Multiprocessing with the Exokernel Operating System

by

Benjie Chen

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Bachelor of Science
and
Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2000

Author . . . .
Department of Electrical Engineering and Computer Science
February, 2000

Certified by . . . .
M. Frans Kaashoek
Associate Professor
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Multiprocessing with the Exokernel Operating System

by

Benjie Chen

Submitted to the Department of Electrical Engineering and Computer Science
on February, 2000, in partial fulfillment of the
requirements for the degree of
Bachelor of Science
and
Master of Engineering in Computer Science and Engineering

Abstract

Exokernel is a minimal operating system kernel that safely multiplexes hardware resources, while leaving all system abstractions to applications. An exokernel exhibits better performance and offers more functionality because applications can provide optimized system abstractions, at the user-level, based on their needs. Current design of the exokernel system, however, does not support multiprocessor architectures. This thesis presents a symmetric multiprocessing exokernel and demonstrates that unprivileged library implementation of operating system abstractions is viable on a multiprocessor system.

This thesis focus on three issues. First, it presents synchronization strategies used in kernel. Second, this thesis describes three new exokernel interfaces: message passing, kernel support for multithreading, and multiprocessor scheduling. Third, because exokernel applications do not trust each other, traditional synchronization primitives used to guard system abstractions, such as voluntary memory locks, do not function well. This thesis presents and evaluates a strategy for synchronization among untrusted processes. A multiprocessor exokernel and a synchronized library operating system result from this thesis. Performance analysis shows that the overheads of synchronization in both the kernel and the library operating system are small.

Thesis Supervisor: M. Frans Kaashoek
Title: Associate Professor

Multiprocessing with the Exokernel Operating System

by

Benjie Chen

Submitted to the Department of Electrical Engineering and Computer Science
on February, 2000, in partial fulfillment of the
requirements for the degree of
Bachelor of Science
and
Master of Engineering in Computer Science and Engineering

Abstract

Exokernel is a minimal operating system kernel that safely multiplexes hardware resources, while leaving all system abstractions to applications. An exokernel exhibits better performance and offers more functionality because applications can provide optimized system abstractions, at the user-level, based on their needs. Current design of the exokernel system, however, does not support multiprocessor architectures. This thesis presents a symmetric multiprocessing exokernel and demonstrates that unprivileged library implementation of operating system abstractions is viable on a multiprocessor system.

This thesis focus on three issues. First, it presents synchronization strategies used in kernel. Second, this thesis describes three new exokernel interfaces: message passing, kernel support for multithreading, and multiprocessor scheduling. Third, because exokernel applications do not trust each other, traditional synchronization primitives used to guard system abstractions, such as voluntary memory locks, do not function well. This thesis presents and evaluates a strategy for synchronization among untrusted processes. A multiprocessor exokernel and a synchronized library operating system result from this thesis. Performance analysis shows that the overheads of synchronization in both the kernel and the library operating system are small.

Thesis Supervisor: M. Frans Kaashoek
Title: Associate Professor

I would like to thank members of the PDOS group at MIT LCS for their support and help over the past year and half on this thesis. Particularly, Frans, Robert, Chuck, and Dave.

To all my friends and brothers at MIT and ZBT.

To my parents.

For Melina.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Traditional operating systems safely multiplex resources for applications by providing common abstractions applications can use. While these abstractions provide a portable and protected view of the system, they only approximate what applications need. Often they provide either too much or not enough functionality, thus hinder certain application's performance. The exokernel operating system addresses this problem by applying the end-to-end argument to operating systems: the kernel only multiplexes hardware and protects resources, leaving all software abstractions and resource management policies for user-level applications to define. To increase usability, most exokernel applications do not directly communicate with the kernel, but instead are linked with user-level libraries implementing operating system abstractions. Kaashoek et al [23] showed that exokernel applications exhibit better performance, allow more flexibility, and offer more functionality because traditional abstractions can be optimized based on application needs.

The goal of the thesis is to present a multiprocessor exokernel capable of supporting parallel applications, and to show that unprivileged library implementation of operating system abstractions is viable on multiprocessor systems. We are motivated by observations made by numerous researchers that traditional multiprocessor operating systems are not flexible enough to support high performance parallel applications. Poor integration between user-level threads and kernel [3], rigid communication abstractions [5], and the lack of application specific information in process scheduling [9], for example, can severely limit the performance gain of running a parallel application on multiple processors. On an exokernel operating system, however, these limitations may not exist because the appropriate scheduling decisions and communication abstractions can be implemented by each application.

1.1 Background

The goal of exokernel is to securely multiplex system resources to unprivileged applications. There are four important design principles.

1.1.1 Exokernel Provides Protection, Not Management

An exokernel protects, but does not manage, system resources. Through system calls, it exports functions that directly affect protection: allocation, sharing, and revocation of resources. When to allocate, when to revoke, and how to authenticate, on the other hand, are management policies left for applications to define.

For example, on an exokernel system, each execution context is known as an environment (represented using `Env` objects). An environment essentially represents a protection domain: it defines an address space and stores states for the current execution context. This abstraction is necessary for protected sharing of the processor and memory: unless explicitly specified to be, data and code in one address space are not visible in other address spaces, therefore a malicious application cannot tamper or destroy anyone else's program. On the other hand, while exokernel offers system calls for allocating, deallocating, and scheduling environments, it does not impose restrictions on the organization of the address space or scheduling of the environment. Those decisions are left for applications to make.

Dynamically created hierarchically-named capabilities [26] serve as another example that protection is decoupled from management. A process must specify a capability when using a system call to access a resource. Exokernel performs only the requested action if the capability supplied dominates the capability guarding the resource. Because capabilities on a resource are supplied by the application when an initial binding is created, an exokernel only enforces access rights, but the decision of who shall have access privileges (i.e., what the capability looks like) is left for applications to make.

1.1.2 Exokernel Exposes Hardware

An exokernel gives applications direct access to hardware, rather than encapsulating them in high-level abstractions. For example, traditional operating systems represent physical memory as virtual memory. Virtual to physical address translations are managed by the operating system. An exokernel, on the other hand, multiplexes physical memory directly. Virtual to physical address translations are managed by applications. Each physical page is exposed to applications through a `ppage` object, which stores a page number, status of the page (i.e., free, dirty, or used for buffer cache), capabilities, and ownership. Supplying the correct capabilities, an application can use system calls to choose which physical pages it wants to use, and where they should be mapped to. Page faults are handled by applications as well, allowing powerful user-level virtual memory abstractions to be constructed. For example, an application can exploit the extra three bits on each valid x86 page table entry (i.e., with present bit set) to perform copy-on-write; or it can elect to save network addresses in invalid page table entries (i.e., without present bit set) and perform remote paging in the page fault handler.

1.1.3 Exokernel Offers Basic Interfaces

Exokernel offer basic interfaces that applications can use as building blocks for higher level abstractions. These interfaces can have default behaviors, as long as they can be by overridden by applications. For example, exokernel provides a protected control transfer mechanism that allows context switching from one environment to another in a safe way: kernel changes the program counter to an agreed-upon value in the callee, donates the current time slice to the callee, and passes data to the callee through registers. Protected control transfer is basic enough that applications can use it to implement other control transfer abstractions, such as synchronous or asynchronous IPC. On the other hand, applications can choose not to use protected control transfer and implement IPC another way. For example, through shared memory and priority scheduling, as done on UNIX.

1.1.4 Unprivileged Extensibility Requires Protected Sharing

Some extensible systems [1, 8, 16, 30] allow applications to alter system behaviors by inserting application specific extensions. To ensure safety, these systems often apply restrictions on where and how extensions can be inserted and executed. In comparison, the exokernel approach to extensibility provides more power: an exokernel only securely multiplexes available hardware, leaving all system abstractions for applications to construct and use in library operating systems. Recursively applying this approach to extensibility, some abstractions defined by the library operating system place their states in shared memory, giving each application the option of implementing even more customized abstractions based on these states.

An important consequence of the library operating system structure is that applications cannot trust each other. This property stems from the fact that unlike traditional implementations of system abstractions, libraries are unprivileged and can be modified or replaced at will. Thus, when sharing a common resource, a process linked with a library cannot trust other processes to exhibit similar behaviors.

Because exokernel applications are not trusted, providing protected access to operating system abstractions become important. While placing abstractions in a privileged domain (e.g., inside a protected method [12] or a server) provides protection, it reduces extensibility and performance. Instead, an exokernel offers two mechanisms for implementing system abstractions in an unprivileged yet safe fashion. One, exokernel requires hierarchically-named capabilities be specified on each system call. In order to access a shared resource, a process must supply the correct capability [26]. This prevents a malicious process from gaining read or write privileges to protected memory pages. Two, an uniprocessor exokernel provides robust critical sections: inexpensive critical sections implemented by logically disabling software interrupts and using restartable atomic sequences on the rare occasion that the critical section is interrupted because of fairness violations [7]. Aside from negotiating concurrency, critical sections are crucial to protected sharing: when a process manipulates shared data in a critical section, it can depend on the shared data to remain consistent until the end of the critical section. Consequently, protected sharing can be provided if processes use defensive programming to verify the correctness of the shared data at the beginning of each critical section.

The advantage of placing abstractions in shared memory is extensibility. Sometimes, however, shared memory cannot be used. When application semantics require strict invariants to be preserved, shared resources are often placed in privileged domains. Forcing processes to use these domains can be done by implementing the domains as IPC servers, protected methods [12], or downloadable kernel modules [8]. For example, when processes share a file system that requires file modification times be accurate, disk operations must be guaranteed to modify file access time. Processes sharing this file system must either have mutual trust in each other, or share the file system through a privileged domain.

In three cases, however, placing abstractions in shared memory is sufficient. One, when processes have mutual trust, shared data can be placed in shared memory pages with capabilities only these processes own, hence safe from malicious applications. Two, shared memory can be used between processes with unidirectional trust. For example, sharing of resources between a parent process and an untrusted child process occur frequently in UNIX. Often, the parent process creates a resource for the child process to use, but does not care how the resource is being used after it is handed off to the child. In this case, if the child process corrupts the resource, it is at the expense of the child process itself. Network servers often follow this organization: a privileged process (e.g. `inetd`) accepts a network connections, forks, and then drops privilege in the child to perform actions on behalf of a particular user. While the parent and the child shares the connection, the state of the connection only matters to the child. Three, when mutually untrusted processes share an abstraction with verifiable or no invariants, this abstraction can be placed in shared memory as well. If a malicious process tampers with the shared resource, other processes can use defensive programming techniques to protect themselves. For example, Engler [13] describes a UNIX pipe implementation that uses a shared record to track the number of bytes in the buffer used for the pipe. While it would be desirable for this byte count to remain consistent, it is a software engineering, rather than protection, issue. Since there are no guarantees on the sensibleness of the data in the pipe, knowing how many bytes are in the pipe provides no gain.

1.2 Contribution of the Thesis

This thesis applies the exokernel principles to a multiprocessor computer. The two major contributions are:

1. This thesis presents a working multiprocessor exokernel that multiplexes multiple processors to applications.

2. This thesis demonstrates that unprivileged library implementation of operating system abstractions is viable on a multiprocessor system. It presents a simple multiprocessor library operating system, named VOS.

1.2.1 A Multiprocessor Kernel

A multiprogrammed system offers virtual parallelism to applications, where as a multiprocessor computer offers physical parallelism, in the form of multiple processors, to applications. Previous exokernels are all designed for uniprocessor computers. We present SMP-Xok, a working multiprocessor exokernel that multiplexes processors using three new interfaces: message passing between processes on different processors, kernel support for multithreading, and flexible multiprocessor scheduling.

1.2.2 Building Multiprocessor Library Operating Systems

In previous uniprocessor library operating systems, critical sections have been used extensively to provide protected sharing of abstractions in shared memory [11]. Providing critical sections in a multiprocessor library operating system, however, is more difficult. On an uniprocessor exokernel system, only one process is running at a time. Mutual exclusion can be easily provided by disabling software interrupts (not relinquishing control of the processor until critical section is completed), therefore temporarily disallowing other processes from running. On a multiprocessor system, however, multiple processes may be running concurrently. This implies that a critical section must be guarded using system-wide, not processor-wide, primitives. Existing multiprocessor synchronization techniques (e.g. memory locks and condition variables) are voluntary. Critical sections can be provided if processes are expected to use these techniques. Since exokernel applications are untrusted, this expectation cannot be made. Consequently, traditional synchronization techniques cannot be used.

This thesis introduces the Copy-Modify-Replace (CMR) algorithm that negotiates concurrency among untrusted processes: first, a private copy of the shared data, writable by one process but readable by all other processes, is obtained; then, the private copy of the shared data is modified; last, if no contention has occured, the private copy replaces the previous version of the data with a pointer modification. The CMR algorithm is optimistic. It assumes that malicious processes do not exist. When this assumption is proven false through correctness checks on the shared data and/or detection of a broken voluntary mutual exclusion protocol, the critical section is either terminated or retried.

1.3 Related Work

There is a plethora of research literature on multiprocessor operating systems, most focus on different synchronization strategies and techniques. Several papers influenced this thesis heavily.

Anderson et al [2, 3, 4, 5] discuss operating system support for multiprocessor computers. In particular, much emphasis has been given to the implications of different designs and implementations of thread, communication, and scheduling systems. Three contributions from their work influenced the design and implementation of SMP-Xok. One, a decentralized approach to structuring system services on a multiprocessor computer removes synchronization bottlenecks, thus improves global performance. Two, user-level threads are more efficient than kernel threads because thread operations do not require kernel crossing. Three, flexible scheduling support and explicit event notification can improve the performance and functionality of thread and communication packages.

Herlihy [19] proposes a general methodology for implementing concurrent data structures with non-blocking synchronization. When unexpected delays (e.g., page faults, hardware interrupts, and scheduling preemption) and/or failures occur to a process, blocking synchronization can prevent other faster and healthy processes from entering critical sections guarded by locks the failing process holds. This starvation causes performance degradations. Non-blocking synchronization techniques solve this problem by taking the approach that critical sections should be provided by individual processes: when entering a critical section, each process optimistically assumes that there are no contention. Anticipating that the assumption may be false, a duplicate copy of the shared data is used to prevent modifications by contending processes. At the end of the critical section, if the assumption is shown to be false, the critical section is retried. Otherwise, the duplicated data replaces the real data (usually done through a pointer swap). This approach allows a process to complete its critical section if other processes have been delayed or crashed. This thesis presents a similar synchronization technique that negotiates concurrency among untrusted processes. In our case, in addition to fighting delays and failures, we also try to prevent malicious processes from tampering the data structure while another process is in a critical section.

1.4 Thesis Organization

Chapter 2 describes synchronization strategies in SMP-Xok. Chapter 3 describes three new interfaces offered by SMP-Xok: message passing, kernel support for multithreading, and multiprocessor scheduling. Chapter 4 describes synchronization issues in a library operating system, focusing on an algorithm that provides synchronization among untrusted processes. Chapter 5 presents some experimental results. Chapter 6 concludes.

1.5 Software Availability

Source code of SMP-Xok, the multiprocessor exokernel, and VOS, the multiprocessor library operating system, can be downloaded from the exokernel homepage: http://www.pdos.lcs.mit.edu/exo/.

Chapter 2

Kernel Synchronization

SMP-Xok uses spinlocks and read-write locks to synchronize kernel data structures. Three other synchronization mechanisms are often employed in a multiprocessor kernel: semaphores, condition variables and non-blocking algorithms. Semaphores and condition variables are well suited for multithreaded preemptive environments, where a thread waiting for a busy lock can be suspended to make room for other threads (e.g. the thread holding the lock). An exokernel is non-preemptive, hence these mechanisms cannot be deployed. Furthermore, since non-blocking algorithms are designed to combat delays from preemption, they do not provide visible benefit.

This chapter presents SMP-Xok's synchronization strategy. We focus on synchronization between kernels running on different processors concurrently. We discuss synchronization issues between kernel and applications in a later chapter.

2.1 Overview of Synchronization Strategies

The granularity of synchronization directly contributes to the performance of a synchronized kernel. With coarse-grained locking, a long critical section or a large amount of shared data are protected by a single lock. There are two advantages to using coarse-grained locking. One, the overhead of synchronization is small because only one lock acquire operation is needed for each critical section. Two, it is simple to reason and implement. Coarsed-grain locking reduces computation parallelism, however. When long critical sections are used, contending processors are forced to wait for a long time, thus decreasing processor utilization. When a lock is used to guard a large amount of data, critical sections that use different subsets of the data appear to contend with each other. Fine-grained locking solves these two problems: a critical section can be broken up into smaller critical sections; and data can be broken down into smaller pieces, each synchronized by a separate lock. Fine-grained locking increases the overhead of synchronization, on the other hand, because more lock operations are needed. Hence, striking a balance between when to use fine-grained locking and when to use coarse-grained locking is very important.

A good synchronization strategy minimizes lock contention, latency of lock operations, and the overall overhead of synchronization. To achieve these goals, SMP-Xok applies the following principles when deciding what form of synchronization to use:

- **When data are localized to each processor, no synchronization is needed**. For example, each processor has its own quantum vector. Kernel schedulers running on different processors do not need to synchronize against each other.

- **When contention is high, but caused mostly by readers, use read-write locks.** For example, each process has a list of capabilities it can use to access system resources. While this list is seldomly updated, it is read on every system call. Thus, using a read-write lock reduces locking bottleneck because in most cases the lock acquire operation does not block. SMP-Xok also uses read-write locks to synchronize the use of packet filters.

- **When contention on a shared resource is high and mostly among writers, use fine grained locking on the resource.** For example, synchronization of the `ppage` objects that represent physical memory pages are provided by spinlocks, one on each physical page.

- **When contention on a shared resource is low or when it is on the slow path, use coarse-grained locking with spinlocks.** For example, the queue of IPC messages for each process is rarely contented, and the entire queue is guarded by one spinlock.

2.2 Synchronization Algorithms

2.2.1 Wait-in-Cache Spinlocks

Synchronization using spinlocks has been heavily studied in past literature. There are two major concerns. One, in a preemptive environment, the use of spinlocks can cause significant performance degradation and priority inversion when the process or thread holding a spinlock is preempted due to scheduling decisions, page faults, or hardware interrupts [18, 25, 31]. This is not a concern in SMP-Xok, however, because exokernels are non-preemptive. Two, on certain architectures, trivial implementations of spinlocks can slow down critical sections and cause communication bottleneck [2]. We discuss this problem in the next section. This section describes the implementation of a wait-in-cache spinlock.

Each spinlock in SMP-Xok contains a lock byte and an usage counter. The usage counter of a spinlock can be utilized by applications to provided synchronized read of kernel data structures (see Section 4.2.1). A simple spinlock implementation (Figure 2-3 presents algorithm in C) uses an atomic test-and-set primitive to acquire a lock sitting in main memory. If the lock is busy, the acquire operation blocks until the lock can be reacquired. To avoid unnecessary memory writes issued by the x86 test-and-set implementation (presented

```
char  test_and_set(char* lock)
{
   char t = *lock;
   *lock = 1;
   return  t;
}
```

Figure 2-1: An implementation of atomic test-and-set. This code segment models the `xchg` instruction on x86 architectures.

```
unsigned int  double_cmpxchg(unsigned int *p1, unsigned int *p2,
                             unsigned int  v1, unsigned int  v2,
                             unsigned int  n1, unsigned int  n2)
{
   if (*p1 == v1 && *p2 == v2)
   {
      *p1 = n1; *p2 = n2;
      return  1;
   }
   return  0;
}
```

Figure 2-2: Specification for atomic double compare-and-swap: atomic double compare-and-swap can be implemented using the `cmpxchg8b` instruction on x86 architectures.

```
typedef struct
{
    char lock;
    unsigned int owner;
    unsigned long usage;
} spinlock_t;

per processor variable
   unsigned int my_cpu_id;    /* processor identification */

void  spinlock_acquire(spinlock_t *s)
{
   while (s→lock == 1 || test_and_set(&(s→lock)) == 1);     /* first wait in cache, then try to do test and set */
   s→owner = my_cpu_id;
   s→usage = s→usage + 1;
}

void  spinlock_release(spinlock_t *s)
{
   s→usage = s→usage + 1;
   s→lock = 0;
}
```

Figure 2-3: Wait-in-cache spinlock

in Figure 2-1), the acquire operation first waits for the value of the lock to change in its cache. When the lock is released, the cache coherence protocol, implemented in hardware, updates the cache.

2.2.2 Queue-based Spinlocks

When used in short critical sections, wait-in-cache spinlock causes significant performance degradation. This possible degradation is triggered by expensive bus transactions issued by the Pentium Pro hardware to preserve cache coherence across multiple processors [21]. Consider the following sequence of events.

Refer to Figure 2-4. Suppose a number of processors are spinning reading the lock variable in their caches. When the lock is released, the memory write in the lock release procedure triggers a Read For Ownership (RFO) bus transaction, which invalidates all cached copies of the lock. Each processor will then incur a read miss to fetch the new value back into its cache. Each read miss causes a Bus Read Line transaction. The first Bus Read Line transaction is snooped by the processor that just released the lock, and triggers an Implicit Write Back response. This response provides the updated lock value to the requesting processor, and also updates the memory. Subsequent Bus Read Line transactions obtain data from the memory.

Read misses are satisfied serially. Some processors will receive a shared copy of the lock before the lock is reacquired. Each of these processors will perform a test-and-set operation, and one will succeed. The test-and-set operation issues a RFO transaction because it needs to modify the lock [1]. Thus, each test-and-set operation again invalidates every cached copy of the lock, forcing processors that had been spinning in their caches after a failed test-and-set attempt to read miss again. Eventually, the last processor performs a test-and-set, forcing every other processor to incur one last read miss. These bus transactions not only increase the latency of each lock operation, but also may slow down the processor in critical section by consuming bus bandwidth.

This sequence of events is depicted in Figure 2-4. In this figure, bus transactions, events on four processors, and the status of the cache line (**I**nvalid, **S**hared, **E**xclusive, or **M**odified) containing the lock variable are shown on corresponding time lines, in temporal order. In this example, processor 1 re-acquires the lock after processor 0, initially holding the lock, releases it. Processor 2 and 3 contend for bus bandwidth after processor 1 has already acquired the lock. The shaded region shows unnecessary bus transactions: processor 2 and processor 3 each performs an unnecessary test-and-set, and processor 3 causes processor 2 to incur an additional read miss.

Since each processor waiting on the lock can cause all other processors to incur cache misses by performing an unnecessary test-and-set, the number of bus transactions caused by the hardware cache coherence protocol is proportional to the number of processors squared. For long critical sections, these bus transactions have insignificant effects on performance. For short critical sections, however, the bus transactions dominate

[1]Even if the implementation of test-and-set uses a compare-and-swap instruction where if the compare fails the swap never occurs, this RFO transaction still occurs. The Intel architecture requires the compare-and-swap instruction to be issued with a *LOCK* prefix to guarantee atomicity. When a processor receives a locked operation, it first executes a RFO transaction to bring the line into its cache for exclusive write.

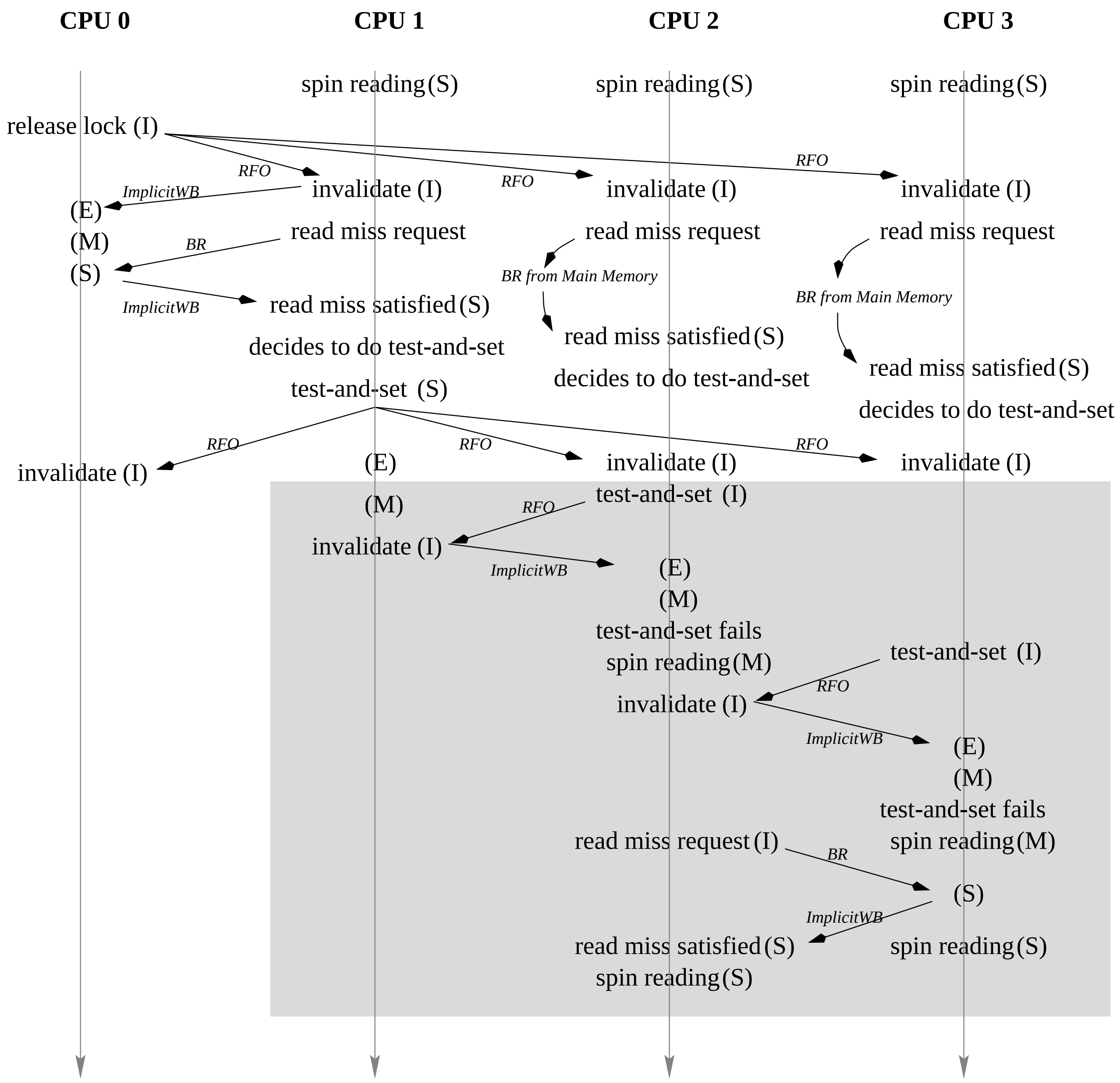

Figure 2-4: Bus transactions caused by the trivial implementation of spinlock: bus transactions are arrows with italic labels. RFO: read for ownership transaction. BR: bus read line transaction. ImplicitWB: implicit write-back response. Status of the cache line containing the lock variable is in parenthesis. S: Shared. I: Invalid. E: Exclusive. M: Modified. The shaded area represents unnecessary bus transactions.

performance, especially as the number of processors in the system increases.

Anderson et al. [2] propose several spinlock design alternatives to combat this performance degradation. SMP-Xok implements one of these alternatives, using a queue and a “ticket system.” Instead of waiting on a single lock variable, when a processor contends for a lock, it uses an atomic read and increment instruction to obtain an unique ticket. Each ticket contains a boolean variable, initially set to `false`. When a processor is allowed to enter a critical section, its ticket value is changed to `true` by the previous lock holder. Because ticket numbers are obtained using an atomic read and increment instruction, and the processor with the lowest ticket number is granted the lock first, tickets effectively orders all waiting processors into a queue. This algorithm is shown in Figure 2-5. Wisniewski et al. [31] also describes a similar algorithm.

This ticket mechanism reduces unnecessary bus transactions because a lock is never contended, but rather handed off to the next waiting processor. When tickets are allocated to different cache lines, each hand off triggers three bus transactions: a RFO transaction issued by the previous lock holder prior to changing the boolean value, a cache miss transaction issued by one waiting processor for reading the boolean value, and a RFO transaction issued by that waiting processor prior to reseting the boolean value. Hence, the overhead of each lock operation is constant.

2.2.3 Read-Write Locks

On some shared resources, read operations occur much more frequently than write operations. For example, SMP-Xok uses the Dynamic Packet Filtering system [14] to filter network packets. An application downloads packet filters for network packets it wants to receive. The kernel constructs a decision tree based on these filters. When a packet arrives, the destination of the packet can be determined by searching the decision tree using a simple backtracking scheme. While the decision tree is traversed for every packet received, it is modified only when new filters are downloaded.

When read operations dominate write operations, SMP-Xok uses read-write locks to negotiate concurrency. Read-write locks provide mutual exclusion for write operations, but allow multiple read operations to occur simultaneously when there are no write operations. This type of synchronization reduces bottleneck because most likely read operations can occur without spin waiting.

Figure 2-6 presents an implementation of read-write lock. This implementation uses an atomic double compare and swap operation (see Figure 2-2) to prevent multiple write operations or a read operation and a write operation to occur concurrently. SMP-Xok implements the atomic double compare and swap operation using the x86 `cmpxchg8b` instruction.

2.2.4 Non-blocking Techniques Are Not Useful

Non-blocking synchronization algorithms use atomic instructions such as double compare-and-swap or linked-load/store-conditional to optimistically update shared data with new values. When multiple updates occur

```
typedef struct
{
     bool tickets[P]; /* P: number of processors in the system */
     unsigned int next_ticket;
     unsigned long usage;
} queuelock_t;

initial values
   bool tickets[0] = true;
   bool tickets[1...P-1] = false;
   unsigned int next_ticket = 0;

per processor variables
   unsigned int my_cpu_id;    /* processor identification */
   unsigned int myTicket;     /* processor's ticket, one for each queuelock instance */

unsigned int  read_and_increment(unsigned int *v)
                /* atomic. models the xadd instruction on x86 architectures */
{
   t = *v
   *v = *v + 1
   return  t
}

void  queuelock_acquire(queuelock_t *q)
{
   myTicket = read_and_increment(&(q→next_ticket)) mod P;
   while (q→tickets[myTicket] == false);
   q→tickets[myTicket] = false;
   q→usage = q→usage + 1;
}

void  queuelock_release(queuelock_t *q)
{
   q→usage = q→usage + 1;
   q→tickets[(myTicket+1) mod P] = true;
}
```

Figure 2-5: A scalable spinlock implementation using queues and tickets.

```
typedef struct
{
    unsigned int nreaders;
    int writer;
    unsigned long usage;
} rwlock_t;

void  rwlock_read_acquire(rwlock_t *r)
{
    unsigned int  nreaders;
retry:
    while (r→writer != −1);
    nreaders = r→nreaders;
    if (double_cmpxchg(&r→nreaders, &r→writer, nreaders, −1, nreaders+1, −1) == 0)
        goto retry;
}

void  rwlock_read_release(rwlock_t *r)
{
    /* use x86 atomic decl to implement the following line */
    r→nreaders = r→readers − 1;
}

void  rwlock_write_acquire(rwlock_t *r)
{
    while (double_cmpxchg(&r→nreaders, &r→writer, 0, −1, 0, my_cpu_id) == 0);
    r→usage = r→usage + 1;
}

void  rwlock_write_release(rwlock_t *r)
{
    r→usage = r→usage + 1;
    r→writer = −1;
}
```

Figure 2-6: Implementation of read-write lock

concurrently, only one is permitted by the underlying hardware. This synchronization technique has been used mostly to reduce the adverse effects of preemption during a critical section [18, 19, 25], an impossible scenario under the non-preemptive exokernel.

Originally, we had intended to use non-blocking synchronization algorithms in SMP-Xok for a different reason: on data structures such as queues, stacks, and linked lists, a non-blocking operation has lower latency then spinlock acquire and release because a compare-and-swap instruction that changes the head of the queue, stack, or linked list atomically can often replace a lock acquire, the pointer manipulation, and a lock release. While this is true in theory, it is not the case in practice on the x86 architecture: we used non-blocking synchronization on several kernel queues (e.g., memory buffer queue for malloc) and observed no effects on the performance of kernel operations using those queues. The intuition behind this observation is that the performance gain of this trick are so small that they are easily dominated by cache performance, especially when processors contend for the critical section by repeatedly retrying. Subsequently, we have abandoned non-blocking synchronization algorithms in favor of the easy to understand spinlock and queuelock.

2.3 Implementation

This section describes the synchronization strategies employed for several data structures and algorithms within SMP-Xok.

2.3.1 Environments

An exokernel environment, represented by an `Env` object, describes a process or a thread. The `Env` object collects run-time statistics, store run-time states (e.g. registers during a kernel trap), maintain information such as environment ID and granted capabilities, and communicate information between application and kernel.

Synchronization of `Env` objects is trivial. Because an environment can only execute on one processor at a time, collection of run-time statistics and saving of run-time states are race free. Some data, such as environment ID and address space ID, are set once only, therefore do not need to be synchronized. Data such as the IPC message ring and wakeup predicates are rarely contented. They are synchronized with simple spinlocks. A read-write lock is used to synchronize the list of capabilities since read operations dominate write operations on this list.

For synchronization of the scheduling system, see Section 3.4.

2.3.2 Physical Memory

Each `ppage` object uses a separate spinlock. Additionally, the list of free pages is synchronized with its own lock. This locking granularity works well. Frequent operations on the list of free pages are short (insertion and removal), thus a single lock for the list suffice. Since physical memory usage for every environment varies, protecting more than one `ppage` object with each spinlock does not make sense: either there are too many `ppage` objects protected by the same spinlock, thus causing synchronization bottleneck; or there are few `ppage` objects per lock, and chances of accessing two physical pages using the same lock become small.

Fine grained locks yield low contention. Consequently, simple wait-in-cache spinlocks are used without sacrificing throughput. There are two benefits to this decision. The first benefit concerns memory. Each cache line is 32 bytes long on the Intel memory architecture. Because each ticket must reside on a separate cache line for improved cache performance, if a system supports 16 processors, it would need over 512 bytes of memory per lock for storing tickets. When resources are synchronized with fine grained spinlocks, this memory overhead could be overwhelming. For example, on a dual SMP system with 256 megabytes of physical memory, the number of `ppage` objects is well over 60,000. If each `ppage` object uses a scalable spinlock, the amount of memory overhead reaches 32 megabytes. Using the trivial implementation, each spinlock resides on a cache line, yielding a small memory overhead of less than 2 megabytes. The second benefit concerns latency. The trivial algorithm has lower lock acquire and release latencies because it does not need to perform enqueue and dequeue operations. Since latency is a concern with fine grained locking, the trivial algorithm becomes attractive.

The kernel malloc implementation uses pools of pre-allocated buffers to improve performance. Each pool is protected by a single spinlock. Operations on a pool are mostly insert and removal of buffers. These operations are short and not a source of bottleneck.

2.3.3 Device Interrupt

SMP-Xok uses the Advanced Programmable Interrupt Controller (APIC) to handle interrupts. An IO-APIC unit is programmed to route interrupts generated from external devices (e.g., network card) to the APIC of every processor, which in turn interrupts the processor [20, 22]. There are two types of interrupts: edge triggered and level triggered. A processor must acknowledge a level triggered interrupt by sending an End-Of-Interrupt (EOI) signal to the IO-APIC before another interrupt can be sent by the IO-APIC.

Synchronization of edge triggered interrupts, such as interrupts from devices on the ISA bus, is done using two test-and-set operations. This algorithm is shown in Figure 2-7. Edge triggered interrupts are asserted high for only a bus clock cycle. Hence, each edge triggered interrupt will be detected by the IOAPIC only once. When an APIC interrupts a processor, the kernel running on that processor first sets an interrupt pending bit using test-and-set. If no other processors are in the middle of handling this interrupt, the interrupt handler is called. Mutual exclusion of the interrupt handler is done by the second test-and-set. Furthermore, the interrupt handler executes in a while loop, and only stops running when the interrupt pending bit is no longer set.

Level triggered interrupts, such as interrupts from devices on the PCI bus, are handled differently. A level triggered interrupt is asserted high and remains high for more than one bus clock cycle. If the interrupt remains unmasked on the IO-APIC during this period, the IO-APIC may generate more than one interrupt

```
Each physical device has its own device data structure, containing a intr_pending byte
and a in_intr byte.

In interrupt handler:

  test_and_set (&intr_pending);
  while (intr_pending > 0 && test_and_set(&in_intr) == 0)
  {
    while (intr_pending > 0)
    {
      intr_pending = 0;
      run interrupt handling code
    }
    in_intr = 0;
  }
```

Figure 2-7: Algorithm for handling edge-triggered interrupts

messages to the processors. This effect is called *IRQ storming*. SMP-Xok solves this problem by masking the interrupt on the IO-APIC before sending the EOI signal. When the interrupt handler finishes, the interrupt is unmasked. A consequence of masking and unmasking the interrupt around the interrupt handler is that the interrupt handler is automatically race-free.

Each DMA based device uses a linked list of DMA descriptor objects to store packets. Each time a descriptor object is removed from the list (e.g., stored away for demultiplexing), a new one replaces it. On an uniprocessor, there is one pool of DMA descriptor ring. To avoid a potential bottleneck, SMP-Xok allocates a separate pool for each DMA device. Because devices are already synchronized, insertion and removal from local pools are race free.

Packets demultiplexed to applications are stored in packet rings. Each packet ring is synchronized with a separate spinlock.

2.4 Summary

SMP-Xok uses three types of locks to synchronize its data structures: wait-in-cache spinlocks, queue-based spinlocks, and read-write locks.

Wait-in-cache spinlocks use the test-and-set primitive to provide mutual exclusion for processors. Although easy to implement, when used in short critical sections, it causes significant performance degradation. This degradation is triggered by expensive bus transactions issued by the Pentium Pro hardware to preserve cache coherence.

Queue-based spinlocks use queues processors waiting to obtain a lock. When a lock is released, the next processor on the queue is notified. Queue-based spinlocks reduce unnecessary bus transactions because a lock is never contended, but rather handed off to the next waiting processor.

Read-write locks are deployed in SMP-Xok to synchronize read-only data structures or data structures that rarely changes.

Chapter 3

Exokernel Support for Multiprocessing

In addition to uniprocessor exokernel interfaces, a multiprocessor exokernel must multiplex additional processors for applications. SMP-Xok offers three new interfaces not found on uniprocessor exokernels: message passing through kernel maintained message buffers; multiple environments running within the same address space; and multiprocessor scheduling.

This chapter first describes inter-kernel messaging, a kernel to kernel communication mechanism used by SMP-Xok to preserve TLB coherence and propagate scheduling requests. It concludes with a presentation of each of the three new interfaces.

3.1 Inter-Kernel Messages

Occasionally, copies of the kernel running on different processors must communicate with each other. For example, virtual to physical address translations are cached on Translation Look-aside Buffers (TLBs) on each processor. The kernel is responsible for maintaining TLB coherence across multiple processors. Hence, when a translation is changed on one processor, other processors that are running in the same address space must be notified of the change. We discuss preservation of TLB coherence in more detail in Section 3.3.3. In this section, we present a kernel to kernel communication mechanism called *inter-kernel messaging*.

An inter-kernel message is a 64-bit message accompanied by a possible interruption of the processor receiving the message. Figure 3-1 shows what happens when an inter-kernel message is sent. Each processor is given an inter-kernel message queue. The sending processor of an inter-kernel message pushes the 64 bit message onto the receiving processor's message queue. An inter-processor interrupt is then generated by the sender. Upon receiving the interrupt, the receiving processor's interrupt handler dispatches messages in its message queue to specific handlers.

An inter-processor interrupt (IPI) is similar to a device interrupt, except the source of the interrupt is another processor rather than a device. Physically, a processor generates an IPI by talking to the local APIC

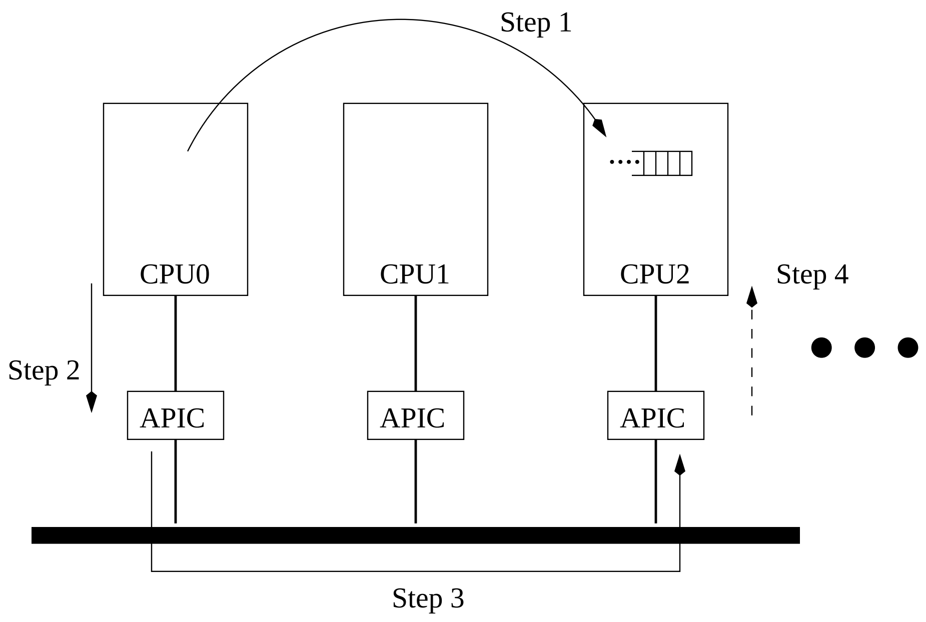

Figure 3-1: Generation and delivery of an inter-kernel message. Step 1: CPU 0 generates a message and puts it onto CPU 2's message queue. Step 2: CPU 0 asks its APIC to initiate an IPI targeted at CPU 2. Step 3: CPU 0's APIC sends IPI to CPU 2's APIC. Step 4: CPU 2's APIC interrupts CPU 2. This last step is unreliable.

(Advanced Programmable Interrupt Controller). The local APIC sends the IPI on a bus connecting all processors. The APIC for the target processor receives the IPI and interrupts the target processor. IPIs are unreliable: although an IPI will always reach the APIC of the target processor, occasionally the APIC may not deliver the interrupt to the processor.

Anticipating the possible loss of IPIs, a routine that checks the inter-kernel message queue for all pending messages is invoked at every timer interrupt. Upon discovering pending messages, each message is dispatched to the appropriate handler. This gives a bounded latency of 10 ms for processing the first message on the queue. Since lost IPIs are very rare, this occasional long latency is acceptable. Furthermore, unless IPIs are lost, the message queue is rarely full because each processor can handle a message as soon as it comes in. When the message queue becomes full, messages are dropped from the back.

An exokernel system does not receive interrupts while executing in kernel mode. A potential deadlock could occur if processor A is spin waiting for a response from processor B, and processor B is spin waiting for a response from processor A. This deadlock is avoided easily: when a processor spin waits for a response, it periodically checks its message queue for pending messages.

3.2 Inter-Process Communication through Message Passing

On an uniprocessor exokernel, fast inter-process communication (IPC) can be implemented using protected control transfer. Protected control transfer guarantees two important properties. First, to applications, a

```
/* a message ring entry */
typedef struct  msgringent
{
   struct msgringent *next;
   unsigned int free;
   char buf[32];
} msgringent_t;

unsigned int  sys_msgring_setring (msgringent_t head)
unsigned int  sys_msgring_modring (msgringent_t new, msgringent_t old, unsigned int replace)
unsigned int  sys_msgring_delring ()
unsigned int  sys_ipc_sendmsg (unsigned int e, unsigned int size, unsigned int address)

unsigned int  sys_ipi (unsigned int k, unsigned int c, unsigned int e)
```

Figure 3-2: IPC related system calls in SMP-Xok. The argument c is a processor ID. The argument e is an environment ID.

protected control transfer is atomic: once initiated it will reach the callee immediately. Second, the kernel will not touch callee saved registers, hence these registers can be used as buffers to pass data from the callee to the caller. IPC on a multiprocessor system, however, has different requirements. We examine why:

- Atomicity is impossible to achieve across processors because two processes communicating with each other may not be running concurrently.
- With appropriate scheduling support, using shared memory to implement IPC on a multiprocessor system can eliminate the cost of context switch [5]. Since each environment manages its own physical to virtual address translation, exokernel must provide a mechanism that applications can use to negotiate the physical address of a shared memory region.
- While protected control transfer can be used for that purpose, because it is upcall based, on a multiprocessor system it may force a process queued to run on one processor to migrate to a different processor. This is against the exokernel principle of providing only protection, but not management of resources.
- Alternatively, passing data between processes on different processors can be achieved through message passing using kernel maintained message buffers. SMP-Xok supports this form of message passing.

3.2.1 Message Passing With Kernel Maintained Buffers

An IPC message is 32 bytes long. SMP-Xok maintains an IPC message ring for each environment. The structure of the message ring is defined by the kernel, but the message ring object of each environment is created in user-space and downloaded into the kernel by that environment. This approach allows an environment to process IPC messages by reading and writing to the ring object directly, without making any system calls.

Figure 3-2 presents message passing related system calls. Semantics of these system calls are:

- **sys_msgring_setring:** downloads the specified ring into kernel as the IPC message ring for the current environment. The kernel does not use this ring directly. Instead, it constructs a duplicate ring using kernel virtual addresses (the entire physical memory is mapped into the higher half of each address space, accessibly only by the kernel), rather than application virtual addresses, as pointers in the ring. While constructing this second ring, the kernel verifies each pointer in the ring, making sure that it points to a region of the memory that the environment is allowed to modify. Having a separate ring prevents the ring structure or virtual address mapping to be changed to point to a privileged location of the memory.
- **sys_msgring_modring:** modifies a previously downloaded ring at the given pointer in the ring.
- **sys_msgring_delring:** removes a previously downloaded ring.
- **sys_ipc_sendmsg:** exokernel keeps a ring index for each message ring. If the designated environment's message ring has a free entry at its current ring index, the IPC message at the specified address is copied onto that entry. The ring index is subsequently incremented. If the message ring does not have a free entry at the original ring index, the message is dropped. Since the message ring is created in user space, an environment can free a message ring entry by setting its free field.

A sleeping environment can use a wakeup predicate [15, 23] to monitor its IPC message ring: a wakeup predicate that returns true whenever the IPC message ring becomes non-empty can be downloaded into the kernel before an environment goes to sleep. When an IPC message arrives at the ring, the kernel scheduler wakes up the environment.

3.3 Kernel Support for Multithreading

SMP-Xok allows multiple environments to execute within the same address space. On an uniprocessor exokernel, each environment runs within its own address space. The notion of multiple threads within an address space is left for the application to define. This decision was influenced by the observation that user-level threads, given adequate scheduling support, are more efficient [3, 5]. While this observation is still true on a multiprocessor system, physical processors must be multiplexed. Because an environment can only execute on one processor at a time, to support threads executing concurrently on different processors, multiple environments must be allowed to execute within the same address space, therefore sharing the same set of user-level threads.

3.3.1 Interface

SMP-Xok offers the following new system call:

- **sys_env_clone(unsigned int k)** - creates an environment that uses the same address space as the current environment. *k* is a capability index. The corresponding capability is used to protect the new environment and its Uarea object.

3.3.2 Implementation

The decoupling of environment and address space is done in three steps:

1. **Decouple page directory from address space.** On a multiprocessor exokernel, a separate copy of the kernel runs on each processor. Each kernel requires its own kernel stack and some physical pages to store per-kernel data (e.g., CPU identifier, page fault mode, etc). Instead of indexing an array of stack objects and physical pages, SMP-Xok uses a separate page directory for each processor. These page directories belong to the same address space: all virtual to physical address translations are shared among these directories, except those for kernel stack and per-kernel data. Page directories of the same address space are stored in an `AddrSpace` object, synchronized using a spinlock.

2. **Decouple environment from address space.** Instead of embedding a page directory, each `Env` object has a pointer to an `AddrSpace` object.

3. **Insert Uarea page and pick appropriate page directory on context switch.** Each environment has an Uarea page where data shared between the kernel and application are stored (e.g. addresses for upcall handlers). When a kernel context switches to an environment, it inserts that environment's Uarea page into the page directory for the current processor. That page directory is then used to perform the context switch.

   Since it is uncommon that different environments of the same address space will be running on the same processor, this page directory insertion is not always done. A tag is used to remember which environment last used a page directory. If the value of the tag matches the environment identifier, no insertion is done.

3.3.3 Preservation of TLB Coherence

When the physical address mapping of a virtual address is modified on one processor, the kernel is responsible for flushing the TLB entry for this virtual address on all processors running in the address space where the change occured. This procedure is called *TLB shootdown*.

Figure 3-3 presents the TLB shootdown algorithm used in SMP-Xok. The first **for** loop sends a TLB invalidate inter-kernel message to every processor running in the address space where the mapping change occured. The second **for** loop waits for ever TLB flush to complete before proceeding. The inter-kernel message contains a TLB_INVALIDATE command, and a pointer to a 32 bit location containing the virtual

```
unsigned int vaptr[P][P];
void tlb_invalidate (unsigned int va, unsigned int addrspace_id)
{
   int i;
   if (current address space ID == addrspace_id)
      invalidate local TLB;
   for (i=0; i<P; i++)
   {
      vaptr[my_cpu_id][i] = 0;
      if (i == my_cpu_id) continue;
      if (address space of processor i == addrspace_id)
      {
         vaptr[my_cpu_id][i] = va;
         send inter-kernel message to processor i, encoding the TLB_INVALIDATE
            command and &vaptr[my_cpu_id][i] in the 64 bit message;
      }
   }
   for (i=0; i<P; i++)
   {
      while (vaptr[my_cpu_id][i] != 0)
         check inter-kernel message queue and process pending message
   }
}
```

Figure 3-3: The TLB shootdown algorithm

address whose mapping has been changed. Upon receiving a TLB_INVALIDATE inter-kernel message, a kernel flushes the TLB entry for the given virtual address, then writes a zero to the address contained in the message. This last step tells the sending processor that a TLB flush is completed.

3.4 Multiprocessor Scheduling

Traditional operating systems multiplex the CPU through a priority-based scheduler implemented in kernel, oblivious to the run-time behaviors of applications. Consequently, the scheduling needs of an application cannot be satisfied [3, 6, 9]. Exokernel solves this problem by exporting a minimal scheduling abstraction to its applications [15]. Scheduling can be done either in a centralized fashion, through a hierarchical scheduler applications, or in a decentralized fashion, by applications themselves.

An uniprocessor exokernel represents a processor as a linear vector, where each element corresponds to a time slice, or quantum. An application, supplying the correct capability, can schedule itself on one or more quanta on the vector. The vector provides the notion of execution order and time bound: scheduling is done round-robin by cycling through the vector quantum by quantum, and invoking the application scheduled at each quantum. At the end of the quantum, kernel delivers a software interrupt to the application by upcalling

unsigned int sys_quantum_alloc (**unsigned int** k, **int** q, **unsigned int** c, **unsigned int** e)
unsigned int sys_quantum_set (**unsigned int** k, **int** q, **unsigned int** c, **unsigned int** e)
unsigned int sys_quantum_free (**unsigned int** k, **int** q, **unsigned int** c)
unsigned int sys_quantum_get (**unsigned int** k, **unsigned int** c)
unsigned int sys_cpu_revoke (**unsigned int** k, **unsigned int** c, **unsigned int** e)

Figure 3-4: Scheduling related system calls in SMP-Xok. The argument c is a processor ID. The argument q is a quantum ID. The argument e is an environment ID.

to a known address. The application is then given a short grace period to give up the processor before being forcefully interrupted.

A multiprocessor exokernel extends this abstraction: SMP-Xok represents each processor with a single quantum vector. An exokernel application can schedule itself on quanta from any one or all vectors. This abstraction can be used to provide latency or throughput. A compute-intensive application with a high amount of parallelism can create and schedule a thread to run on each processor, therefore using concurrency to improving execution latency. On the other hand, applications with limited parallelism can schedule themselves evenly across all processors to achieve better throughput.

3.4.1 Interface

Figure 3-4 presents scheduling related system calls in SMP-Xok. The semantics of the these system calls are:

- **sys_quantum_alloc:** allocates quantum *q* on processor *c* for environment *e*. If *q* is not free, return error.
- **sys_quantum_set:** replaces the environment scheduled for quantum *q* on processor *c* with environment *e*. If *q* is not already scheduled, return error.
- **sys_quantum_free:** deallocates quantum *q* on processor *c*. If *q* is not free, return error.
- **sys_quantum_get:** return the quantum number of the current quantum on processor *c*.

Each of these system calls take, as one of its arguments, a capability number *k*. The kernel uses *k* to index the list of capabilities owned by the calling environment. During a **sys_quantum_alloc** call, the supplied capability is copied onto the newly allocated quantum. Subsequent **sys_quantum_set** and **sys_quantum_free** calls must supply a capability that dominates the capability on this quantum.

The **sys_cpu_revoke** system call reschedules environment *e* if it is running on processor *c*: if environment *e* has not received a software interrupt during the duration of the current quantum, processor *c* generates the software interrupt. This system call serves as a resource revocation [15] mechanism for multiprocessor scheduling. On an uniprocessor, this service is not necessary as only one environment is running at a time.

Revocation is implemented using inter-kernel messaging (see Section 3.1). When **sys_cpu_revoke** is invoked on a processor, the local kernel checks if environment *e* is indeed running on processor *c*. If so, it

```
void  sys_cpu_revoke  (unsigned int k, cpuid c, envid e)
{
    if (cpu[c].current_process.id != e)
        return;
    if (current_process.capabilities[k] does not dominate remote quantum)
        return;
    send inter-kernel message to c with e encoded in the 64 bit message
}
```

On processor c, in the inter-kernel message handling routine:

```
    if (current_process.id != e)
        continue;
    if (current_process already been notified of a revocation)
        continue;
    revoke_processor();                 /* revoke_processor() procedure sends software interrupt to current process */
```

Figure 3-5: An algorithm for revoking a process running on another CPU

sends an inter-kernel message to *c*, with *e* encoded in the message. Upon receiving the message, processor *c* ensures that *e* is running, and generates a software interrupt if one was not sent earlier. Checking to ensure that *e* is running on *c* occurs twice, once on the processor the original system call is invoked, once on processor *c*. Only the second one is necessary. The first check alone would result in a race condition, we do it solely to cut down generating unnecessary interrupts. This algorithm is depicted in Figure 3-5.

sys_cpu_revoke also takes a capability number *k*. The corresponding capability must dominate the capability on the quantum that the reschedule environment is scheduled on.

3.4.2 Implementation

Each quantum vector is implemented as a vector of `quantum` objects. The `quantum` data structure contains the identifier of the environment scheduled to run at this quantum and the number of clock ticks accumulated by this environment during the duration of the quantum. There are several places that a quantum vector is modified: in the system calls exported to manage the vector and in a kernel's round-robin scheduler. The system calls are synchronized using spinlocks. The kernel round-robin scheduler does not require synchronization for two reasons. First, quantum vectors are localized to each processor, therefore concurrent kernel schedulers do not contend with each other. Second, for each `quantum` instance, the Intel memory architecture guarantees the environment ID variable and the clock ticks counter, both 32 bits, can be read atomically. Because system calls manipulating the quantum vectors only modify these two variables, no synchronization is necessary.

An exokernel allows an environment to be running only on one processor at a time, because the data structure representing the environment is used to communicate and store information between the thread of computation in the environment and the kernel that the environment is running on. When a kernel scheduler encounters a quantum with an environment currently running on another processor, the quantum is skipped.

SMP-Xok does not guarantee kernel schedulers running on different processors are synchronized with each other. For example, if one processor is currently running using quantum number one, other processors may be using quanta of different positions on their respective vectors. This asymmetry is due to the non-uniform load on each processor.

3.4.3 User-level Scheduling

Hierarchical schedulers [17, 29] can be used on uniprocessor systems to implement different scheduling policies. Exokernel systems offer a *yield* interface for hand-off scheduling. A scheduler for a process group can allocate a set of quantum and schedule its processes via *yield*. Subsequently, each process can further schedule other processes or threads. A *yield* operation involves a kernel call, a context switch, and an upcall. On a dual Pentium Pro 200Mhz machine running SMP-Xok, it takes 4 microseconds. On quanta of 80 milliseconds, this is an acceptable overhead.

On a multiprocessor system, user-level schedulers can be similarly constructed. A multithreaded scheduler can allocate a set of quantum across different processors. Each thread of the scheduler runs on a different processor. These scheduling threads can share a single run queue, or use per-processor run queue and work stealing to dispatch other processes [4, 10]. Synchronized scheduling (e.g., gang scheduling [27]) can be done by using the **sys_cpu_revoke** interface.

3.5 Summary

SMP-Xok uses inter-kernel messaging, a kernel to kernel communication mechanism implemented with Intel's inter-processor interrupts. Inter-kernel messaging is used for two purposes: preserve TLB coherence across processors and actively notify another processor of scheduling requests.

A multiprogrammed system offers virtual parallelism to applications, where as a multiprocessor computer offers physical parallelism, in the form of multiple processors, to applications. SMP-Xok offers three new interfaces for multiplexing processors: message passing; multiple environments running within the same address space; and multiprocessor scheduling.

Chapter 4

Synchronization in a Multiprocessor Library Operating System

The atomicity of a critical section dictates that when a process enters a critical section to modify a shared state, the state cannot be changed by another process until the critical section ends. This guarantee enables an efficient implementation of protected sharing. Upon entering a critical section, a process can use defensive programming techniques to verify the correctness of the shared state. If the state is correct at the beginning of a critical section, then a process can expect the state to be correct at the end of the critical section as well. In previous uniprocessor library operating systems, aside from negotiating concurrency, critical sections have been used extensively to implement protected sharing of abstractions in shared memory [11].

On uniprocessor exokernel systems, inexpensive critical sections are implemented by logically disabling software interrupts and using restartable atomic sequences on the rare occasion that the critical section is interrupted because of fairness violations [7]. Disabling software interrupts, however, does not work on a multiprocessor system because processes running on different processors may concurrent access the same resource. Hence, a system-wide, rather than processor-wide, synchronization technique must be employed to provide critical sections on a multiprocessor system. Popular system-wide synchronization techniques are memory locks, semaphores, or optimistic non-blocking algorithms. Because exokernel applications are also untrusted, these voluntary techniques cannot be used.

Synchronization between untrusted processes differ from traditional synchronization problems only in that malicious processes exist. Malicious processes can do two things. One, while in a critical section, malicious processes can tamper with the shared data and therefore violate the atomicity guarantee of a critical section. Two, malicious processes can perform denial of service attacks by never relinquish rights to a critical section. These two problems are similar to the two problems that Herlihy's non-blocking synchronization techniques [19] address: delays and failures.

When unexpected delays (e.g. page faults, hardware interrupts, and scheduling preemption) and/or failures occur to a process, blocking synchronization can prevent other faster and healthy processes from entering critical sections guarded by locks the failing process holds. This starvation causes performance degradations. Non-blocking synchronization techniques solve this problem by taking the approach that critical sections should be provided by individual processes: when entering a critical section, each process optimistically assumes that there are no contention. Anticipating that the assumption may be false, a duplicate copy of the shared data is used to prevent modifications by contending processes. At the end of the critical section, if the assumption is shown to be false, the critical section is retried. Otherwise, the duplicated data replaces the real data (usually done through a pointer swap). This approach allows a process to complete its critical section if other processes have been delayed or crashed.

In a library operating system, denial of service attacks on a critical section are analogous to unexpected delays and failures. Preventing malicious tampering of shared data can be considered as preventing modifications by contending processes. There are three differences. One, duplicated copies of the shared data, made by each process that wants to modify the data, need to be protected from other processes to prevent malicious tampering. Two, because previous modifications to the shared data cannot be trusted, the shared data must be validated before use. If the shared data is found to be incorrect, the question of how to continue should be left for programmers to decide. Three, retries should be done carefully to prevent starvation by malicious processes.

This chapter presents a synchronization technique that captures these simililarities and differences. Like Herlihy's algorithm, this technique has three steps: first, a private copy of the shared data, writable by one process but readable by every other processes, is obtained; then, the private copy of the shared data is modified; last, if no contention has occured, the private data replaces the previous version of the shared data with a pointer modification. While the copy step can be expensive, we demonstrate that it can be avoided in several common cases. Additionally, we describe how this technique is used in implementing concurrent and protected sharing of TCP sockets.

We do not address synchronization issues within a library or between trusted processes. Such synchronization can be trivially provided with voluntary techniques (e.g., those described in Chapter 2).

4.1 An Algorithm For Synchronization Between Untrusted Processes

We introduce the Copy-Modify-Replace (CMR) algorithm for synchronized access to shared data among untrusted processes. before entering a critical section, a process makes a copy of the shared state. While in critical section, the process works on this copied version, free of tampering. After the critical section, the process "replaces" the shared state with the copied version, by modifying a pointer. To negotiate concurrency, CMR assumes that no malicious processes exist: it uses a voluntary synchronization protocol, such as memory lock, to provide mutual exclusion for cooperative processes. If this assumption is false, the algorithm recovers either through validating the shared data or by detecting lock failures. In a later section, we present a variant of the CMR algorithm that uses non-blocking synchronization instead of memory locks.

Figure 4-1 presents pseudo-code for the CMR algorithm. We use three kinds of variables to model sharing of memory in an operating system. A global variable can be read and modified by every process. This is equivalent to placing the variable in unprotected shared memory. A private variable can only be read and modified by the one process. This models a program variable in the address space of that process. A protected variable can be read by every process, but can only be modified by one process. This is equivalent to placing the variable on a memory page mapped read-write in the address space of one process and read-only in address spaces of all other processes.

When a process creates a piece of data, the content is stored in `data[i].buf`, where `i` is the ID of the creator. When this data becomes shared, the initial value of `latest` is set to `i` by the creator. When a process P wishes to modify the shared data, it copies content of `data[latest].buf` to `data[P].buf` (lines A3 to A10). Process P then modifies `data[P].buf` (lines A14-A16). Since `data[P].buf` is protected, no other process can modify it. Lastly, to replace the shared data with the modified version, `latest` is changed to P (line A17). By using the `latest` variable, we avoid having to copy the shared data twice.

To ensure graceful recovery from corrupted data, after a copy of the shared data is obtained from an untrusted process (line A11 determines the trust relationship), the `validate` procedure is called to verify its correctness (line A12). If verification fails, the `bail` procedure is called (line A13). The implementations of `validate` and `bail` are application specific. Generally, it would be a good idea to throw an exception in the `bail` procedure to warn user that a malicious process was detected.

The `mem_barrier()` routine is needed to serialize reading of the version number and the memory copy. The Pentium Pro memory architecture performs read operations speculatively [22]. Since the version number is updated on another processor, the most recent update to the version number, although already retired by that processor, may be sitting in that processor's pending store queue instead of the global memory image (a combination of cache and main memory, but not the pending store queue), therefore not visible to other processors. If the second version number read (line A11) is speculatively performed before the memory copy, an inconsistent memory copy can be executed even though the two version numbers read are equal. Memory barriers prevent out of order reads. They can be implemented on Pentium Pro with either the `cpuid` instruction or a locked read-modify-write instruction.

The copy-modify-replace sequence (lines A2 to A17) alone prevents tampering of data in a critical section, but does not provide mutual exclusion: cooperative processes can still modify different copies of the shared data at the same time. To solve this problem, the CMR algorithm protects the copy-modify-replace sequence with a voluntary mutual exclusion protocol, such as acquiring and releasing a memory lock, as shown in Figure 4-1 (lines A1 and A18). The voluntary protocol is used optimistically: we assume that no malicious processes exist, and recover when this assumption is false. There are two ways to detect when this assumption is false: through validation of the shared data (line A12) and through detection of when the voluntary mutual exclusion protocol is broken.

```
typedef struct
{
   unsigned int  version;
   char  *buf;
   unsigned int  size;
} versioned_data_t;

/* global variables */
unsigned int  latest;                    /* last modified by this process */
memory_lock_t l;                         /* a spinlock */

/* private variables */
unsigned int  self;                      /* ID of current process */

/* protected variables */
versioned_data_t data[self];             /* to each process: data[self] is writable, */
                                         /* for each i != self, data[i] is readable */

A1    memory_lock_acquire(&l,t)                                  /* acquire lock, wait for t ticks */
A2    if (latest != self)                                        /* did I last modify this data? */
A3       tmp_v = data[latest].version;                           /* copy version number */
A4       if (tmp_v % 2)                                          /* someone is modifying the data, lock corrupted */
A5          bail();
A6       mem_barrier();                                          /* see Section 4.1 */
A7       memcpy (data[latest].buf,data[self].buf,data[latest].size);  /* memcpy(from, to, len) */
A8       mem_barrier();                                          /* see Section 4.1 */
A9       if (data[latest].version != tmp_v)                      /* version changed, lock corrupted */
A10         bail();
A11      if (!trusted(latest, self))                             /* trust this process? */
A12         if (!validate(data[self].buf))                       /* is data corrupted? */
A13            bail();
A14   atomic_inc(data[self].version);                            /* atomically increment version number */
A15   ...                                                        /* update data[self].buf */
A16   atomic_inc(data[self].version);                            /* atomically increment version number */
A17   latest = self;                                             /* let everyone know I have the lastest version */
A18   memory_lock_release(&l);                                   /* release lock */
```

Figure 4-1: CMR: a synchronization technique among untrusted processes

```
typedef struct
{
   unsigned int  lock;
} memory_lock_t;

unsigned int  memory_lock_acquire(memory_lock_t *m, unsigned long  t)
{                                                   /* acquire, wait for t amount of time */
   unsigned long  i = current_time();
   while (current_time() < i+t)
      if (test_and_set(&(m→lock)))                  /* see Figure 2-1 for specification */
         return  1;
   return  0;
}

unsigned int  memory_lock_release(memory_lock_t *m) { m→lock = 0; }
```

Figure 4-2: A memory lock implementation with limited waiting in its acquire operation

Because the memory lock lives in unprotected shared memory, a malicious process can prevent it from working correctly by either hoarding the lock or prematurely release the lock when another process has it. The CMR synchronization technique guards against the second form of attack by attaching a protected counter (i.e. only one process can increment it, but everyone can read it) with each copy of the shared data. All counters are initially zero. Each time a process wishes to update its own copy of the shared data, it increments the counter before and after the update (lines A14 and A15, respectively). Therefore, if the counter is odd (lines A3 to A5), or if the counter after memory copy does not equal to the counter before memory copy (lines A9 and A10), the lock is corrupted. The `bail` procedure is called to recover from such corruptions (lines A5 and A10).

Preventing a malicious process from hoarding the memory lock is much more difficult. A lease-based lock, where each process can only hold the lock for a limited amount of time, solves this problem. Lease-based locks, however, require a trusted entity to enforce "leasing", therefore not practical in our system.

We chose to address this problem reactively, rather than proactively. This decision stems from the realization that if a malicious process hoards the lock, that process can also corrupt the shared data and cause the `bail` procedure to be called. Therefore, trying to protect against hoarding is not necessary. If a process waits on the lock for too long, a warning can be issued, or `bail` can be invoked. Figure 4-2 gives a simple memory lock implementation that waits for a given amount of time in its acquire operation.

Synchronization using blocking memory locks can exhibit performance drawbacks due to possible preemption [31]. In Figure 4-3, we present a variant of the CMR algorithm that uses non-blocking synchronization instead of memory lock. A double compare-and-swap primitive (specification given in Figure 2-2) is used to implement the optimistic non-blocking mutual exclusion algorithm. Before making a private snapshot of the data, a process P saves the global counter `counter`, and the ID of the last writer, `latest` (line B1). After modification, process P compares the values of `counter` and `latest` with the saved value to

```
typedef struct
{
   unsigned int  version;
   char  *buf;
   unsigned int  size;
} versioned_data_t;

/* global variables */
unsigned int  latest;                        /* last modified by this process */
unsigned int  counter;                       /* counter to prevent ABA problem */

/* private variables */
unsigned int  self;                          /* ID of current process */

/* protected variables */
versioned_data_t data[self];                 /* to each process: data[self] is writable, */
                                             /* for each i != self, data[i] is readable */

B0     retry:
B1     last_c = counter; last_latest = latest;              /* save counter, latest */
B2     if (last_latest != self)                             /* did I last modify this data? */
B3        tmp_v = data[last_latest].version;                /* copy version number */
B4        if (tmp_v % 2)                                    /* someone is modifying the data, lock corrupted*/
B5           goto retry;
B6        mem_barrier();                                    /* see Section 4.1 */
B7        memcpy (data[last_latest].buf,data[self].buf,data[last_latest].size);  /* memcpy(from, to, len) */
B8        mem_barrier();                                    /* see Section 4.1 */
B9        if (data[last_latest].version != tmp_v)           /* version changed, lock corrupted */
B10          goto retry;
B11       if (!trusted(last_latest, self))                  /* trust this process? */
B12          if (!validate(data[self].buf))                 /* is data corrupted? */
B13             bail();
B14    atomic_inc(data[self].version);                      /* atomically increment version number */
B15    ...                                                  /* update data[self].buf */
B16    atomic_inc(data[self].version);                      /* atomically increment version number */
B17    if !double_cmpxchg(&counter, &latest, last_c, last_latest, last_c+1, self)
B18       if (latest == self)                               /* someone corrupted the global counter */
B19          bail();
B20       goto retry;                                       /* atomically compare and swap version, latest */
```

Figure 4-3: A CMR variant that uses non-blocking synchronization instead of memory lock

detect possible contention. If no contention has occurred, P increments `counter` and updates `latest`. If there are contention, P retries the critical section. Atomicity of the compare and update operation is guaranteed by the double compare-and-swap primitive (line B17). This mutual exclusion algorithm is optimistic: it assumes there are low contention on shared resources. When this is the case, the double compare-and-swap rarely fails, hence no retries are needed.

Because detection of contention does not occur until the very end, each process can update its own copy of the data while another process is attempting a memory copy. We use counters attached to private copies to avoid this contention. Races can be detected by checking if the counter value read before memory copy is odd and if the value changed after memory copy. The number of retries can be bounded to prevent a malicious process from hoarding a resource by constantly changing its own copy or by constantly updating the `counter` or `latest` variable.

4.2 Common Cases

4.2.1 Synchronized Read of Trusted Data

Frequently, a trusted process has write privilege on a shared data, and all other processes can only read the data. Modifications to the shared data must go through the trusted process, via IPC or protected methods [12]. To eliminate the overhead of domain crossing on read operations, memory containing the shared data can be exported read-only to untrusted processes. Untrusted processes can use the algorithm depicted in Figure 4-4 to obtain a consistent value of the shared data. This is merely the "copy" part of the CMR algorithm. Since the data is trusted, no validation is needed.

This technique can also be used to synchronize an exokernel application against the kernel. Exokernel exposes to applications several kernel data structures and the spinlocks used to synchronize them. Exokernel applications may use these data structures to construct abstractions. Synchronized read of kernel data structures can be implemented using a modified version of the algorithm in Figure 4-4, using a kernel spinlock's usage count (incremented on acquire and release) as the counter. In practice, this form of synchronization between applications and kernel seldomly occur. In a multiprocessor library operating system implementation, we observed that only a handful abstractions depend on reading exposed kernel data structures, and most of these abstractions only need to read one or two independent variables. Such reads are guaranteed to be atomic by the underlying hardware.

4.2.2 Temporal Locality

Frequently, shared resources are accessed multiple times by one process in a short period of time, then another process in a short period of time. Interleaving access to shared resources tend to be separated by a long time. For example, when file descriptors are shared between parent and child processes, frequently the

```
/* global data */
char  *data;
unsigned int  data_size;
unsigned int  counter;
spinlock_t l;

/* synchronized read of global data into local buffer (tmp) */
retry:
v1 = counter;
if (v1%2) goto retry;
mem_barrier();
memcpy(data, tmp, data_size);              /* memcpy(from, to, len) */
mem_barrier();
v2 = counter;
if (v1 != v2) goto retry;
...                                        /* use tmp as a snapshot of the global data */

/* updating data - performed by privileged processes only */
spinlock_acquire(&l)
counter++;
...                                        /* modify global data */
counter++;
spinlock_release(&l)
```

Figure 4-4: An optimistic algorithm for synchronized read of shared data.

parent process opens and closes the descriptor, while the child process uses the descriptor while it is open.

When usages of a shared resource exhibit temporal locality, the value of the `latest` variable will almost always equal to `self`. Hence, the extra memory copy and the validation of the snapshot can both be avoided. Figure 4-5 presents this case. The shaded code segments are not executed.

4.2.3 Unidirectional Trust

Our experience building uniprocessor library operating systems indicates that shared memory are used most frequently to implement sharing of common resources between parent and children processes. In this case, children processes can trust the parent process.

When sharing occurs between processes with unidirectional trust, a process does not need to validate the private copy of a trusted process. This eliminates the validation step. Figure 4-6 presents this case. The shaded code segments are not executed.

4.3 Implementation

We have implemented the CMR algorithm in a simple library operating system, VOS. This section briefly discusses how sharing and the synchronization technique are implemented.

```
A1    memory_lock_acquire(&l,t)                                 /* acquire lock, wait for t ticks */
A2    if (latest != self)                                       /* did I last modify this data? */
A3        tmp_v = data[latest].version;                         /* copy version number */
A4        if (tmp_v % 2)                                        /* someone is modifying the data, lock corrupted */
A5            bail();
A6        mem_barrier();                                        /* see Section 4.1 */
A7        memcpy (data[latest].buf,data[self].buf,data[latest].size);  /* memcpy(from, to, len) */
A8        mem_barrier();                                        /* see Section 4.1 */
A9        if (data[latest].version != tmp_v)                    /* version changed, lock corrupted */
A10           bail();
A11       if (!trusted(latest, self))                           /* trust this process? */
A12           if (!validate(data[self].buf))                    /* is data corrupted? */
A13               bail();
A14   atomic_inc(data[self].version);                           /* atomically increment version number */
A15   ...                                                       /* update data[self].buf */
A16   atomic_inc(data[self].version);                           /* atomically increment version number */
A17   latest = self;                                            /* let everyone know I have the lastest version */
A18   memory_lock_release(&l);                                  /* release lock */
```

Figure 4-5: A common case for synchronization among untrusted processes: accesses to the shared resource exhibit temporal locality. Shaded lines are not executed

```
A1    memory_lock_acquire(&l,t)                                 /* acquire lock, wait for t ticks */
A2    if (latest != self)                                       /* did I last modify this data? */
A3        tmp_v = data[latest].version;                         /* copy version number */
A4        if (tmp_v % 2)                                        /* someone is modifying the data, lock corrupted */
A5            bail();
A6        mem_barrier();                                        /* see Section 4.1 */
A7        memcpy (data[latest].buf,data[self].buf,data[latest].size);  /* memcpy(from, to, len) */
A8        mem_barrier();                                        /* see Section 4.1 */
A9        if (data[latest].version != tmp_v)                    /* version changed, lock corrupted */
A10           bail();
A11       if (!trusted(latest, self))                           /* trust this process? */
A12           if (!validate(data[self].buf))                    /* is data corrupted? */
A13               bail();
A14   atomic_inc(data[self].version);                           /* atomically increment version number */
A15   ...                                                       /* update data[self].buf */
A16   atomic_inc(data[self].version);                           /* atomically increment version number */
A17   latest = self;                                            /* let everyone know I have the lastest version */
A18   memory_lock_release(&l);                                  /* release lock */
```

Figure 4-6: Another common case for synchronization among untrusted processes: sharing with unidirectional trust. Shaded lines are not executed

4.3.1 Shared Memory

Each exokernel process maintains and manages its own physical address to virtual address mappings. Consequently, shared memory must be described using physical addresses, rather than virtual addresses. Our library operating system uses *sbuf* (short for shared buffer) objects to describe shared memory. A sbuf object contains a physical page number, an offset into the physical page (0 to 4095), and size of the buffer. Hence, each sbuf describes a physical memory segment within a page (the granularity of protection). To represent a memory segment larger than a page, a list of sbuf objects is used. When a process allocates a sbuf object, it also allocates the corresponding shared memory segment.

Exokernel uses hierarchical capabilities [26] as an access control mechanism. When a process allocates a sbuf object for sharing, it must assign the physical memory page described in the sbuf object a correct set of capabilities. If the sbuf object describes a shared memory writable by all processes, the page must be write and read protected by a capability other processes have. On the other hand, if the sbuf object describes a shared memory used for read-only sharing, the page must be write protected by a capability other processes lack.

Most shared resources are no more than a few hundred bytes. Since the granularity of protection is a page, sharing resources can cause inefficient memory utilization if each shared resource is placed on a separate page. To minimize inefficiency, we coalesce shared memory segments protected by the same set of capabilities onto the same pages. Similar memory management strategy also appears in IO-Lite [28].

4.3.2 Sharing A Resource

The CMR algorithm requires the mapping between a trusted process and its private copy of the shared data to be trusted. This is to prevent a malicious process from pretending to be a trusted process. An untrusted process can publish a pointer to its private copy (we used a `where` variable) when updating the `latest` variable. Since malicious processes can corrupt the `latest` variable already, the correctness of this pointer is not important. A trusted process can pass the location of its private copy of the shared data to other processes when trust is established, through IPC call or forking.

Each shared abstraction has its own shared memory region. Along with the `latest`, `where`, and `lock` variables, the region must also leave space for a copy of the shared resource itself. This is to allow a dying process to relocate its private copy if it happens to have done the last modification.

Section 4.4 describes how the CMR algorithm is used in the implementation of concurrent and protected sharing of TCP sockets. Below, we present two simpler examples.

4.3.3 Example: Pipe

Two mutually untrusted processes can communicate with each other using a pipe. The pipe is implemented as a FIFO in a shared memory. Its memory layout is divided into two sections: the first section contains meta data describing the FIFO; the second section contains the FIFO itself.

Processes pass the sbuf descriptor for a pipe among each other via IPC. A IPC handler maps the shared memory used by the pipe onto its own address space. The CMR algorithm is used to synchronize operations on the pipe. To avoid unnecessary copying of data, only the first section of the shared memory, containing the meta data, is copied. After each modification, a process updates the `latest` and `where` variables appropriately. Most of the time, reading by one process and writing by another process are interleaved. Hence for each read and write, copy and validation are both performed.

4.3.4 Example: Process Table

VOS uses *procd*, a privileged process, to manage process table. Each running process inserts an entry into the process table via an IPC call. The entry contains shared data between this process and other processes. For example, scheduler hints for hierarchical schedulers. The trust relationship is unidirectional: a process P can trust other processes that use P's process table entry because capabilities can be used to protect the entry. On the other hand, processes using P's entry cannot trust P. The CMR algorithm is used to synchronize access to process table entries.

For example, we implement signals using *procd* and the process table. Each process table entry contains a table of signals posted to that process. When process P1 posts a signal to process P2, P1 makes an IPC call to *procd*. *procd* examines the process table entry shared with P2, possibly copies and validates the entry if P2 has the latest version. *procd* then updates the table of signals and the `latest` variable. When P2 wakes up, it copies the process table entry from *procd* (verification can be skipped because *procd* is trusted), modifies the table of signals, and updates the `latest` variable. Currently, a process table entry is less than 128 bytes, therefore the cost of memory copy and verification are minimal.

If P2 corrupts its process table entry, the verification step performed by *procd* would fail. Subsequently, *procd* can choose to terminate P2.

4.4 Efficient, Concurrent, and Protected Sharing of TCP Sockets

Previous exokernel literatures [13, 15, 23] give guidelines for constructing protected sharing of common abstractions, but does not describe any implementation of such an abstraction. Briceño [11] implemented an UNIX flavor library operating system by leaving all abstractions in shared memory, completed unprotected from attacks by malicious processes. This section describes an efficient implementation of protected sharing of TCP sockets based on shared memory. By doing so, we demonstrate that a multiprocessor implementation of protected sharing of TCP sockets among untrusted processes is viable.

4.4.1 Organization of An User-level TCP Socket Implementation

Our TCP sockets are based on an existing library implementation of TCP, used in earlier exokernel networking applications [24, 23]. The programming interface, similar to that of traditional UNIX, offers eight procedures of significance: socket, bind, listen, connect, accept, read, write, and close. Their semantics are described below.

- The **socket** procedure creates a data structure, called `tcpsocket_info`, that contains a list of `tcpsocket` objects, with one member initially. A `tcpsocket` object represents a TCP socket. A handle to the first `tcpsocket` object is returned as the socket descriptor.

- The **bind** and **connect** procedures each create a new packet filter and an associated packet ring for the given socket. Dynamic packet filters (DPF filters) [14] and packet rings are interfaces exposed by an exokernel to support user-level networking. When a packet arrives in a device driver, the kernel passes the packet through a filtering system and identifies a packet ring which the packet is copied onto. Applications can free packet ring entries as packets are processed.

  A DPF filter compares an input packet with specified values at given byte positions. The **bind** and **connect** procedures each create a DPF filter for the addresses which they are bound or connected to. For example, if **bind** is called on a socket with port 1234 and IP address 18.26.4.102 as arguments, the DPF filter created will match incoming TCP packets with 1234 as the destination port number and 18.26.4.102 as the destination IP address.

- The **listen** and **connect** procedures each inserts a DPF filter into kernel.

- The **accept** procedure blocks until a new TCP connection is received. When this happens, a new `tcpsocket` object is created and inserted into the `tcpsocket_info` object. New DPF filter and packet ring are created for this connection. The new DPF filter captures the TCP connection by matching a TCP packet's source and destination port number and IP addresses (four criteria). A handle to the new `tcpsocket` object is returned as the connected socket descriptor.

- The **read** and **write** procedures process packets on the packet ring and send outgoing packets via the kernel's network device interface.

- The **close** procedure closes the current TCP connection, if one exists. It also removes the socket's packet ring and DPF filter from the kernel, and deallocates the `tcpsocket` object. If this is the last `tcpsocket` object, the `tcpsocket_info` object is also deallocated.

4.4.2 Concurrent Sharing of TCP Sockets Among Untrusted Processes

Sharing occurs when a process passes a TCP socket descriptor to another process, or more frequently, when a child process inherits a TCP socket from its parent. Passing of socket descriptors only occurs among trusted processes. An untrusted child, however, can inherit a TCP socket from a parent in two common situations.

One, a network server can listen on a set of TCP sockets using the `select` interface. Each TCP socket uses a separately protected shared memory region to store its data. When a network connection arrives at a socket, the server forks, drops privilege, and loads up another, potentially untrusted, executable binary. The child process, now untrusted, has a copy of the socket. In this case, concurrent and protected sharing can be easily achieved using the CMR algorithm. When either the parent or the child process wants to access the socket, it can execute in a critical section. Two optimizations are done. First, since the parent process only cares about the state of the socket, only the `tcpsocket` and `tcpsocket_info` objects are copied (ignoring all TCP packets). These two data structures, together, are about one kilobytes in size. Second, since child process will be accessing the socket for the duration of the network connection, the common case described in Figure 4-5 apply. Memory copy and validations are avoided. If at anytime the network server wishes to use the socket and discovers an invalid `tcpsocket` or `tcpsocket_info` object, it may deallocate the socket, kill the child process, and create a new socket. A parallel version of the UNIX daemon `inetd` can be implemented in this fashion.

Two, a network server can listen on a single TCP socket. When a connection arrives, the server uses the `accept` procedure to create a connected socket. The server then forks and closes the accepting socket in the child process. When the child process loads an untrusted executable binary, only the accepted socket remains shared. Protected sharing can be achieved in two steps. First, the parent process read and write protects all of its shared memory, except those used for the `tcpsocket` object and the packet ring for the connected socket. This ensures that a malicious child process cannot tamper with other connections that the parent process may accept at a later time. Since the child process only has the DPF filter for the current connection, it also cannot receive packets designated for other TCP connections. Second, the parent process and child process both use the connected socket in critical sections provided by the CMR algorithm. Since the parent process and the child process only share the `tcpsocket` object, any corruption of that object, if detected by the parent, can cause the child process to be terminated. We have implemented some network servers in this fashion.

This second case of sharing differs from the first one only in that a `tcpsocket_info` object is not created on calls to the `accept` procedure. This is done to optimize for the common case that processes sharing connections through a single TCP socket are often trusted. Allowing them to share `tcpsocket` objects and packet rings of all connections open up possibility for performance optimizations.

4.5 Summary

This chapter introduces the Copy-Modify-Replace (CMR) algorithm that negotiates concurrency among untrusted processes: first, a private copy of the shared data, writable by one process but readable by all other processes, is obtained; then, the private copy of the shared data is modified; last, if no contention has occured, the private copy replaces the previous version of the data with a pointer modification. The CMR algorithm is optimistic. It assumes that malicious processes do not exist. When this assumption is proven false through correctness checks on the shared data and/or detection of a broken voluntary mutual exclusion protocol, the critical section is either terminated or retried.

This chapter also demonstrates how the CMR algorithm can be used to implement protected sharing of abstractions in unprotected shared memory. An implementation of TCP sockets is given.

Chapter 5

Experimental Results

This chapter presents performance comparison of three synchronization primitives used in SMP-Xok: wait-in-cache spinlock, queue-based spinlock, and read-write lock. This chapter also presents the performance of message passing in kernel and CMR, the synchronization algorithm for untrusted processes.

VOS, a simple multiprocessor library operating system, serves as the testbed for SMP-Xok and CMR. It implements process table, shared memory, threads, pipes, tcp and udp sockets and IPC abstractions. Because several important abstractions, such as signals, files, and terminal emulation, have not been implemented, ExOS [11], an uniprocessor library operating system, is used to bootstrap VOS: system critical binaries such as `initd` and shells are linked with ExOS. Binaries linked with VOS are executed from the shell by an user. Additionally, VOS reads and writes files by communicating with a file server implemented in ExOS.

Experiments are performed on an Intel SMP workstation with four 180 Mhz Pentium Pro processors and 128 megabytes of RAM and 128 kilobytes of cache for each processor.

5.1 Performance of SMP-Xok

5.1.1 Performance of Kernel Synchronization Algorithms

Three synchronization primitives are used in SMP-Xok: wait-in-cache spinlock, queue-based spinlock, and read-write lock. Wait-in-cache spinlock uses an atomic test-and-set primitive to acquire a lock sitting in main memory. If the lock is busy, the acquire operation spin-waits in its cache until the lock can be reacquired. Queue-based spinlock issues a ticket, who initial value is set to “false”, to each processor waiting on a lock. Waiting processors are ordered into a queue based on their ticket numbers. When the lock is released, the ticket value of the processor at the top of the queue is changed to “true”, allowing that processor to enter the critical section. Queue-based spinlock reduces bus contention because processors are not competing for one lock. Read-write lock allows multiple read operations on a shared resource to execute concurrently, but forces a write operation to execute exclusively.

Operation	**Cost μs**	**Cost cycles**
Empty procedure call	0.05	9
Acquire and release of wait-in-cache spinlock	0.29	53
Acquire and release of spinlock with tickets	0.43	78
Acquire and release of a read-write lock for reading	0.43	78
Acquire and release of a read-write lock for writing	0.33	59

Table 5.1: Latency of kernel synchronization algorithms

Table 5.1 presents performance comparison of the three synchronization algorithms. Wait-in-cache spinlock has the lowest latency. Enqueueing on the acquire operation and dequeuing and notifying the next ticket holder on the release operation contribute to the higher latency exhibited by ticket based spinlocks. Acquiring and releasing a read lock has a higher latency than acquiring and releasing a write lock because instead of zeroing out the number of writers on release, an atomic decrement instruction must be issued to decrement the number of readers.

In another experiment, four processors are contending for a single lock, each executing in a loop. In each iteration of the loop, a processor acquires a spinlock, computes for a small amount of time, release the spinlock, then computes some more. Ideally, the average duration of each iteration would be slightly less than four times longer (less than four times because the computation after releasing the lock can be carried out in parallel with other processors) than the iteration would take on an uniprocessor.

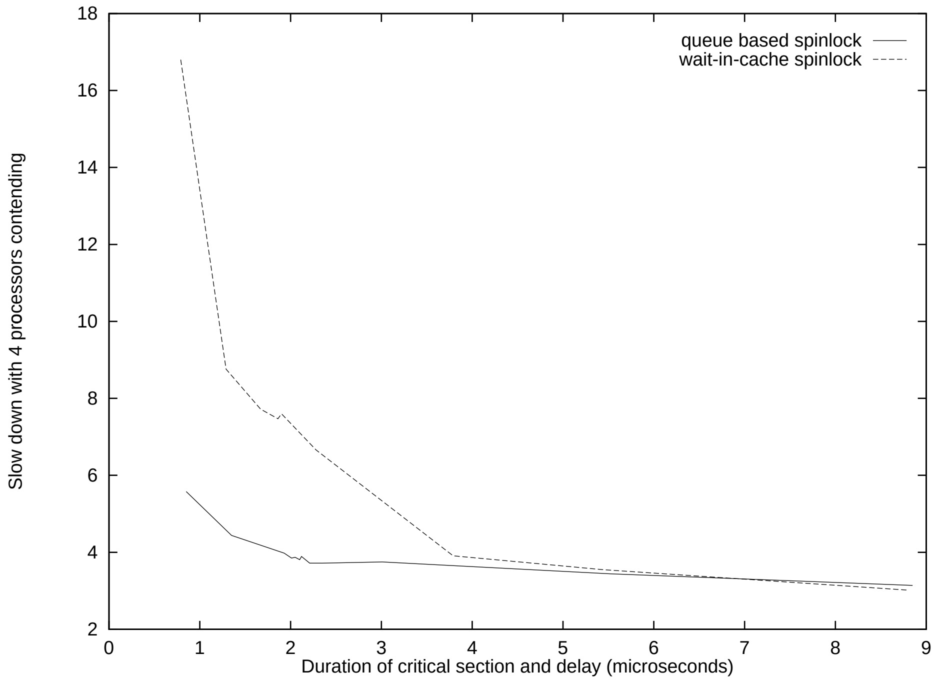

Figure 5-1: Slow down of critical section due to bus contention

Figure 5-1 presents the slow down of average iteration duration. The x-axis represents the duration

Operation	**Cost on Xok μs**	**Cost on SMP-Xok μs**
Null system call	1.5	1.5
Page fault	5.8	5.8
Protected control transfer roundtrip	6.8	7.7
Creating and destroying an environment	124	152
Cloning an environment	-	37
Mapping 128 physical pages into virtual memory	530	1350

Table 5.2: Overhead of multiprocessing on kernel operations

of each iteration of the loop under no contention, and the y-axis represents the slow down of the average iteration duration when four processors are contending. This figure shows that when the critical section is small (less than 3 microseconds), the slow down of a critical section synchronized with a wait-in-cache spinlock is dramatic (6 to 17 times). When the number of processors increases, we expect the slow down to increase for larger critical sections. The results support the use of queue-based spinlock in SMP-Xok: queue-based spinlocks are used to synchronize linked list operations and updates to system statistics, short critical operations with high probability of contention.

5.1.2 SMP-Xok Multiprocessing Overhead

Table 5.2 presents the performance of kernel operations in SMP-Xok, compared with those in Xok, an uniprocessor exokernel. The overhead of multiprocessing comes from synchronization exclusively. For example, while no spinlock is used on the protected control transfer path, a compare-and-swap instruction is performed to ensure no two processors are upcalling to the same environment. Furthermore, the rather significant overheads in creating and destroying an environment and mapping physical pages come from the decision to use fine-grained locking on each physical page's `ppage` object. Fine-grained locking results in a large number of spinlock acquire and release operations.

5.1.3 Communication Performance

Table 5.3 presents performance comparison of different implementations of IPC, using interfaces exported by SMP-Xok. In our experiment, two processes communicate with each other by making IPC requests and replies. We measure the roundtrip time of each IPC request and reply.

In the message passing experiment, a server process waits for available messages by examining the message ring downloaded into the kernel. When both the server and the client are scheduled to execute on the same processor, the latency of an IPC reply and request is 14.7 μs, which equals the speed of a full context switch plus the overhead of passing messages in kernel. On the other hand, when the two processes are scheduled to execute on different processes, much of the context switch time can be eliminated if the server can detect new IPC messages as soon as they appear on the ring. This was the case in our experiment, as the

Operation (roundtrips)	**Cost μs**
Protected control transfer, processes on the same processor	7.7
Message passing, processes on the same processor	14.7
Message passing, processes on different processors	6.4
Shared memory, processes on the same processor	9.0
Shared memory, processes on different processors	2.0

Table 5.3: Performance of different IPC implementations

server was the only process running on the second processor. Similarly, in the shared memory experiment, a server process waits for available messages by examining a pipe in shared memory. When two processes are scheduled to run on different processors concurrently, no context switches and system calls are necessary, thus the speed of an IPC roundtrip is dramatically improved. While IPC over shared memory is preferred on a multiprocessor system, message passing is needed to establish memory sharing between server and client processes because each exokernel process manages its own physical to virtual address translation.

5.2 Overhead of Synchronization in a Library Operating System

A `sendto` operation on an UDP socket in VOS creates an Ethernet packet composing the UDP datagram and passes it directly to the Ethernet card through an exokernel system call. The system call places the packet onto the device's transmit queue and returns. VOS then waits for the device to complete the transmit before returning from `sendto`. This last step allows VOS to free memory allocated for the Ethernet packet.

Table 5.4 presents the performance of CMR, the synchronization algorithm for untrusted processes (see Figure 4-1). In this experiment, an UDP socket is being shared between a parent process and a child process (Section 4.4 describes the semantics of sharing a network socket in VOS). Each process executes 10,000 `sendto` operations in a loop. Each iteration of the loop contains one `sendto` operation and some computation. To obtain the overhead of CMR, we measure the performance of each iteration when the socket is synchronized with CMR and with a voluntary synchronization algorithm. We present the average cost per iteration.

`sendto` **operation**	**Average cost μs**
socket synchronized by disabling interrupts, one process sending	96.2
socket synchronized with CMR, one process sending	96.4
socket synchronized by disabling interrupts, two processes sending on the same processor	193.1
socket synchronized with CMR, two processes sending on the same processor	195.0
socket synchronized by spinlock, two processes sending on different processors	181.7
socket synchronized with CMR, two processes sending on different processors	184.5

Table 5.4: Performance of `sendto` on sockets synchronized by CMR

The first row shows the base cost of a `sendto` operation. As a crude comparison, running the same experiment on Linux results in 30 μs per `sendto` operation. `sendto` in VOS is slower because it waits for the network device to finish sending the packet before returning, where as `sendto` in Linux places the packet onto a queue and immediately returns. Since sockets are implemented in kernel on Linux, the kernel can free memory allocated for the socket at a later time (e.g. when the device finishes transmitting).

The first two rows show that when only one process is sending packets, CMR does not introduce significant performance degradation. This is expected because when only one process is using an abstraction, the `latest` variable will always point to that process, thus no copying or validation is necessary.

The next four rows show that the performance overhead of CMR due to copying and validation is minimal when two processes are sharing and using the same socket. There are two interesting points. One, the average cost per `sendto` operation is doubled when two processes are sending packets on the same processor, but is slightly less than doubled when two processes are sending on different processors. This decrease in cost is caused by the parallalizable computation after the `sendto` operation in each loop iteration. Two, the overhead of CMR is more evident when two processes are sending packets on different processors, because copying data between processes on two different processors involve invalidating one processor's cache first. When processes are scheduled on the same processor, data to be copied can likely be found in the processor's cache.

Operation	**Cost μs**
Synchronization using spinlock	86
Actual `sendto` operation	84
Copying by CMR	2
Extra computation	12

Table 5.5: Overhead of CMR

Table 5.5 shows the cost breakdown of each iteration of the loop when two processes are sending on different processors. The cost of synchronization includes spin waiting for the other process to release the lock. This cost, appropriately, equals to the cost of each `sendto` operation and the overhead of CMR. Row three shows that the overhead of CMR is 2 μs, which is consistent with the numbers shown in Table 5.4.

5.3 Summary

This chapter measures the performance of SMP-Xok. The results demonstrate that most kernel operations in SMP-Xok are competitive with those in Xok. A few kernel operations, such as creating an environment and inserting physical pages into a virtual address space, are slowed considerably because fine-grained synchronization is used to synchronize `ppage` objects.

This chapter also measures the performance of CMR, the synchronization technique for untrusted processes. The results show that the overheads of synchronization caused by CMR are only slightly worse than those caused by voluntary synchronization algorithms.

Chapter 6

Conclusions

This thesis presents SMP-Xok, a synchronized multiprocessor exokernel capable of supporting parallel applications. SMP-Xok offers three new interfaces not found on past uniprocessor exokernels: message passing between processes on different processors using kernel maintained message buffers, kernel support for multithreading, and flexible multiprocessor scheduling.

This thesis also presents VOS, a multiprocessor library operating system. Although incomplete, it demonstrates an unprivileged library implementation of operating system abstractions is viable on a multiprocessor computer. Unprivileged and decentralized system abstractions are implemented by library operating systems in shared memory. In previous uniprocessor library operating systems, critical sections have been used extensively to provide protected sharing of these abstractions [11]. Providing critical sections in a multiprocessor library operating system, however, is much more difficult for two reasons. One, a system-wide, not processor-wide primitive must be used on a multiprocessor system. Two, voluntary synchronization protocols such as spinlock and semaphores cannot be used because applications are mutually untrusted. This thesis introduces the Copy-Modify-Replace (CMR) algorithm that negotiates concurrency among untrusted processes: first, a private copy of the shared data, writable by one process but readable by all other processes, is obtained; then, the private copy of the shared data is modified; last, if no contention has occurred, the private copy replaces the previous version of the data with a pointer modification. The CMR algorithm is optimistic. It assumes that malicious processes do not exist. When this assumption is proven false through correctness checks on the shared data and/or detection of a broken voluntary mutual exclusion protocol, the critical section is either terminated or retried.

Several open questions and tasks remain for future investigation and work:

- **Is the scheduling interface offered by SMP-Xok adequate?** We have only implemented a simple load balancing multiprocessor scheduler, but we believe that the interfaces SMP-Xok offers can be used by a hierarchical scheduler to implement more aggressive multiprocessor scheduling algorithms such as gang scheduling [27] and work stealing [10].

- **Is symmetric multiprocessing the right exokernel abstraction?** SMP-Xok is a symmetric multiprocessing system in that each processor runs the same copy of the kernel. Is this the right exokernel abstraction for multiprocessing? Would a master-slave style kernel be better for flexible and high performance multiprocessing?

- **We still need a complete multiprocessor library operating system.** VOS is merely a testbed for demonstrating that a library implementation of operating system abstractions is viable on a multiprocessor computer. It lacks many real abstractions such as buffer cache and file system.

Despite these open questions, this thesis presents a working multiprocessor exokernel system. Source code of SMP-Xok and VOS can be downloaded from http://www.pdos.lcs.mit.edu/exo/.

Bibliography

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: a new kernel foundation for UNIX development. In *Proceedings of the Summer 1986 USENIX Conference*, pages 93–112, July 1986.

[2] T. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. In *IEEE Transactions on Parallel and Distributed Systems*, volume 1(1), January 1990.

[3] T. Anderson, B. Bershad, E. Lazowska, and H. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 95–109, October 1991.

[4] T. Anderson, E. Lazowska, and H. Levy. The performance implications of thread management alternatives for shared-memory multiprocessors. *IEEE Transactions on Computers*, 38(12):1631–1644, 1989.

[5] B. Bershad. High performance cross-address space communication. Technical Report 90-06-02 (PhD Thesis), University of Washington, June 1990.

[6] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990.

[7] B. Bershad, D. Redell, and J. Ellis. Fast mutual exclusion for uniprocessors. In *Proceedings of the Fifth International Conference on Architecture Support for Programming Languages and Operating Systems*, pages 223–237, October 1992.

[8] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, December 1995.

[9] D. Black. Scheduling support for concurrency and parallelism in the mach operating system. In *IEEE Computer*, May 1990.

[10] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 356–368, November 1994.

[11] H. Briceño. Decentralizing UNIX abstractions in the exokernel architecture. Master's thesis, Massachusetts Institute of Technology, February 1997.

[12] G. Candea. Flexible and efficient sharing of protected abstractions. Master's thesis, Massachusetts Institute of Technology, May 1998.

[13] D. Engler. *The Exokernel operating system architecture*. PhD thesis, Massachusetts Institute of Technology, 1999.

[14] D. Engler and M. F. Kaashoek. DPF: fast, flexible message demultiplexing using dynamic code generation. In *ACM Communication Architectures, Protocols, and Applications (SIGCOMM) 1996*, pages 53–59, 1996.

[15] D. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 251–266, 1995.

[16] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In *Proc. of the Second Symposium on Operating Systems Design and Implementation*, October 1996.

[17] B. Ford and S. Susarla. Cpu inheritance scheduling. In *Proc. of the Second Symposium on Operating Systems Design and Implementation*, October 1996.

[18] M. Greenwalk and D. Cheriton. The synergy between non-blocking synchronization and operating system structure. *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 123–136, 1996.

[19] M. Herlihy. A methodology for implementing highly concurrent data structures. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.

[20] Intel Corporation. *Intel 82093AA I/O Advanced Programmable Interrupt Controller (IOAPIC) Specification*. May 1996. Order Number 290566-001.

[21] Intel Corporation. *Intel Pentium Pro Developer's Manual. Volume 1: Specifications*. January 1996.

[22] Intel Corporation. *Intel Pentium Pro Developer's Manual. Volume 3: Operating System Writer's Guide*. January 1999.

[23] M. F. Kaashoek, D. Engler, G. Ganger, H. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 52–65, 1997.

[24] M. F. Kaashoek, D. Engler, G. Ganger, and D. Wallach. Server operating systems. In *Proceedings of the 1996 SIGOPS European Workshop*, pages 141–148, September 1996.

[25] H. Massalin and C. Pu. A lock-free multiprocessor os kernel. Technical Report 005-91, Columbia University Computer Science Department, 1991.

[26] D. Mazieres and M. F. Kaashoek. Secure applications need flexible operating systems. In *Proc. of the Sixth Workshop on Hot Topics in Operating Systems*, 1996.

[27] J. Ousterhout. Scheduling techniques for concurrent systems. In *Proc. of 3rd International Conference on Distributed Computing Systems*, 1982.

[28] V. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A unified i/o buffering and caching system. *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, February 1999.

[29] T. Pinckney III. Operating system extensibility through event capture. Master's thesis, Massachusetts Institute of Technology, February 1997.

[30] M. Seltzer, Y. Endo, C. Small, and K. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proc. of the Second Symposium on Operating Systems Design and Implementation*, October 1996.

[31] R. Wisniewski, L. Kontothanassis, and M. Scott. High performance synchronization algorithms for multiprogrammed multiprocessors. *Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming*, 1995.