EXTENDED ABSTRACT *

Remote Procedure Call Implementations Of Micro-kernel Virtual Memory Services Degrade System Performance

David K. Raila See-Mong Tan Roy H. Campbell
Department of Computer Science
University of Illinois at Urbana-Champaign
Digital Computer Laboratory
1304 W. Springfield
Urbana, IL 61801
{*roy,raila,stan*}@cs.uiuc.edu

Abstract

Virtual memory subsystem design is critical to the performance of micro-kernel operating systems which partition services into multiple protection domains. Micro-kernel virtual memory implementations that use synchronous remote procedure call as a communication and control mechanism suffer performance degradation due to the mismatch between the call-return model of remote procedure calls and the asynchronous, often unidirectional, behavior of virtual memory systems. Analysis of common virtual memory actions shows that remote procedure calls are an inefficient mechanism for micro-kernel virtual memory implementations and that the performance of the system can be improved by a design tailored to the communication and synchronization patterns of the system. In this paper we describe the problems associated with the use of remote procedure calls to implement micro-kernel virtual memory implementations. Then we present a design for the μ*Choices* virtual memory system that minimizes context switching and eliminates unnecessary control flows transfers through the system by using a novel approach to kernel-pager communication and synchronization. We are able to reduce the number of protection boundary crossings in the system by one half and obtain greater parallelism and independence of operation between parts of the system.

1 Introduction

The virtual memory subsystem is one of the most complex and frequently used services in modern operating systems. It is used for protection between the kernel and the processes running on the system, protection of hardware registers, and for data transfer and communication between the hardware, kernel, and application programs. Features such as memory mapped files, external paging support, and copy on write require complex implementations that affect the performance of the entire system. Support for multiprocessors, threaded applications, and high bandwidth multimedia applications that require high performance virtual memory add complexity to the system, making it prone to performance problems.

Micro-kernel architectures often split the virtual memory subsystem design into modules implemented in separate address spaces connected by control and interprocess communication facilities, such as remote procedure calls, or message passing systems. Separation breaks module dependencies and aids module replacement, but it adds complexity and overhead from communications, context switching, and synchronization[3]. Although the performance of individual modules may be improved, the performance of the entire system can be degraded by context switching, synchronization, and communication overhead if a monolithic implementation is simply split up in a micro-kernel.

As we redesigned the Choices[16, 11] operating system as a micro-kernel based system[7, 17] we analyzed the virtual memory designs in Choices, Spring[13, 12], and Chorus[1], studying the control, communication, and synchronization patterns of the common virtual memory requests shown in table 1.

Diagrams illustrating control flows and communica-

*This is an extended abstract submission. By the publication date we expect to have a full paper, including performance data.

Map an Object
Unmap an Object
Page In (Page Available)
Page In (Page Unavailable)
Page Out
Page Replace
Protection Fault (Update Mapping)

Table 1: Common Paging Actions Studied

tions between modules in the system, such as the one shown in figure 1, were developed to evaluate each design. As we evaluated the systems and our own preliminary designs based on a remote procedure call implementation of the *Choices* virtual memory system, we found an abundance of unnecessary protection boundary crossings and communication between modules within the system. The two contributing factors to these problems are the use of remote procedure calls as an implementation mechanism for a complex asynchronous subsystem, and the lack of recognition, within the design, of the communication and synchronization patterns possessed by the system.

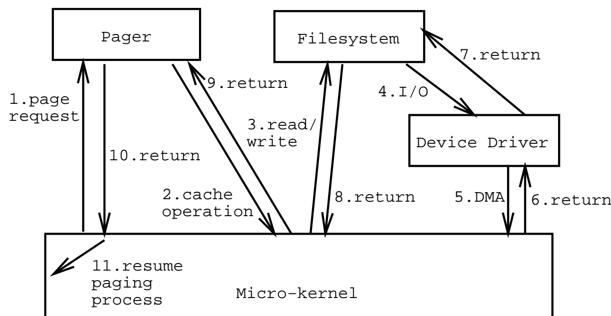

Figure 1: Flow Control During Paging for A Remote Procedure Call Based Implementation

2 Problems With Remote Procedure Calls

Remote procedure calls are a good mechanism for a wide variety of applications because they provide a call-return model, and isolate the synchronization and communication issues from the programmer[6]. However, when used to implement a complex asynchronous system that does not follow a call-return model, the remote procedure call model forces unnecessary flow control returns to the caller. Additionally, the remote procedure call model does not provide good support for parallelism and high performance data transfer. These issues require additional programming in order to resolve the mismatch between the remote procedure call model and the requirements of the system[18].

2.1 Context Switching

A remote procedure call invocation results in a call to the server, a return to the caller, and two way communication of arguments and results along the call path. When a virtual memory activity crosses several modules this results in several calls and returns along the path, each resulting in a context switch and blocking of the caller until the server returns. In addition to the costs associated with the switch itself[4], there are secondary issues such as the cache effects of context switching[14] that reduce performance. In our study of common paging activities we found that many of the return values were simply passed through to the caller and often no useful work was taking place in the intermediate modules along the call path. For example, during a page-in operation that traverses the pager, filesystem, and driver address spaces, the result from the I/O operation traverses the filesystem and pager domains unnecessarily during the remote procedure call return. The I/O driver can resume the blocked (paging) process directly, bypassing three address space and context switches.

2.2 Parallelism

Remote procedure calls emulate a single thread of control using two or more communicating, synchronized threads. Many actions within the virtual memory system can operate in parallel and do not require synchronization between caller and callee at each remote procedure call. For example, posting page faults from multiple processors or posting I/O requests to devices are actions that can proceed in parallel and require less synchronization than the remote procedure call model provides. To design the system to take advantage of parallelism additional code must be written to accommodate the mismatch between the remote procedure call model and the parallel implementation of the system[18].

Servers that can process parallel requests are written using multiple threads to service incoming calls in parallel. This also maintains availability of the server when threads within the server make calls to other servers during processing that might block. However, servers that wish to re-prioritize or batch process incoming requests need access to *all* pending calls[1] to accomplish their task. This requires additional server code for synchronization, aggregation of single requests into structures available to all server threads, and for arranging return values with the appropriate threads. This complicates virtual memory modules

[1] This is the case for message-based systems as well.

that prioritize and re-order parallel requests, such as pagers. To solve this problem a mechanism is needed that allows the server access to all pending requests simultaneously, minimizes synchronization, and does not enforce the call-return model of remote procedure calls.

3 Design Problems in Micro-kernel Virtual Memory Systems

In the remote procedure call based implementations that we studied, we found that there were aspects of the designs that required excessive synchronization, kernel intervention, or address space switching. These problems are caused by the use of remote procedure calls to implement the system, resulting in misplaced policy structures in the designs, and complication in the design of the protocol that drives the paging activities. When the remote procedure call model is abandoned the design of the system becomes cleaner, with simple protocols and better placement of policy information within the system (section 4).

3.1 Control Flow

The control flow of a micro-kernel virtual memory system has a dramatic impact on the amount of context switching and communication in the system. A kernel driven system implemented with remote procedure calls can suffer from *ping pong* effect, in which control transfers back and forth visiting modules in the system multiple times. For example, the Choices[15] and Spring[13] virtual memory systems are driven by cache objects located in the kernel, resulting in control flows that thread in and out of the kernel during processing. Figure 1 shows the cyclical flow control for a kernel driven virtual memory subsystem decomposed into multiple address spaces connected by remote procedure calls.

After careful consideration of the virtual memory protocols we were able to develop control flows for the *μChoices* virtual memory system that were unidirectional except in exceptional conditions, minimizing protection boundary crossing and context switching within the system (section 4).

3.2 Location of Policy and State Information

Fast access to policy and state information within the virtual memory subsystem is necessary to achieve high performance, especially in cases where accesses are frequent, such as in pagers. To process a page fault external to the kernel the pager may need access to a variety of state information including:

- Address space protection information
- Page frame information (accessed and modified bits)
- System memory allocation information
- Per-cache information
- Per-process memory information

Traditionally, policy and state information has been kept in the kernel and accessed though system calls, which are restrictive and can inhibit performance. Many micro-kernel designs do not change this situation, with policy information residing in a single location and accessed via remote procedure calls. Policy information, such as page replacement algorithms and cache block transfer sizes, have also traditionally been located in a central location along with state information. Accessing this information often causes the *ping pong* effect, where the pager returns to the kernel to retrieve information it needs to process the fault. State information must be protected, and, since the bulk of it is managed by the hardware, the micro-kernel is a good location for it.

Access to this information can be accomplished using more efficient mechanisms (section 4.2). Policy information can be removed from the kernel, which only provides mechanisms for virtual memory. Policy information should not reside in the kernel. It should be located in the pager where it is easily modified or replaced. A common example is the location of the cache block transfer size, which determines how much data to transfer between the virtual memory cache and the mapped object during paging operations. This information should reside in the pager, which is more capable of deciding how much data to transfer than the kernel based on the application. Pagers for applications exhibiting bursty data throughput, such as a multimedia data stream filter, might vary the transfer size dynamically without kernel intervention.

4 The *μChoices* Virtual Memory Design

The virtual memory system for *μChoices*[7] is a full featured memory system designed to support external paging, distributed virtual memory, copy on write, replacement policy customization, etc. The design attempts to increase performance by minimizing protection boundary crossings and context switching during virtual memory operations. *μChoices* does not use remote procedure calls in the virtual memory implementation. Instead, *μChoices* uses customized communication and control transfer facilities with features similar to those found in LRPC[5], Continuations[8], Split-Level Scheduling[10], and Scheduler Activations[2].

The *µChoices* virtual memory model consists of a core of virtual memory mechanisms inside the micro-kernel with the pager, filesystem, device drivers, and applications running on top of the micro-kernel in separate address space[2]. Pagers in *µChoices* are normal applications with no special privileges. Paging operations are *pager driven*, with the kernel providing *only* virtual memory mechanisms and protection enforcement within the system. The communications between the kernel and the pager are accomplished using customized, shared memory and an access protocol that allows maximum parallelism and minimal synchronization across module boundaries.

4.1 Pager Driven Protocol

In *µChoices* the pager drives all paging operations, policy decisions, and actions on behalf of the object that it pages. There is minimal kernel involvement. The kernel passes control to the pager and is involved again only to service I/O requests, update mappings (if required), and resume or terminate the thread that made the paging request. The kernel provides only basic paging mechanisms related to the hardware such as address translation operations, cache flushing, and physical page management.

By customizing the protocols between the modules associated with the virtual memory system *µChoices* provides high degree of parallel operation within the virtual memory system. The customized protocols and communication mechanisms limit the control flow to passing through each module only once during processing, except in error cases. The absence of remote procedure calls eliminates the additional context switching and synchronization overhead associated with the remote procedure call model. Figure 2 illustrates the control flow of a paging request in the customized *µChoices* system. In contrast to figure 1, which shows the remote procedure call implementation, there is a clean flow of control and one half the number of protection boundary crossings.

4.2 Shared Memory Communication

The *µChoices* kernel communicates with the pager via shared memory in three ways. Virtual memory state information in the kernel is mapped into the pager address space with read-only protection, the kernel and the pager communicate requests for service from each other through a shared memory region similar to split level scheduling[10]. Capabilities to the address space of the paging process are passed to the pager to avoid copying between address spaces.

[2]It is possible to locate modules, such as the filesystem and device driver, in the same address space.

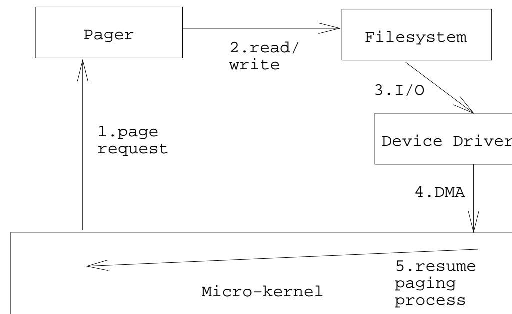

Figure 2: *µChoices* Based Control Flow for Page Fault

By mapping key kernel data structures into the pager address space the pager is given full access to the entire virtual memory state within the kernel directly, without requiring system calls. All relevant virtual memory structures such as protection bits, cache sizes, page allocations, etc. are directly accessible through the address space of the pager, eliminating the need for system calls to access this information.

A region of writable shared memory is used to communicate between the kernel and the pager. The kernel deposits paging requests to the pager in this region, and the pager deposits requests for actions, such as address translation updates, to the kernel. To avoid synchronization between the kernel and the pager each communication slot in the region contains a flag that specifies the state and owner of the slot. The kernel is free to insert requests in any slot that it owns and refrains from accessing those owned by the pager, and the pager is free to insert kernel requests in any slot that it owns and refrains from accessing kernel-owned slots. When a request is inserted into the slot the ownership bit is toggled to notify the kernel or pager that a request is pending. The protocol is similar to the one used by the LANCE Am7990 ethernet chip[9] to transfer packets between the kernel and the network interface hardware efficiently.

To avoid data copying of pages between address spaces, the *µChoices* kernel passes capabilities from the paging processes' address space with requests to the pager. The pager passes the capability on, if necessary, to the filesystem and it eventually arrives back in the kernel when the I/O operation for the region occurs. The I/O operation does Direct Memory Access (DMA) directly to the paging processes memory and the costs associated with data copying are eliminated. The pager, filesystem, or any other module along the way may substitute its own capability for the I/O request. This flexibility enables customized data transfer mechanisms such as the implementation

of double copying in a memory mapped filesystem.

4.3 Synchronization

μChoices minimizes synchronization across protection boundaries by using customized communication structures such as the shared memory communication block between the pager and kernel that was presented in section 4.2. In addition, *μChoices* uses customized synchronization mechanisms to achieve performance by minimizing extraneous synchronization while allowing processes along the path of processing to register for notification of events.

In *μChoices* the kernel directly resumes the paging process once I/O is complete, as shown in figure 2. This is similar to Continuations[8]. It avoids the unnecessary maintenance of implicit information on the remote procedure call stack, and is only possible in the absence of remote procedure calls in the implementation. The resumed thread, still in the kernel, performs actions left by the pager in the shared memory communication area, such as address translation or protection updates, before returning from the kernel to the application.

To provide flexibility in constructing systems with more complex synchronization needs, *μChoices* also provides the ability for modules to pass synchronization capabilities to the virtual memory system that allow a process to obtain notification about errors or completion of actions. The facility is similar to the virtual memory address capability presented in section 4.2, with the addition of a stacking feature that allows multiple entities to register for events.

5 Conclusion

Remote procedure calls are an inefficient mechanism for implementing a micro-kernel virtual memory system. The remote procedure call model imposes unnecessary synchronization and context switching and requires significant additional code to program parallel operations. By careful examination of the components of the system and their interactions simple mechanisms can be developed to implement a micro-kernel virtual memory system that minimizes context switching, data copying, and has a high degree of parallelism. The *μChoices* virtual memory subsystem is a customizable virtual memory implementation that uses a pager driven protocol, shared memory communication, and customized communication and synchronization structures that enable the reduction, by half, of the number of protection boundary crossings during virtual memory operations.

References

[1] Vadim Abrossimov, Marc Rozier, and Michel Gien. Virtual Memory Management in CHORUS. Technical report, Chorus syste'mes Technical Report CS/TR-89-30, 1989.

[2] T. Anderson, Brian Bershad, Edward Lazowska, and Henry Levy. Scheduler Activations: Effective Kernel Support for the User Level Managment of Parallelism. *ACM Transactions on Computing Systems*, 10(2):53–79, February 1992.

[3] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska. The interaction of architecture and operating system design. In *ASPLOS, International Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 108–120, Santa Clara, CA (USA), April 1991.

[4] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska. The Interaction of Achitecture and Operating System Design. In *Proceedings of ASPLOS-IV*. ACM Press, 1991.

[5] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight remote procedure call. Technical Report 89-04-02, Department of Computer Science, University of Washington, Seattle, WA (USA), April 1989.

[6] Andrew D. Birell and Bruce J. Nelson. Implementing Remote Procedure Calls. In *ACM Transactions on computer systems*, February 1984.

[7] Roy H. Campbell and See-Mong Tan. μChoices: An Object-Oriented Multimedia Operating System. In *Fifth Workshop on Hot Topics in Operating Systems*, Orcas Island, Washington, May 1995. IEEE Computer Society.

[8] Richrd P. Daraves, Biran N. Bershad, Richard F. Rashid, and Randall W. Dean. Continuations: Unifying Thread Management and Communication in Operting Systems. Technical report, School of Computer Science, Carnegie-Mellon University, 1991.

[9] Advanced Micro Devices. *The Am7990 LANCE Family IEEE-802.3/Ethernet Node*. Advanced Micro Devices.

[10] Ramesh Govindan and David P. Anderson. iScheduling and IPC Mechanisms for Continous Media. Technical report, University of California at Berkeley, March 1991.

[11] G. M. Johnston and R. H. Campbell. "An Object-Oriented Implementation of Distributed Virtual Memory". In *Workshop on Experiences with Building Distributed and Multiprocessor Systems*, pages 39–57. Usenix, 1989.

[12] Y. Khalidi and M. Nelson. A Flexible External Paging Interface. In *A Spring Collection*. SunSoft, 1993.

[13] Y. Khalidi and M. Nelson. The Spring Virtual Memory System. In *A Spring Collection*. SunSoft, 1993.

[14] Jeffrey C. Mogul and Anita Borg. The Effect of Context Switches on Cache Performance. In *Proceedings of ASPLOS-IV*. ACM Press, 1991.

[15] V. F. Russo and R. H. Campbell. "Virtual Memory and Backing Storage Management in Multiprocessor Operating Systems Using Object-Oriented Design Techniques". In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 267–278, October 1989.

[16] Vincent Russo and Roy H. Campbell. Virtual Memory and Backing Storage Management in Multiprocessor Operating Systems using Class Hierarchical Design. In *Proceedings of OOP-SLA '89*, pages 267–278, New Orleans, Louisiana, September 1989.

[17] See-Mong Tan, David Raila, and Roy H. Campbell. An Object-Oriented *Nano-Kernel* for Operating System Hard ware Support. In *Fourth International Workshop on Object-Orientation in Opera ting Systems*, Lund, Sweden, August 1995. IEEE Computer Society.

[18] A. S. Tanenbaum and R. van Renesse. A critique of the remote procedure call paradigm. In *EUTECO '88 Proceedings, Particpants Edition*, pages 775–783, Amsterdam, Netherlands, 1988. North-Holland.