Fast Interrupt Priority Management in Operating System Kernels

Daniel Stodolsky
J. Bradley Chen
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh PA 15213
danner@cs.cmu.edu
bchen@cs.cmu.edu

Brian N. Bershad
Department of Computer Science
and Engineering
University of Washington
Seattle WA 98195
bershad@cs.washington.edu

Abstract

In this paper we describe a new, low-overhead technique for manipulating processor interrupt state in an operating system kernel. Both uniprocessor and multiprocessor operating systems protect against uniprocessor deadlock and data corruption by selectively enabling and disabling interrupts during critical sections. This happens frequently during latency-critical activities such as IPC, scheduling, and memory management. Unfortunately, the cycle cost of modifying the interrupt mask has increased by an order of magnitude in recent processor architectures. In this paper we describe *optimistic interrupt protection*, a technique which substantially reduces the cost of interrupt masking by optimizing mask manipulation for the common case of no interrupts. We present results for the Mach 3.0 microkernel operating system, although the technique is applicable to other kernel architectures, both micro and monolithic, that rely on interrupts to manage devices.

1 Introduction

This paper describes a new technique, *optimistic interrupt protection*, that reduces interrupt management cost over conventional methods. While modern processor architectures have led to substantial overall performance improvements, operating systems have received significantly less benefit than application code [Anderson et al. 91, Chen & Bershad 93, Ousterhout 90]. One processor function that has not scaled well with processor speed is interrupt management. Operating systems use interrupts to control scheduling and I/O, and use interrupt masking to guarantee integrity of system resources shared across interrupt levels. This approach was efficient in many previous processor architectures (e.g, VAX), where the cost changing interrupt levels was small - generally less than ten instructions [Morse et al. 82, Levy & Eckhouse 89]. In modern architectures, however, interrupt masking may be up to an order of magnitude more expensive, contributing to poorer performance of system code.

This research was sponsored in part by The Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing", ARPA Order No. 7330, issued by ARPA/CMO under Contract MDA972-90-C-0035, by the Xerox Corporation, and by Digital Equipment Corporation. Bershad was partially supported by a National Science Foundation Presidential Young Investigator Award.
The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of ARPA, XEROX, DEC, the NSF, or the U.S. government.

Machine	Processor	Instructions	Cycles
Luna88k	Motorola 88100 (25MHz)	68	96
Flamingo	Alpha 21064 (150MHz)	25	21
DECstation 5000/120	R3000 (20MHz)	65	80
DECstation 5000/200	R3000 (25MHz)	14	14
386 PC	Intel 386DX-25 (25MHz)	51	179
Gateway 66V	Intel 486DX2-66 (66MHz)	51	279

Table 1: *Overhead of changing the interrupt mask. Cycle counts are estimated, assuming no cache misses.*

Optimistic interrupt protection avoids the performance penalty of interrupt mask manipulation while preserving the semantics of the interrupt model. We have implemented optimistic interrupt protection in the Mach 3.0 microkernel for several different processor architectures. On the Omron Luna88k, we observed a 50% reduction in interrupt management overhead, resulting in a 5.3% speedup for interprocess communication.

The rest of this paper describes the technique and its performance. In Section 2 we review the basic problems introduced by interrupts, discuss the general model of interrupt handling into which optimistic interrupt protection fits, and motivate the need for a high performance mechanism. In Section 3 we describe the use and implementation of optimistic interrupt protection. In Section 4 we discuss the performance of our approach. In Section 5 we discuss related work. Finally, in Section 6 we present our conclusions.

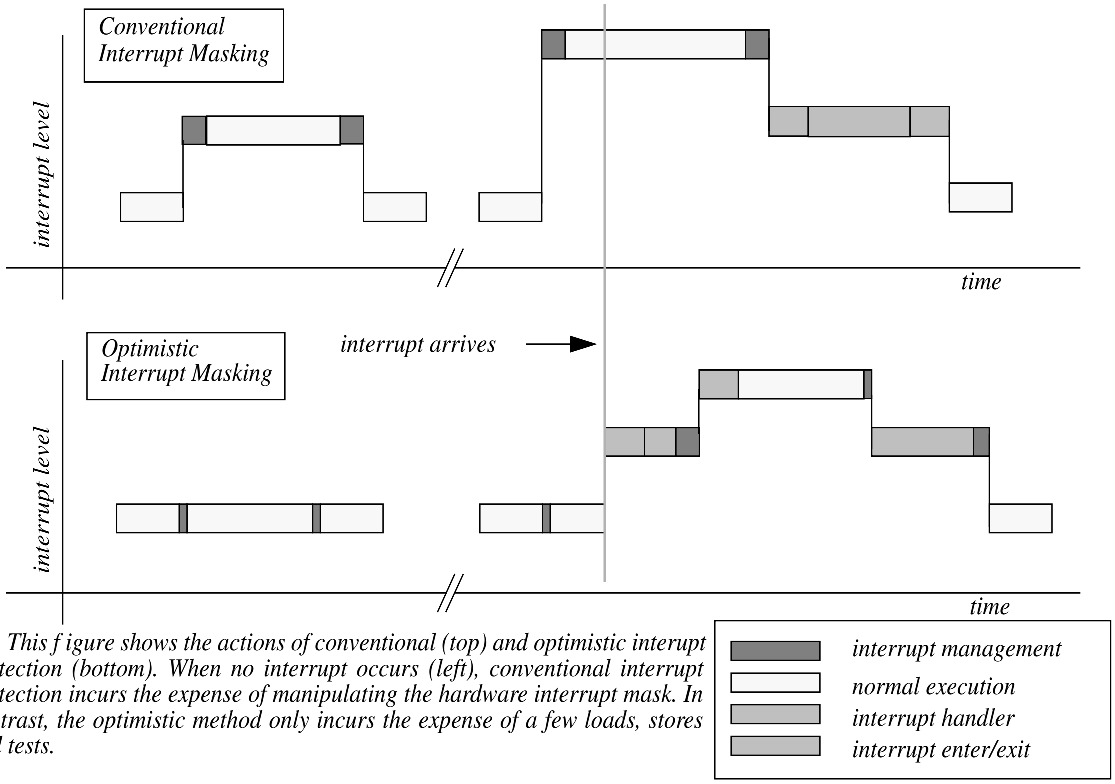
2 Interrupt management

Operating systems generally rely on interrupts to respond to asynchronous events. Because interrupts introduce concurrency into the operating system kernel, system-level mechanisms are necessary to avoid deadlocks and protect system data structures from inadvertant concurrent accesses. *Interrupt masking* is a common technique for data protection in the presence of asynchronous events. Access to a potentially concurrent data is protected by setting the processor interrupt level to prevent all events that could potentially alter the data in question. Interrupt masking has been used successfully in a large number of operating systems, including Mach, Unix, VMS, and NT [Accetta et al. 86, Leffler et al. 89, Levy & Eckhouse 89, Custer 93]. It maps well onto a diverse array of hardware, from systems with a single interrupt level to processors with a rich interrupt structure [Bell et al. 82, Intel 90]. On a uniprocessor, no additional synchronization constructs are required. An important property of the interrupt masking model is that latency-sensitive events can preempt long-running low priority activities. Although alternatives to the interrupt model have been proposed [Cheriton 88, Massalin & Pu 89], simplicity, as well as the significant investment in existing system code and programmer experience provide significant economic incentives for preservation of interrupts as a model of system data protection.

Traditionally, interrupt masking has been efficient, requiring only a few cycles. Unfortunately, the time required to modify the hardware interrupt level has not scaled with processor speed improvements. In pipelined processors, writing the processor interrupt mask typically requires a pipeline flush [Motorola 90, DEC 92]. In superscalar systems, interrupt level manipulations require scalar instruction issue, further limiting performance [Sites 92]. Many recent RISC CPU implementations provide only a part of the interrupt mask logic on the processor package, with the remainder of interrupt masking implemented by off-processor hardware [Motorola 90, DEC 92]. For these systems, interrupt masking is a three step process: 1) disable processor interrupts, 2) write the off-chip mask register(s), and 3) finally re-enable processor interrupts. The first stage requires a pipeline flush, and the second stage requires a potentially expensive off-chip access. This represents a significant increase in the relative latency of interrupt mask manipulations. Table 1 shows the cost of a general interrupt mask raise/lower pair within the Mach 3.0 microkernel on a variety of architectures.

3 Optimistic interrupt protection

Optimistic interrupt protection exploits the fact that, in the common case, interrupts do not occur during critical sections. When a processor executing in the kernel enters a critical section, its sets a *software interrupt mask*, which indicates the interrupts that need to be masked. The hardware interrupt mask is not changed. In the uncommon case that a lower-priority interrupt does occur, the interrupt handler prologue constructs an *interrupt continuation* (described below), updates the hardware interrupt mask as specified by the software interrupt mask, and returns control to the interrupted activity. Updating the hardware interrupt mask when the interrupt actually occurs prevents additional logically masked interrupts from occurring until the deferred handler has been executed. Though not strictly necessary, this tends to simplify the code. Moreover, it occurs after the interrupt, and is therefore off the anticipated fast path.



This figure shows the actions of conventional (top) and optimistic interrupt protection (bottom). When no interrupt occurs (left), conventional interrupt protection incurs the expense of manipulating the hardware interrupt mask. In contrast, the optimistic method only incurs the expense of a few loads, stores and tests.

If an interrupt does occur (right), hardware masking defers the delivery of the interrupt until the end of the critical section in the conventional case. The interrupt is delivered promptly with optimistic interrupt protection, causing control to transfer to the interrupt handler. The interrupt handler recognizes this interrupt is logically masked, constructs an interrupt continutation, sets the hardware interrupt mask to the logical mask, and returns from the interrupt. Since the interrupt mask is raised, the critical section can run to completion without further interruption. When the critical section is done, the kernel discovers the presence of an interrupt continuation, resets the hardware interrupt mask, and executes the continuation. After the continuation is complete, the interrupt mask is cleared and normal processing resumes.

Figure 1: *Conventional and Optimistic Interrupt Protection*

An interrupt continuation is a data structure containing the state of the system at the time an interrupt is deferred. The interrupt continuation contains sufficient information to service the interrupt condition at a later time. The amount of information is typically quite small (e.g, the program counter and interrupt vector). At the end of the critical section, the processor checks for an interrupt continuation. Normally there is none, and processing continues following the critical section. If an interrupt continuation does exist, the processor handles the corresponding interrupt condition before resuming "normal" computation (see Figure 1). The interrupt continuation handles the deferred interrupt, restores the hardware interrupt mask to its original level, and returns to the normal execution stream.

As with traditional interrupt control, optimistic interrupt protection defers the execution of a masked interrupt handler until the end of the protected critical section. Unlike the traditional masking mechanisms, it requires that the (hardware and software) execution of the interrupt prologue code be both allowed and safe during protected sequences. As an example, if the interrupt prologue required a valid stack pointer, any code which places the stack pointer in an invalid state could not use optimistic interrupt protection. For the Mach 3.0 kernel, there are no such sequences on the Omron Luna88k, DECstation, or DEC Alpha.

In the optimistic case (the protected sequence runs without interruption), protection overhead is minimal. One variable is set before the critical section, and at the end of the critical section that variable is reset and another variable (corresponding to the interrupt continuation) is checked. In the Omron Luna 88k implementation, this corresponds to two stores, one load and a test, all of which are executed by the processor at full speed. Not only is protection overhead small, it also scales with processor performance.

4 Performance

We have implemented optimistic interrupt protection in the Mach 3.0 kernel on the Omron Luna88k and Mips R3000 DECstation series. In both architectures, the interrupt continuation consisted of the register state at the time of the trap and a few additional words of state. Implementation took less than 3 days and and required no modification to assembler code routines. Table 2 shows the fast path overhead for interrupt management on these architectures. This sequence replaces the interrupt mask manipulations of Table 1. By using optimistic interrupt protection the length of the interrupt management path has been roughly halved.

Machine	Processor	Instructions	Cycles
Luna88k	Motorola 88100	51	51
DECstation 5000/120	R3000	31	31
DECstation 5000/200	R3000	31	31

Table 2: *Overhead of virtual interrupt mask manipulation. Cycle counts are estimated, assuming no cache misses. The Luna88k is a multiprocessor, so the virtual interrupt state is maintained on a per CPU basis. Most of the extra 20 cycles of overhead on the Luna88k are directly attributable to multiprocessor induced array indexing computations.*

To measure the impact of optimistic interrupt protection, we measured the performance of the Mach interprocess communication path. This path has already been highly optimized and contains only one interrupt protected critical section [Draves et al. 91]. Table 3 shows the performance of a cross address space null RPC with conventional and optimistic interrupt protection. The performance gain is larger than suggested by Tables 1 and 2 due to the idealized nature of those numbers. Both tables assume no TLB misses, cache misses, invalidation traffic or write buffer stalls; in practice, operating system code incurs a large contribution to cycles per instruction from all these factors [Chen & Bershad 93]. The reduction in path length and number of memory references in the interrupt management path therefore produces a greater than predicted benefit.

5 Related Work

One of the fundamental design decisions in an operating system is how to handle coordination between synchronous and asynchronous event handlers. Synchronous events happen within the context of the current execution stream (e.g, a system call), while a given asynchronous event can occur in the context of any instruction stream (e.g, I/O completion interrupts). Three approaches have been taken: interrupt masking as previously described, non-preemptable handlers, and lock-free synchronization.

Machine	Conventional	Optimistic	Speedup	Cycles saved
Luna88k	4400	4225	5.3%	175
DECstation 5000/120	2140	1840	14%	300
DECstation 5000/200	1234	1198	2.9%	36

Table 3: *IPC performance. Shown are the cycles for a null RPC with optimistic and conventional interrupt management. One cycle on the Luna88k is 40 nanoseconds, so IPC latency is reduced by 7 microseconds from 176 to 169. The cycle times on the 5000/120 and 5000/200 are 50 and 40 nanoseconds respectively.*

In the non-preemptable approach, both synchronous and asynchronous event handlers run uninterruptably to completion. The V kernel and many real time systems follow this approach [Berglund 86, Stankovic & Ramamritham 88]. Unfortunately, non-preemptable interrupt handlers impose serious constraints on handler structure: all handlers must be short to ensure that the latency of high priority events is low, and handlers cannot containing blocking operations (e.g. device status register polling). While this approach can lead to a high performance operating systems, difficulties inherent in this code style have prevented its widespread use.

Recent research has demonstrated the use of highly concurrent lock-free data structures [Herlihy 90, Wing & Gong 92]. A system using lock-free synchronization can be free from data corruption, deadlock and priority inversion even in the case of interrupts [Massalin & Pu 91]. In addition, lock-free data structures provide the necessary synchronization for both multiprocessors and non-preemptive execution. Consequently, lock-free data structures suggest an attractive approach for structuring operating systems. Unfortunately, lock-free data structures can require special synchronization hardware that is neither generally available nor inexpensive [Herlihy 91, Motorola 90][1]. Recently, researchers have proposed architectural modifications to efficiently support lock-free operations [Herlihy & Moss 93].

The division of synchronization mechanisms into an inexpensive optimistic and (relatively more) expensive pessimistic case has been applied elsewhere. Restartable atomic sequences offers a mechanism for constructing efficient user-level synchronization primitives in a preemptively scheduled environment [Bershad et al. 92] .

6 Conclusions

Optimistic interrupt protection is an application of optimistic synchronization to interrupt priority management in operating system kernels. It provides the same semantics as traditional interrupt management with much less overhead. A measurable speedup of the IPC path in the Mach 3.0 microkernel was obtained by using this technique. The method is applicable to any kernel that uses interrupt masking to guarantee data integrity.

References

[Accetta et al. 86] Accetta, M. J., Baron, R. V., Bolosky, W., Golub, D. B., Rashid, R. F., Tevanian, Jr., A., and Young, M. W. Mach: A New Kernel Foundation for Unix Development. In *Proceedings of the Summer 1986 USENIX Conference*, pages 93–113, July 1986.

[Anderson et al. 91] Anderson, T. E., Levy, H. M., Bershad, B. N., and Lazowska, E. D. The Interaction of Architecture and Operating System Design. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 108–121, April 1991.

[Bell et al. 82] Bell, C. G., Newell, A., and Siewiorek, D. P. *Computer Structures:Principles and Examples*, pages 110–128. McGraw-Hill, 1982.

[Berglund 86] Berglund, E. J. An Introduction to the V-System. *IEEE Micro*, 10(8):35–52, August 1986.

[Bershad et al. 92] Bershad, B. N., Redell, D. D., and Ellis, J. R. Fast Mutual Exclusion for Uniprocessors. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.

[Chen & Bershad 93] Chen, J. B. and Bershad, B. N. The Impact of Operating System Structure on Memory System Performance. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, December 1993. To appear.

[Cheriton 88] Cheriton, D. R. The V Distributed System. *Commun. of the ACM*, 31:314–333, March 1988.

[Custer 93] Custer, H. *Inside Windows NT*. Microsoft Press, Redmond, WA, 1993.

[DEC 92] DEC. *DECChip 21064-AA RISC Microprocessor Preliminary Data Sheet*. Digital Press, Manyard, MA, 1992.

[Draves et al. 91] Draves, R. P., Bershad, B. N., Rashid, R. F., and Dean, R. W. Using Continuations to Implement Thread Management and Communication in Operating Systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 122–136, October 1991.

[Herlihy & Moss 93] Herlihy, M. and Moss, E. Transactional Memory - Architectural Support for Lock Free Data Structures. In *The 20th Annual International Symposium on Computer Architecture*, May 1993.

[1] With only minimal hardware support (e.g, load linked / store conditional), implementation of nontrivial data structures (doubly linked lists, trees) requires substantial data copying. In addition, when these primitives have been implemented, their performance is often quite poor (alpha L_l/S_c) - often as long as an uncached load and store [DEC 92].

[Herlihy 90] Herlihy, M. A Methodology for Implementing Highly Concurrent Data Structures. In *Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 197–206, March 1990.

[Herlihy 91] Herlihy, P. M. Wait-free synchronization. *ACM Transcations on Programming Languages and Systems*, 13(1):124(26), January 1991.

[Intel 90] Intel. *i486 Microprocessor Programmer's Reference Manual*. Intel, Mt. Prospect, IL, 1990.

[Leffler et al. 89] Leffler, S. J., McKusick, M., Karels, M., and Quarterman, J. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1989.

[Levy & Eckhouse 89] Levy, H. M. and Eckhouse, R. H. *Computer Programming and Architecture: The VAX-11 (2nd Edition)*. Digital Press, Bedford, MA, 1989.

[Massalin & Pu 89] Massalin, H. and Pu, C. Threads and Input/Output in the Synthesis Kernel. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 191–201, December 1989.

[Massalin & Pu 91] Massalin, H. and Pu, C. A Lock-Free Multiprocessor OS Kernel. Technical Report CUCS-005-91, Department of Computer Science, Columbia University, 1991.

[Morse et al. 82] Morse, S. P., Ravenel, B. W., Mazor, S., and Pohlman, W. B. *Computer Structures:Principles and Examples*, pages 615–646. McGraw-Hill, 1982.

[Motorola 90] Motorola. *MC88100 RISC Microprocessor User's Manual*. Prentice Hall, Englewood Cliffs, NJ, 1990.

[Ousterhout 90] Ousterhout, J. K. Why Operating Systems Aren't Getting Faster As Fast As Hardware. In *Proceedings of the Summer 1991 USENIX Conference*, pages 247–256, June 1990.

[Sites 92] Sites, R. L., editor. *Alpha Architecture Reference Manual*. Digital Press, 1992.

[Stankovic & Ramamritham 88] Stankovic, J. and Ramamritham, K. A New Hard Real-Time Kernel. In *Hard Real-Time Systems*, pages 361–370. IEEE, 1988.

[Wing & Gong 92] Wing, J. M. and Gong, C. A Library of Concurrent Objects. CMU CS TR 90-151, School of Computer Science, Carnegie Mellon University, 1992.