kept in Mach's kernel-level pmap data structures, but ruled that out because of the protection issues inherent in moving data structures required by the kernel into user-level memory. The second option was to have the kernel maintain a copy of the desired information in a region of memory shared between the external pager and itself. This option was investigated but discarded as being too complex for the initial implementation. The final option, and the one we chose, was to have the kernel communicate with memory objects using Mach RPC. Using MIG with Mach RPC was appealing primarily because of its simplicity and modularity.

4.3 Communication Content

Certain page replacement policies may require more than just page reference information. For this reason, we include the ability for a premo pager to select which subset of the following set of page usage data the kernel should maintain on its behalf: referenced, active/inactive, dirty/clean. This selection is made when a memory object is initialized by the kernel. Future work in this area would be to identify and incorporate other information external pagers could want.

4.4 Communication Frequency

Page reference information is currently provided to the Mach kernel directly by the hardware. The implementation varies somewhat with the particular architecture (e.g., VAXes don't have reference bits, so information about when a page is touched has to be simulated by the VAX-dependent parts of Mach). In turn, we could have the kernel provide page usage information to the pager as often as each page access, or as infrequently as not at all. Clearly either extreme has major drawbacks. Providing information too

[5] An adaptive paging policy can adjust to the memory needs of the application. An example is the working set model [Denning 70] of page replacement. Its goal is to reduce the thrashing that occurs when a task cannot acquire enough physical memory to run efficiently. Other adaptive policies could be implemented with PREMO pagers to achieve high performance in less dire circumstances.

infrequently would leave the memory object unable to order its replacement lists with reasonable accuracy. Too frequent updates would reduce system throughput because of communication overhead.

As a compromise, our implementation provides page reference information to a premo pager each time a page is activated or deactivated. This information is provided via the `memory_object_ page_info` call from the kernel. The premo pager's information is thus updated at a sufficiently fine granularity, and communication isn't so frequent that it bogs down the kernel. Other page usage information, such as whether a page is dirty or clean, is similarly provided at a fine enough granularity that the memory object can have reasonably complete information about its pages, while not significantly affecting system performance.

5 Structure of a System Using PREMO Pagers

In an application that uses an external pager, the existing pager can be replaced by a premo pager without having to make any changes. The application acquires a communication port from the memory object server and links the memory object to a region of virtual memory using the kernel call `vm_map`. During the processing of the `vm_map` call, the kernel calls `memory_object_init` to let the memory object server know it will be responsible for this region of virtual memory. Up to this point the interaction is identical to the standard external pager initialization. As the premo pager initializes itself in `memory_object_init`, it lets the kernel know it will be making its own paging decisions by calling `vm_map_premo`. At this point everything is properly initialized; the region of virtual memory defined in the `vm_map` call is backed by a page replacing memory object.

When the Mach kernel needs to reclaim some physical pages, it looks at the page at the top of the kernel's replacement list.[6] If that page is handled by a non-premo pager, the kernel deactivates the page. If the page belongs to a premo pager, the page is instead moved to the tail of the replacement list (to keep it from being repeatedly selected), and the premo pager is asked to select one or more pages to relinquish. This information is passed to the kernel by calling `memory_object_hints_provided`. This implements a two-level page replacement scheme. The choice of which memory object must give up a physical page is determined by the kernel-maintained LRU ordering of pages caching the contents of memory objects. When the selected memory object is managed by a premo pager, the choice of which of the pages will be relinquished is made by the premo pager. One can envision a scheme whereby both levels of this two-level approach are variable. How one would choose a suitable policy and mechanism for the first level is an interesting problem, however it is beyond the scope of this paper.

6 Implementation and Performance

The premo pager interface has been incorporated into Mach 2.5 running on a MicroVAX II. Synthetic applications with known paging behavior were coupled with premo pagers to verify the correctness of the implementation. We are currently investigating incorporating premo pagers into real applications.

There are two points to consider when deciding whether to incorporate premo pagers into an application. The most important consideration is whether an application's memory access patterns make LRU or its

[6] Mach presently uses second chance FIFO as an approximation to LRU.

variants inappropriate. Premo pagers could potentially improve the performance of such applications by providing an alternate page replacement policy that significantly reduces paging.

The second consideration is the communication overhead incurred by using premo pagers, and whether the gains due to reduced paging outweigh the added overhead. To measure the overhead of using premo pagers, we compared the elapsed runtime of our synthetic benchmark program using a premo pager that implements the same replacement policy as the kernel, to the same program using the kernel's default pager. Preliminary measurements indicate that the overhead associated with using a premo pager adds approximately 10% to this application's runtime.

To get an idea of the increase in performance made possible by premo pagers, we wrote a premo pager that took advantage of the benchmark program's memory access patterns to reduce page faults. This pager reduced the number of page faults by 15%. While these results are encouraging, it is important to investigate the performance of real applications running with premo pagers. We expect to present more complete performance figures at the workshop.

7 Conclusions and Future Work

The Mach external pager interface allows pagers to specify backing storage for memory objects. Our extension to this interface gives external pagers the capability to manage the paging behavior of their resident pages. Using this extension, paging policies that are more appropriate for special classes of applications, such as database and transaction systems, mapped files, and garbage collectors, may be implemented. Kathy Armstrong is presently investigating the use of alternate paging policies for mapped files. Other work includes investigating the integration of page replacing memory objects into other classes of applications in a distributed environment.

The present implementation of the interface allows potentially misbehaving pagers to monopolize physical memory. As the implementation matures we will add security checks to ensure that each pager minimally fulfills its paging responsibility. If, as our experience grows, communication costs turn out to be a limiting factor to performance, we may consider implementing the shared memory communication method.

8 Acknowledgments

We wish to thank Tom Anderson, Brian Bershad, Eric Koldinger, Ed Lazowska, Brian Pinkerton, and John Zahorjan for their helpful comments. We also appreciate the efforts of Jan Sanislo and Eric Lundberg in obtaining the source code and hardware necessary for our implementation.

9 Appendix – Partial interface between an external pager and the kernel

From kernel to memory object

`memory_object_hint_request` requests hints from a page replacing memory object. Type C (see Figure 1.)

`memory_object_page_info` provides requested information about a page to a page replacing memory object. Type C.

`memory_object_init` serves notice to a memory object that Mach has been asked to map a given memory object into a task's virtual address space. Type C.

`memory_object_data_request` is a request for data from the specified memory object. This data is returned to the kernel with the `memory_object_data_provided` call. Type C.

`memory_object_data_write` provides the memory object with data that has been modified while cached in physical memory. Once the memory object has written this data to second level store, it is expected to deallocate the physical memory, using `vm_deallocate.` Type C.

From user level to kernel

`vm_map` maps a region of a memory object into a task's address space. Type A.

`vm_map_premo` informs kernel this is a page replacing memory object. Type B.

`memory_object_hints_provided` tells kernel which cached pages may be discarded. Type B.

`memory_object_data_provided` provides data for a region of virtual memory. Type B.

`memory_object_data_error` indicates that the memory object cannot return the data requested for the given region. Type B.

`memory_object_data_unavailable` indicates that this memory object does not have data for the given region. Type B.

`memory_object_lock_request` allows a memory object to make special requests of the kernel. These include write back, flush, lock, and unlock. Type B.

`memory_object_set_attributes` controls how Mach uses the memory object. The memory object can request that Mach keep its pages in physical memory, even when the pages are no longer mapped into any virtual address space. Type B.

References

[Baron et al. 89] Robert V. Baron, David Black, William Bolosky, Jonathan Chew, Richard P. Draves, David B. Golub, Richard F. Rashid, Avadis Tevanian, Jr., and Michael Wayne Young. "Mach Kernel Interface Manual" Technical Report, Department of Computer Science, Carnegie-Mellon University, June 1989.

[Denning 70] P. J. Denning, "Virtual Memory," *Computing Surveys*, Volume 2, Number 3, 1970.

[Draves et al. 89] Richard P. Draves, Michael B. Jones, Mary R. Thompson "MIG - The MACH Interface Generator" Technical Report, Department of Computer Science, Carnegie-Mellon University, November 1989.

[Eppinger 89] Jeffrey L. Eppinger. "Virtual Memory Management for Transaction Processing Systems." PhD thesis, Carnegie-Mellon University, 1989.

[Forin et al. 89] Alessandro Forin, Joseph Barrera, and Richard Sanzi. “The Shared Memory Server” *1989 Winter USENIX Conference*, 1989.

[Kearns & DeFazio 89] John P. Kearns and Samuel DeFazio. “Diversity in Database Reference Behavior” *Performance Evaluation Review*. Volume 17, Number 1, 1989.

[Levy & Eckhouse 88] Henry M. Levy and Richard H. Eckhouse, Jr. *Computer Programming and Architecture: The VAX*. 2nd Ed. Digital Press, 1988.

[Ousterhout 84] John K. Ousterhout. “Scheduling Techniques for Concurrent Systems” *Proc. 3rd IEEE International Conference on Distributed Computing Systems*, 1984.

[Peterson & Silberschatz 85] James L. Peterson and Abraham Silberschatz. *Operating System Concepts*. Addison-Wesley Publishing Company, Reading, MA, 1985.

[Stonebraker 81] Michael Stonebraker. “Operating System Support for Database Management” *Communications of the ACM*. Volume 24, Number 7, 1981.

[Tevanian et al. 87] Avadis Tevanian, Richard Rashid, Michael W. Young, David B. Golub, Mary R. Thompson, William Bolosky, Richard Sanzi. “A Unix Interface for Shared Memory and Memory Mapped Files Under Mach” Technical Report, Department of Computer Science, Carnegie-Mellon University, July 1987.

[Young et al. 87] Michael Young, Avadis Tevanian, Richard Rashid, David Golub, Jeffrey Eppinger, Jonathan Chew, William Bolosky, David Black, and Robert Baron. “The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System” *Proc. 11th Symposium on Operating Systems Principles*, 1987.

[Young 89] Michael Young “Exporting a User Interface to Memory Management from a Communication-Oriented Operating System.” PhD thesis, Carnegie-Mellon University, 1989.