Practical Structures for Parallel Operating Systems

Jan Edler

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
Department of Computer Science
New York University
May 1995

Approved: Allan Gottlieb

Dedication

For my wife, Janet Simmons,
who never gave up hope
and for our children, Cory, Riley, and Cally,
whose interests I've tried to keep first.

Acknowledgments

I have had three employers during the course of this work: Bell Labs for the first 14 months, NEC Research Institute for the last 10 months, and New York University for more than 11 years in between. The bulk of this work was supported by the following grants and contracts: Department of Energy contract DE-AC02-76ER03077 and grant DE-FG02-88ER25052, International Business Machines joint study agreement N00039-84-R-0605(Q), and National Science Foundation grants DCR-8413359, DCR-8320085, and MIP-9303014. Views and opinions expressed in this dissertation are solely my own.

The work described here has been done as part of a large ongoing project, spanning more than a decade, and this dissertation couldn't be what it is without the involvement of many people. The following project participants have had particularly notable influence on this work: Allan Gottlieb and Mal Kalos lead the project together from before my involvement in 1982 until 1989; Allan has lead it alone since then. In addition, Allan has been my advisor and guide. Jim Lipkis was a very close collaborator until leaving NYU in 1989. Edith Schonberg, Jim, and I worked as a team on the early design and implementation of Symunix-2. Richard Kenner's contribution was widespread, but his talent for penetrating complex hardware and software problems was especially valuable. David Wood and Eric Freudenthal ported Symunix-1 to the Ultra-3 prototype, and David also did lots of low-level work for Symunix-2. Eric Freudenthal contributed at many levels, but his algorithm design work deserves special attention. Other notable contributors at NYU included Wayne Berke, Albert Cahana, Susan Dickey, Isaac Dimitrovsky, Lori Grob, and Pat Teller. In addition to being valued collaborators on a professional level, all of these people have been good friends.

Another significant force affecting this work came from IBM's RP3 Project [162]; especially influential were Greg Pfister, Alan Norton, and Ray Bryant, with whom we had many valuable debates about highly parallel operating systems.

The enthusiastic support of many prototype users was a crucial factor in attaining acceptable levels of performance, reliability, and robustness; notable among these were Anne Greenbaum from NYU and Gordon Lyons from the National Bureau of Standards. Mal Kalos and Jim Demmel led undergraduate courses using the Ultracomputer prototype to teach parallel programming.

Perhaps underlying everything else technically was the availability of UNIX source code. Countless programmers at Bell Labs, University of California at Berkeley, and many other places deserve acknowledgment for that.

My wife, Janet Simmons, has always provided the most crucial support and encouragement. My parents, Barbara and Karl Edler, helped with encouragement and proofreading.

The Courant Institute library provided a pleasant atmosphere for literature research and countless hours of writing.

This dissertation was prepared using a variety of UNIX systems[1] and a Toshiba T1000SE laptop computer; the *vi* text editor [123, 153, 129, 152] was used in all environments. Some of the figures were produced using the interactive drawing program *xfig* [181]. Formatting was performed with *troff*. (I used both AT&T's Documenter's Workbench [13] and the *groff* package [46], although *groff* still lacks an implementation of *grap* [23]. The primary macro package was −*me* [3].) Other vital software used includes *ghostscript* [54], and *ghostview* [195]. Cross referencing and indexing was done with packages of my own design, *txr* and *kwit*.

[1] IBM RT PC, IBM RS/6000, SGI Indy, and NEC 486-based PC running Linux [202].

Abstract

Large shared memory MIMD computers, with hundreds or thousands of processors, pose special problems for general purpose operating system design. In particular:

- Serial bottlenecks that are insignificant for smaller machines can seriously limit scalability.
- The needs of parallel programming environments vary greatly, requiring a flexible model for run-time support.
- Frequent synchronization within parallel applications can lead to high overhead and bad scheduling decisions.

Because of these difficulties, the algorithms, data structures, and abstractions of conventional operating systems are not well suited to highly parallel machines.

We describe the Symunix operating system for the NYU Ultracomputer, a machine with hardware support for Fetch&Φ operations and combining of memory references. Avoidance of serial bottlenecks, through careful design of interfaces and use of highly parallel algorithms and data structures, is the primary goal of the system. Support for flexible parallel programming models relies on user-mode code to implement common abstractions such as threads and shared address spaces. Close interaction between the kernel and user-mode run-time layer reduces the cost of synchronization and avoids many scheduling anomalies.

Symunix first became operational in 1985 and has been extensively used for research and instruction on two generations of Ultracomputer prototypes at NYU. We present data gathered from actual multiprocessor execution to support our claim of practicality for future large systems.

Table of Contents

Chapter 1: Introduction

The promise of highly parallel computing, to provide vast improvements in performance and cost/performance, has been repeated many times over the last few decades, but realization lags behind. Some of the delay is due to technical difficulties. A central issue is *scalability*, the degree to which a given design can be extended to larger numbers of processors without disastrous consequences in cost or performance. A *serial bottleneck* is an instance of serialization that materially limits scalability. Serial bottlenecks are the Achilles heel of highly parallel computing and may be caused by hardware, software, or both in interaction.

Not all serialization causes bottlenecks, however. For example, a serial code section involving only a sub-linear number of processors may not limit scalability. Furthermore, systems in the real world do not approach asymptotic limits, allowing solutions that exhibit reasonable cost/performance over a realistic range of sizes to be considered "scalable". This observation reduces scalable system design to a problem of "mere engineering" (no mean feat, to be sure). From a research point of view, we would like to stretch the limits of this engineering technology: we will not ignore a bottleneck simply because it is too small or rare to matter on current or anticipated real machines.

Our thesis is that careful design can produce operating systems which are free of serial bottlenecks and support powerful programming models for general purpose computing. Our approach is to implement, use, and evaluate prototype systems. Ultimately, this involves nearly every layer of whole system design, from hardware to architecture, OS kernels, run-time support software, programming languages, compilers, and applications, but the focus of this dissertation is on OS kernels and run-time support software. At the risk of being regarded too narrowly, the presentation is based on our existing and planned operating systems for the NYU Ultracomputer prototypes, but our methods are more general than that.

1.1. Organization of the Dissertation

The remainder of this chapter is organized as follows: We begin in section 1.2 with a narrative history of the project, followed by goals in section 1.3 and a soapbox-style presentation of design principles in section 1.4. In section 1.5 we give a fairly cohesive overview of our current design, and finally we wrap up the chapter in section 1.6 with a guide to the style and notational conventions used throughout this dissertation.

The rest of the dissertation begins in chapter 2 with a survey of previous and related work. Chapter 3 deals with basic algorithms and data structures. Chapters 4, 5, and 6 describe the OS kernel while chapter 7 addresses system software issues outside the kernel. Quantitative data is presented in chapter 8, and we draw conclusions in chapter 9.

1.2. Development History

Operating system work for the NYU Ultracomputer has always been tightly coupled with the prototype hardware effort. Even the earliest hardware was largely compatible with our most ambitious operating system ideas, including direct support for (non-combining) Fetch&Add. The strategy was to decouple hardware and software so that each could develop at the fastest possible rate. Advanced software would not have to wait for more sophisticated hardware, and even the early software could do a reasonable job of exercising advanced hardware. It was always important to minimize the dangers of debugging new software and new hardware simultaneously.

The first parallel UNIX work at NYU was done by this author in 1982, by modifying UNIX for a PDP-11/23 (a uniprocessor). Provision was made for sharing memory among processes by inheritance. This was done by taking advantage of the existing "shared text" mechanism in the kernel. The normal purpose of the shared text mechanism is to allow several processes running the same program to share their read-only instructions ("text"), effectively reducing the amount of memory needed for commonly used programs. The choice of whether to use shared text or not depends on the *magic number* of a program being processed by the `exec` system call. The magic number, which is stored in the first bytes of an executable file, indicates the file's format and determines the way in which `exec` will handle it. The most common magic numbers on PDP-11 UNIX at that time were octal 407 (non-shared writable text), 410 (shared read-only text), and 411 (shared read-only text, separate instruction and data address spaces). Figure 1 (A), on the next page, depicts the traditional virtual memory layout of a UNIX process.

Our modified system supported a new magic number (octal 406), which caused a shared text segment to be set up without the usual write protection and in such a way that another process `exec`ing the same file would not share it too. Thus, any variables put into the "text" segment would be shared by all processes descended from such a process by `fork` but not `exec`. Of course, putting read/write variables into the text segment isn't a normal thing to do in UNIX,[2] but it was easy to modify the assembly language output of the C compiler to accomplish this. Although inelegant, the only serious problems with this approach in practice are the lack of dynamic allocation for shared variables and the lack of a compact representation for uninitialized shared variables in object and executable files, resulting in some very large files (e.g., typical FORTRAN programs with big arrays).

Our first attempt to address the problem of dynamic allocation for shared variables foreshadowed the current operating system trend of support for shared address spaces: we added a system call to append the entire "data" segment to the end of the "text" segment, so everything allocated up to that point (except the stack) would be shared; this is shown in Figure 1 (B).[3] Following this new call, the process had a zero-length data segment. This call could only succeed when there was no real sharing, i.e., before `fork`ing, but it afforded the opportunity to change the data segment size first. This approach wasn't too successful,

[2] Having data in the text segment, whether read-only or not, is incompatible with use of the separated instruction/data address spaces available on larger PDP-11 computers.

[3] This might not seem like an obvious approach, but it was easy to implement by making use of the swapping facility.

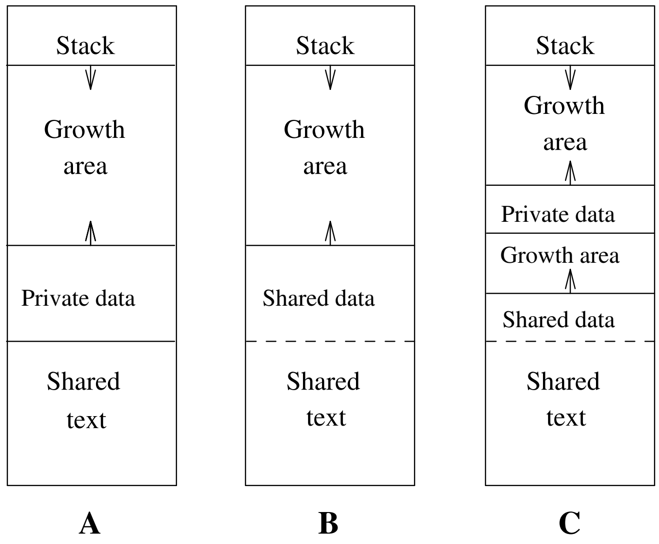


Figure 1: Text, Data, and Stack layouts in Virtual Memory.

The traditional uniprocessor UNIX layout is shown in (**A**). We added the ability to append the data to the text, thus making it shared as well; this is shown in (**B**). Note that in (**B**) there is a zero-sized private data section between the shared data and the growth area. In (**C**), we separated the text and private data, allowing room for text (i.e., shared data) to grow; this represents the situation in Symunix-1.

mostly because of the C library's tendency to statically allocate global variables that must be per-process, such as `errno` and the standard I/O structures. Similar problems have been encountered by others working on systems with fully shared address spaces (Jones [121]). Our approach was to find ways to introduce more flexible sharing mechanisms, which required having a more flexible hardware MMU than found on a PDP-11.

The next step was to try working on a real multiprocessor. Once the microprocessor to use had been chosen (the Motorola 68000), we obtained commercial uniprocessor hardware and a modified version of 7th Edition UNIX that would run on it. (The software came from Lucasfilm Ltd., the hardware was a SUN processor board from Pacific Microsystems, Inc.[4]) We[5] put similar modifications into this system as in the PDP-11 UNIX system. We soon abandoned the commercial hardware, however, and undertook the major effort to modify the MMU-sensitive parts of the kernel to match the Motorola 68451, which we had already chosen to be the MMU in our first hardware prototype (§8.1/p327), and add master/slave multiprocessor support (Hamel [100]). This operating system, later dubbed *MSunix* for

[4] This was back when SUN meant *Stanford University Network* and Sun Microsystems, Inc. was merely an interesting startup company.

[5] The team doing this work consisted of this author, Allan Gottlieb, Jim Lipkis, and Jung Hamel.

Master/Slave UNIX, was the first to support parallel programs on NYU-developed multiprocessor hardware (in 1983).

At some point after moving to a 68000-based machine, with its larger address space, our method of sharing memory between processes was refined to allow extension of the now-misnamed "shared text" segment instead of merely appending the "data" segment to it; this is shown in Figure 1 (C). This approach, which has survived to the present day in the Symunix-1 `xbrk` system call, proved quite usable, except that dynamic allocation of shared variables is still only supported at the beginning of a parallel program, before any real sharing begins. Addressing that limitation was deemed too serious to attempt without adopting a much more general memory management framework. Our current solution is the subject of chapter 5.

The next step, eliminating the major serial bottlenecks associated with a master/slave kernel organization, was done during 1983–84, primarily by this author. Many aspects of traditional UNIX kernel organization were heavily re-worked to provide synchronization for safe multiprocessor operation while avoiding serial bottlenecks. The approach taken included:

- Use of Fetch&Add-based algorithms for semaphores and readers/writers locks (Gottlieb, *et al.* [94]), along with new algorithms for readers/readers synchronization (Edler [70]) and events (Edler [71]). Since explicit low-level locking operations were not used in traditional UNIX kernels, their use involved significant changes in some cases. The work was similar in many ways to that described by Bach and Buroff [14], although they lacked our emphasis on avoiding serial bottlenecks. Our most recent developments in synchronization primitives are described in sections 3.5 and 4.5.
- Incorporation of parallel algorithms and data structures for ordered and unordered lists (Rudolph [174] and Wilson [204]). Section 3.7 covers our most recent view of this material.
- Use of a parallel buddy system memory allocation algorithm (Wilson [203]). Section 5.3.4 on page 244 covers this material as well as subsequent improvements.
- Development of a parallel search and update algorithm to eliminate serial bottlenecks in the file system. Section 3.7.6 describes this algorithm together with more recent enhancements.

The goal was to eliminate all serial bottlenecks applicable to a system with hundreds of processors, except for physical I/O. This goal was achieved with only a few exceptions, notably in the areas of buffer management and group signals (these are addressed in our more recent work and are described in section 6.1.2 and section 4.8.2). The system, dubbed Symunix because of its symmetric use of processors, more-or-less replaced MSunix in 1985. (The name was modified to Symunix-1 when we began to discuss a next generation system in 1986.) Bits and pieces of Symunix-1 are described throughout chapters 3 through 7. Additional differences between the versions of Symunix are described when we present performance data for Symunix-1 in chapter 8.

Ultra-1 hardware gave way to Ultra-2 (see section 8.1), and the combination of Symunix-1 and Ultra-2 finally provided a truly stable environment. By 1986, the system was in use by outside programmers, and by fall 1987 it supported a class of undergraduates in a parallel programming class. Additional classes used it over the next two years. Remote logins were available via the Internet,[6] and mean time between failures was measured in weeks. One machine was kept running in "production" mode for interactive users, and another was reserved for pre-arranged use by individuals, such as users needing a whole machine for timing studies, and OS developers who sometimes needed to reboot or single-step the machine. Additional machines (with fewer processors) were available for hardware development and debugging. During this period, many researchers, both at NYU and elsewhere, worked on this platform. Significant real use by users outside the development group continued into the first part of 1990, and one machine is still running and occasionally used as of this writing.

During 1990, David Wood and Eric Freudenthal ported Symunix-1 to an early version of the Ultra-3 prototype then under construction (Bianchini, *et al.* [29]). Regular use of Symunix-1/Ultra-3 continues to this day.

Design work on a new operating system, to be based on 4.3BSD UNIX rather than 7th Edition UNIX, began in 1986 as part of a joint study agreement between IBM and NYU, with the IBM RP3 (Pfister *et al.* [162]) and Ultra-2 as primary targets. This second generation operating system became known as Symunix-2. A preliminary version of it ran on Ultra-2 hardware and an RP3 simulator in 1987, but that version was not developed to the degree necessary for robust operation. Subsequent work has led to the still-incomplete version of the system described in chapters 3 through 7.

1.3. Development Goals

It should not be surprising that development goals can evolve over time almost as much as the design itself; as earlier goals are approached, they are replaced by more ambitious ones.

In the beginning, the basic goals were to exercise the prototype hardware and the simplest Fetch&Add-based algorithms. Since then, we broadened our field of hardware interest somewhat, defining our *target class* of machines to be those supporting the following:

- MIMD execution[7]
- Shared memory with mostly uniform access times[8]
- Lots of processors, 2^8 or more
- Lots of memory, 2^{24} bytes or more per processor
- Hardware implementation of Fetch&Add
- Hardware combining of memory references
- Per-processor page-oriented address translation hardware

There are no existing machines that possess all of these properties, although the NYU Ultracomputer prototypes all support MIMD execution, shared memory, and hardware Fetch&Add, and the latest, Ultra-3, only falls short on number of processors. Among

[6] The Ultracomputers were not on the network directly, but it was possible (and easy) to log in remotely by use of some magic involving a DECnet connection to the PDP-11 responsible for I/O to the parallel machine (see Figure 16, in section 8.1 on page 328).

[7] MIMD stands for the class of machines with *M*ultiple *I*nstruction streams and *M*ultiple *D*ata streams. This terminology was introduced by Flynn [82].

[8] We will discuss uniformity of access times further in section 2.1.

commercial machines, bus-based systems, such as those made by Sequent [80, 139], Encore, or Silicon Graphics, are also close, primarily lacking in the number of processors they can support, but they also vary in the strength of their hardware support for Fetch&Add. Some non-bus-based commercial machines are also close to our target class, most notably the BBN Butterfly series [18], which lack full support for Fetch&Add, combining, and uniform memory access times.[9] A more recent commercial design close to our target class is the Tera Computer (Alverson *et al.* [4]), which has a variable interleaving scheme reminiscent of the IBM RP3, full/empty bits on each memory word as in the Denelcor HEP (Smith [182]), non-combining Fetch&Add, and many other, more unusual, features.

Limited portability to machines outside the target class is a secondary goal, possibly requiring machine-specific replacements for certain code modules. In the last few years, the following general goals have emerged:

(1) *Scalability.* The cost of serial bottlenecks rises linearly with the size of the machine, so avoiding them is of paramount importance.

(2) *High overall performance.* Systems without good performance are relatively uninteresting to users.

(3) *Portability.* There is significant room for variation even within the target class of machines, and many machines outside the target class can also benefit from our approach.

(4) *Modularity.* This is not only good software engineering practice, but also aids portability and encourages experimentation with design alternatives.

It is also useful to identify *non-goals*, things that are neither pursued aggressively (in marked contrast to current trends in research and industry) nor opposed. Some of our non-goals are object-oriented programming, networking, distributed file systems, message passing, remote procedure call, windows, fault tolerance, real-time processing, and automatic parallelization. These (and others) are fine goals, but we must limit the scope of our effort. We have also not tried to fight in the "UNIX wars" (it is not consistent with our role as researchers) or to promulgate standards (formal standardization in this area is still premature).

1.4. Design Principles

The most important design principles have been adoption of the general UNIX framework and self-service implementation structure. Self-service is in contrast to the use of server processes. While servers can be multi-threaded to avoid the bottlenecks they would otherwise impose, such an approach loses one of the original and fundamental reasons for adopting servers in the first place, namely the implicit serialization, which solves many problems of concurrency. (There are also other important and valid reasons for using servers, some of which we accept, but our general approach is self-service.) Additional design principles we have tried to follow include:

[9] The last machine in this series, the BBN TC-2000, was built to support memory interleaving as an option, thus achieving reasonably uniform access times, but the machine was discontinued without ever being widely used.

- Use highly-parallel data structures and algorithms wherever useful for the target class of machines.
- Optimize for low resource contention in most cases (e.g., two-stage synchronization functions, §3.5/p50 and §4.5/p167).
- Avoid costly protection domain changes in common operations (e.g., temporary non-preemption, §4.6.4).
- Allow (limited) duplication of function between kernel and user-mode libraries. This lowers overhead in common cases, both by avoiding protection domain changes, and by specialization (e.g. user-mode thread management, §7.5).
- Avoid interference with applications that require relatively little operating system support (e.g., permit a non-preemptive scheduling policy, §4.6.3).
- Sacrifice ease of portability for significant performance gains if the beneficiary is a machine in the target class (e.g., widespread use of centralized data structures rather than distributed ones).
- Use the least powerful mechanism available to solve a problem (e.g., the proliferation of list and synchronization types, with different performance/functionality tradeoffs, §3.5/p49, §3.7.1/p82, and §4.5/p166).
- Make performance-critical decisions at compile time whenever possible, even at the cost of duplicating and near-duplicating code. Macros and inline functions are valuable tools for carrying out this design principle (e.g., optional instrumentation support, §3.5.2/p54, and two-stage synchronization with delayed check function evaluation, §3.5/p50).
- Rely on interrupts and signals to reduce the overhead of frequent operations by eliminating checks for unusual situations (e.g., `softresched`, §4.6.1).
- Use lazy evaluation whenever there is significant hope that an operation will turn out to be unnecessary (e.g., lazy and anonymous file creation, §5.2.7/p221).
- Design new kernel features to be as widely usable as possible, while still meeting their original needs (e.g., scheduling groups, §4.6.3, families, §4.2/p148, asynchronous system calls, §4.3.1, and ksems, §7.3.4).

1.5. Design Overview

This section gives a condensed but cohesive overview of Symunix-2, which is presented with much greater detail in chapters 3 through 7.

Key Abstractions

Broadly speaking, one can divide the features and functions of any operating system into three general areas: processing, memory, and input/output. For a less trivial overview of Symunix-2, we can identify several top-level *abstractions* provided by the kernel and describe some of their properties and interactions:

- *Process*. A process is an abstraction of the *processor*, and may be thought of as a *virtual processor*. As such, it inherits the unprivileged computational model of the underlying real processor (control flow, instruction set, registers, data formats, etc.). This unprivileged computational model is augmented with a set of *system calls* (analogous to the various privileged operations in a real processor) and *signals* (analogous to interrupts and traps), that go far beyond what any real processor provides, including even the ability to create and control new processes. Just as the real processors have MMUs to provide *virtual address spaces* by performing address translation, our abstract processes also have abstract MMUs that provide each process with a private address space. Just as a real processor can enlist the aid of other real processors, coprocessors, peripheral processors, or device controllers, a process can invoke *asynchronous system calls* and *page faults* to perform some operations concurrently.

- *Memory*. The memory abstraction derives most of its key properties from the underlying hardware. This includes word size, atomicity, ordering of memory accesses, and any non-transparent cache memory; the programmer or compiler must be aware of these issues. Accessibility of memory is determined by the abstract MMU of each process; arbitrary patterns of sharing and protection may be established by setting up the appropriate mappings.
- *Descriptors*. The closest hardware analogue of a descriptor is a *capability* (Fabry [78]). A descriptor allows access to an abstract object, usually a file or a communication stream. Descriptors are unforgeable; they exist in a per-process name space. With kernel assistance, a new descriptor may be created with the same underlying reference and access rights as another; such copying may be performed within a process or between processes. Descriptors provide a measure of independence for processes, since the standard operations `read` and `write` can be used on most descriptors without detailed knowledge of the underlying object.
- *File System*. The file system is an abstraction of secondary, non-volatile storage devices. We have adopted the traditional UNIX file system model with no crucial modifications.

Overall Structure

The overall structure of Symunix is the same as that of traditional UNIX: a kernel (divided at least somewhat into machine-independent and machine-dependent parts), shells, libraries, and applications. Everything outside the kernel is said to operate in *user-mode*; everything inside the kernel operates in *kernel-mode*, including access to all privileged hardware operations. The kernel itself can be roughly divided into a *top half*, consisting of all those parts executed directly by processes in response to their system calls and exceptions, and a *bottom half*, consisting of all those parts executed in response to asynchronous interrupts.

Processes and Activities

Unlike traditional UNIX systems, the active (schedulable) element in the Symunix-2 kernel is not the process, but a new abstraction called the *activity*. While superficially similar to *kernel threads* provided by other systems, Symunix activities are not explicitly visible to the user-level programmer. Each process has, at any point in time, exactly one activity (known as the *primary* activity) that can execute in user-mode. Other activities are created and destroyed automatically to execute asynchronous system calls and page faults. It is possible for more than one activity of a process to execute concurrently on different real processors, but the expected number of activities per process is low, and isn't closely related to the overall parallelism of the application.

Control Flow

As in traditional UNIX systems, a system call or exception, such as a page fault or division by zero, is executed in the context of the process incurring it, i.e., with the normal set of resources associated with that process: memory, descriptors, identifiers, etc. Interrupt handlers ignore the current process context and never perform activity context-switches (except for the rescheduling interrupt, introduced in Symunix-2 and described on the next page). Except for initialization code and possibly some system processes with no user-mode component, the only way to execute code in the kernel is by synchronous events (such as system calls and exceptions from user-mode) or interrupts.

Scheduling

Context-switching, from activity to activity, happens in one of two ways:

(1) An activity engages in *context-switching synchronization* and cannot proceed; it then becomes *blocked* and another activity is selected to run.

(2) The lowest priority of all interrupts is a software-generated rescheduling interrupt. When a rescheduling interrupt occurs, a search is made for another activity eligible to preempt the interrupted one, and, if found, a context-switch is performed. This kind of preemption can happen whenever the rescheduling interrupt is unmasked, whether the activity was executing in user- or kernel-mode.

In contrast, traditional UNIX kernels permit preemption only at certain points (generally right before making the transition from kernel-mode to user-mode, although Lennert [135] described a more aggressive approach, with multiple preemption points).

The kernel provides several services to allow user control over scheduling, particularly in support of fine-grain parallel processing:

- *Scheduling Groups*. Each process belongs to a scheduling group, and a scheduling group has a *scheduling policy*. Policies include the traditional UNIX time-sharing policy, in which each process is treated independently; a completely non-preemptive policy, which guarantees an approved process the exclusive use of a processor; and a "gang scheduled" policy, which attempts to run all or none of the group members concurrently. Other policies could be incorporated as well.
- *Temporary non-preemption*. Any process may request a fixed amount of time to be guaranteed to run without preemption. The mechanism is cheap (typically 3 memory references) and cannot be abused for any long-term advantage. Temporary non-preemption is typically used to avoid performance anomalies when a process might otherwise be preempted while holding a busy-waiting lock for a short critical section.
- *Preemption warning*. A signal, `SIGPREEMPT`, can be requested to give a process advance notice and a short grace period before preemption. This is typically used to save enough state so that the preempted computation can be resumed by another cooperating process.

User Address Space

The address space of a process consists of all virtual addresses expressible by the underlying processor, typically 2^{32} or more different addresses. Within this space, some contiguous address ranges are *mapped* to contiguous ranges within files; other addresses are invalid and generate exceptions when used. Files that are the target of such mappings are called *image files*, and also serve as backing storage for the corresponding virtual memory. All user-accessible memory appears in such ordinary files; there are no anonymous "swap partitions" or "swap files". This file-mapping approach provides naming and protection mechanisms for shared memory, by virtue of the file system structure.

Each mapping carries access permissions and attributes that determine what happens when a child process is created or a process overlays itself with another program (the UNIX `exec` system call). Address spaces and their mappings are private to each process, but proper use of signals and a new file descriptor transfer mechanism allow efficient emulation of shared or partially shared address spaces, as required by the user's programming environment.

Parallel Program Support

The facilities we have just described were designed to support parallel programs. Such programs are executed initially by an ordinary UNIX `exec` system call, and thus begin life as a serial process. A process can create children with the `spawn` system call, a generalized version of the traditional `fork` that can create more than one child at a time.[10] All the processes related by `spawn`, but not by `exec`, are called a *family*. The family as a whole can be the recipient of certain signals.

The kernel allows great flexibility on the way a program uses its processes or memory. For example, there are no predetermined stack areas and no automatic stack growth. This flexibility is intended to support a wide variety of user-level run-time organizations, such as *user-mode thread systems*.

Parallel Implementation

Throughout the system, as indicated in section 1.3, the highest priority has been given to achieving scalability by eliminating serial bottlenecks. Some of the structural and interface changes made to further that goal have already been described. Such changes allow the use of highly parallel algorithms, often based upon combinable Fetch&Φ operations. Some new bottleneck-free algorithms introduced in this dissertation include:

- A restricted set algorithm, supporting insertion and arbitrary deletion of integers in a compact range (§3.7.5).
- A structure, called a *broadcast tree*, which provides atomic semantics for certain operations on groups of objects, particularly useful for sending a signal to a whole group of processes (§3.7.4). (cf. multicast in ISIS: Birman *et al.* [31, 30])
- A structure, called a *visit list*, which provides a serialization-free way to perform an arbitrary operation on a group of objects (§3.7.3).
- A mechanism, called *random broadcast*, that invokes an interrupt handler on a specified number of arbitrarily chosen processors (§3.6).
- A dictionary algorithm for search, insertion, and deletion of objects with reference counts (§3.7.6).
- An algorithm to keep a list of objects in approximate LRU[11] order (§3.7.2).
- A buddy system memory allocator that is more fragmentation-resistant than previous algorithms (§5.3.4/p247).
- A simple approach for combining busy-waiting and context-switching synchronization into new *hybrid* synchronization algorithms (§7.3.3).
- A context-switching algorithm for group lock, with a new variant of fuzzy barrier (§7.3.5/p304).

---

[10] A UNIX-compatible `fork` call is provided that calls `spawn`.

[11] Least Recently Used.

1.6. Style and Notation

Program excerpts and certain terms, such as the names of variables, constants, and functions, are presented in a distinct typeface, as in

```
old = faa (&var, 1);
```

to emphasize that they're not written in English. Individual identifiers are scattered as needed in the text, and longer program excerpts are written as blocks of code. Excerpts using this typeface differ slightly from "real" code that is or could be used: even those that aren't contrived are often incomplete, and minor transformations have been made as needed to improve their readability in the context of this document. Examples of such changes are:

- Elimination or addition of white space, line breaks, `register` declarations, and some comments.
- Elimination of some assertions. Assertions are executable statements that perform run-time-checking to verify conditions assumed true by the programmer. They have no semantic effect, and are normally disabled at compile time, but can be a valuable debugging tool. In real "production" code, they aid readability, as "living comments", but offer little advantage in the context of a document such as this.
- Elimination of conditionally-compiled code that doesn't contribute to the exposition in this document. Special code for debugging, performance measurement, or special hardware configurations, are examples of such eliminated code. (However, some of this code is described in section 8.2.)
- Elimination of some optimizations that don't affect correctness.
- Replacement of complex or irrelevant code with *words in italic type*.
- Elimination of `volatile` keywords. While most of the algorithms presented here are well suited to use of a relaxed memory consistency model (see Mosberger's survey [154]), we generally assume sequential consistency for simplicity of exposition.
- Breaking complex expressions or statements into several parts, possibly introducing new variables in the process.
- Use of comments in the style of C++ (Stroustrup [187]), i.e.,

  ```
  // to the end of the line.
  ```

  Of course this convention breaks down inside multi-line macros, in which all but the last line end with `\`. We simply use ordinary C `/*` comments `*/` in those cases.
- Allowing variable declarations anywhere a statement may appear, as in C++.
- Elimination of some type conversions (casts).
- As a result of some of these transformations, some variables are eliminated and code sequences contracted.

Coding Conventions

Code fragments in C follow fairly ordinary coding conventions, but there are a few which deserve some explanation:

Leading underscores (_)
: We generally use leading underscores in function or macro names that are "internal" to a module, such as in the implementation of an abstract data type.

Macros instead of simple `#if` / `#else` / `#endif`
: C supports conditional compilation, in the form of "preprocessor" directives, such as

```
#if BW_INSTRUMENT
    stuff only when instrumenting busy-waiting
#endif
```

These surely get confusing when large or nested, but even add a significant clutter factor when only a single line falls between the two directives. In some cases, we eliminate these directives by defining a macro, such as

```
#if BW_INSTRUMENT
#define bw_instrument(x)    x
#else
#define bw_instrument(x)    /* nothing */
#endif
```

which evaluates to its argument, or to nothing, depending on whether or not the feature is enabled. This macro then shows up in usage such as

```
        bw_instrument(stuff only when instrumenting busy-waiting);
```

This is less disruptive to code readability. Similar constructs are used for other forms of instrumentation.

`do { ... } while(0)`
This construct, which might appear baffling, is valuable when constructing macros with complex expansions. Like many other languages, C has flow-of-control statements such as

```
if (cond)          if (cond)          while (cond)
     s;                 s1;           s;
                   else
                        s2;
```

The expressions `s`, `s1`, and `s2` are statements. Compound statements enclosed in `{ ... }` pairs are available and semicolon (`;`) is a statement terminator, rather than a separator. It is not unusual to write a macro with an expansion of more than one statement. One can write

```
#define FOO(x)     s1;s2
```

but a programmer calling such a macro must know to call it as

```
if (cond)
     { FOO(x); }
```

when dealing with such flow control statements. The braces could be put inside the macro, but then

```
if (cond)
     FOO(x);
else
     bar();
```

fails with a syntax error because of the null statement before the `else`, unless the calling programmer knows to leave out the semicolon. We use the subterfuge of enclosing complex macros in `do { ... } while(0)` constructs because it works correctly when followed by a semicolon, and creates a syntax error (generating a compiler error message) if the programmer omits the semicolon, i.e., it is callable like an ordinary function. (Of course it still has macro, rather than function, semantics.)

Pseudo-Code

Where minor transformations are insufficient to simplify an algorithm for presentation purposes, we resort to blocks of *pseudo-code*, which use many C-like operators and keywords, but are set in the standard Roman typeface with *italic* variables. To emphasize the more abstract nature of this code, indentation is significant (rather than relying on **begin**...**end** or {...} to delineate blocks), assignment is indicated by $\leftarrow$ rather than by = or :=, and semicolons terminating statements are generally omitted in such pseudo-code blocks.

Chapter 2: Survey of Previous Work

Over the years, a great number of computer systems utilizing more than one processor have been designed (and many of them have been built), representing a wide spectrum of approaches (see, for example, Enslow [77] and Schwartz [175]).

In line with the target class of machines we identified in section 1.3, we restrict our focus in several ways:

- *MIMD*. We pass over the whole range of SIMD machines.
- *Shared memory*. We ignore designs with only private memory, although this distinction is not always very clear. In particular, some designs call for a software implementation of shared memory (see, for example, Li [137], Carter *et al.* [43], bershad *et al.* [27]). While many variations on the software approach have validity, further consideration here would take this dissertation too far afield.
- *General-purpose*. This is meant to exclude systems whose only claim to parallelism rests on architectural features such as dedicated input/output processors, or machines considered incapable of supporting a “complete” operating system. The latter would disqualify parallel processors designed to depend on a conventional host computer for most operating system functions. A “complete” operating system should perform such functions as process and memory management, input/output and file system management, protection, resource allocation, and communications.

As might be expected, the number and variety of operating systems for the resulting class of machines are great.

After a review of basic terminology for shared memory organizations in section 2.1, Section 2.2 provides an overall historical survey to put the more detailed descriptions in section 2.3 into perspective.

2.1. Terminology

Two common ways of organizing a large shared memory multiprocessor are shown in Figure 2, on the next page. Assuming the interconnection network paths are the same for all processor−memory combinations, (A) has been called a *UMA* (Uniform Memory Access) architecture (Rashid [167]). The corresponding term for (B) is *NUMA* (Non-Uniform Memory Access).[12] These terms only capture a single aspect of the multiprocessor design space,

---

[12] Figure 2 (B) is NUMA regardless of the uniformity of network path lengths, while (A) is NUMA only if the network path lengths vary. Machines without shared memory are called *NORMA* (No Remote Memory Access) under this taxonomy. More recently, the term *COMA* (Cache Only Memory Access) has been applied to machines such as the Data Diffusion Machine (Hagersten *et al.* [99]) and

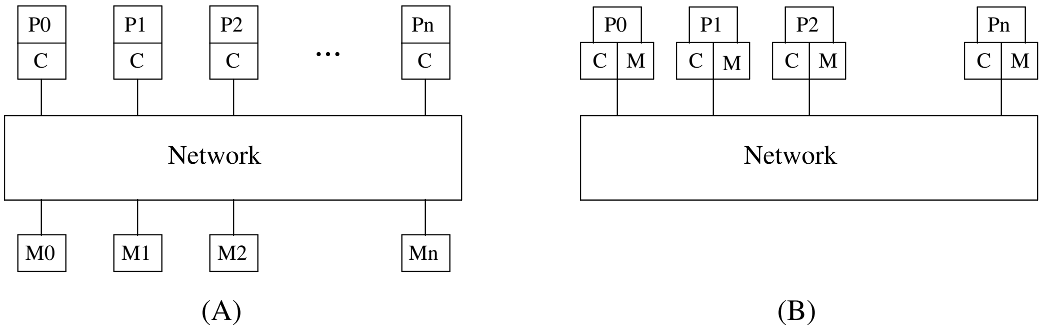


Figure 2: Shared Memory Organizations.

In (A), all memory references that aren't cache hits must traverse the interconnection network. In (B), references to the locally connected memory module are satisfied without using the network. (A) has been called a *dance hall* model, evoking an image of a room with boys lined up along one wall and girls along the other.
P=processor, C=cache, M=memory

however they are generally applied much more broadly. For example, it is generally assumed that NUMA machines employ uninterleaved memory addressing across memory modules; otherwise it would be more difficult to take advantage of local memory. Indeed, the terminology in this case seems to have more to do with how the machine is programmed than with the distribution of access times.

Recently it has become popular to refer to Figure 2 (A) as a "shared memory" design, and (B) as a "distributed memory" (or "distributed *shared* memory") design, despite the fact that both employ physically distributed but shared memory. This unfortunate usage has essentially the same meaning as the UMA/NUMA distinction.

2.2. Historical Development

Early systems were most often the result of adding processors to existing mainframe designs, but even very early, some machines were carefully designed with symmetric multiprocessing in mind. The first "true" multiprocessor (with up to four identical processors, shared memory, and a fairly modern operating system) appears to be the Burroughs D825, introduced in 1960 (Enslow [77]). Other early machines included the Burroughs B5000/B6000 series (the B5000 was available with two processors in 1963; see Organick [159]), the GE-645 (four processors in 1965), the UNIVAC 1108 (three processors in 1965), and the IBM System/360 model 67 (two processors in 1966; see MacKinnon [141]). The goals of these machines were generally increased throughput, reliability, and flexibility, and, although facilities for parallel programming were often available on such systems, any such use was secondary.

An early trend was to dedicate processors to specific functions. This could take several forms, from special-purpose to general-purpose processors, usually dedicated to input/output

---

the KSR-1 (Rothnie [171]), a class that isn't properly represented by either (A) or (B).

or other resource management functions. For example, while IBM channels (Blaauw and Brooks [33], Stevens [186]) are clearly not general-purpose processors, the peripheral processors (PPs) of the CDC 6600 (Thornton [197]) were powerful and general enough to run the bulk of the operating system for that machine. Note, however, that the PPs were still smaller and less powerful than the primary computational engine, and ordinary user programs were not run on them. The BCC-500 (Lampson [133], Jones and Schwarz [120]) exemplified another approach in which several processors of roughly similar power and generality were dedicated (both by hardware and system software) to specific functions. Of the five processors, two were dedicated to running user's programs and one each to managing the memory hierarchy, terminal I/O, and processor scheduling.

As the industry began to deal with the practical problems of parallel programming and operating systems for parallel machines, it was also developing a general theory and approach to operating systems. Concepts such as semaphores (Dijkstra [60, 59]), monitors (Brinch Hansen [40], Hoare [110]), and message passing (Hoare [111], Brinch Hansen [39]), were developed and treated formally.

In the 1970s, minicomputer technology made experimentation with larger multiprocessors more feasible. Two of the more influential research systems were designed and built at Carnegie-Mellon University: C.mmp, described further in section 2.3.3, and Cm*, described further in section 2.3.4. Bolt, Beranek, and Newman built the PLURIBUS system (Katsuki, *et al.* [126]), and used it as an Arpanet Interface Message Processor (IMP). The Naval Postgraduate School experimented with interconnected PDP-11 computers running an early version of the UNIX operating system, described further in section 2.3.5. Most of the current generation of multiprocessor systems owe more of their heritage to these kinds of experimental systems, based on minicomputer technology in the 1970s, than to the multiprocessor mainframe and supercomputers that preceeded or followed. The primary reason for this has to do with the rise of UNIX and what has now become the chief slogan of modern computing: "open systems".

2.2.1. The Role of UNIX in Multiprocessor Research

Since the early 1980s, a lot of operating systems research and development for uniprocessor and network environments has taken place within the framework of the UNIX operating system. Because of this general popularity in the OS research community and the ease of adapting UNIX to run in small multiprocessor configurations, the use of a UNIX framework for multiprocessor OS research became more and more common. Such early parallel UNIX efforts were generally based on a simple "master/slave" organization, but the limitations of such an approach became quickly apparent. Today, because UNIX is the basis for the dominant "open systems" movement in large parts of the computer industry,[13] a high degree of UNIX compatibility is considered mandatory for any general purpose multiprocessor to be a commercial success.

---

[13] Of course the commercial UNIX of today is a far cry from the UNIX of yesterday. Consider, for example, the size of the manuals describing the basic programming interface: the seventh edition UNIX manual [21] consists of two easily managed volumes, while the latest specification of what constitutes a "UNIX" system [212] consists of five volumes, with more than 2700 pages.

Bell Laboratories did some of the earliest UNIX multiprocessor work in a somewhat indirect fashion by layering the UNIX kernel on top of the vendor-supplied operating system of a UNIVAC 1100 series (Bodenstab, *et al.* [37]) or IBM System/370 mainframe (Felton, *et al.* [79]), and relying on the “native” operating system to handle low-level operations such as I/O, process scheduling, and memory management, while a stripped-down version of the UNIX kernel was used to emulate the traditional UNIX environment. Later, Bell Labs researchers developed a complete “symmetric” UNIX system described in section 2.3.7. Some other significant early UNIX implementations for multiprocessors supported machines by MASSCOMP (Finger, *et al.* [81]), Sequent (described further in section 2.3.8), Denelcor (Hoag [109]), Cray (Reinhardt [168]), Alliant, Convex, Encore, FLEX (Matalan [145]), and Sequoia (Mark [142]), all with various levels of support for parallel applications. Nowadays, some flavor of UNIX is standard on essentially every major commercially available multiprocessor.

Some manufacturers, such as Hewlett-Packard (Clegg [47]), ELXSI (Olsen [157]), Apollo, Microsoft, and IBM have produced new proprietary operating systems, and layered UNIX compatibility on top (sometimes as an afterthought, sometimes with third-party help). In the guise of POSIX [113] and many other standards, UNIX has become completely ubiquitous.

2.2.2. The Rise of the Microkernel

Full-function general purpose operating systems are always structured in layers, but the best division into layers seems always to be a subject for debate. Of particular importance is the following key question:

What goes into the kernel, the innermost part, of the operating system?

The flip side of the question, “*what gets left out*”, cannot be answered without also answering the question, “*where does it go?*” The issue is a matter of aesthetics as much as science or engineering.

The traditional UNIX design consists of two main layers: the kernel and everything else. One of the attractive properties of the system throughout much of its history was its small size, especially in relation to its functionality. This didn’t stop people from complaining that the system had become bloated, however, and serious hardware limitations (especially the 64 Kilobyte PDP-11 address space) helped to keep the kernel somewhat trim. By the mid 1980s, with rapid hardware advancement (especially the widespread acceptance of the VAX, with its larger address space) and accompanying growth of software functionality (e.g., modern networking), there was widespread recognition of a kernel size problem.

Systems with more layers have been around much longer than UNIX (indeed, MULTICS (Organick [158]), with many concentric rings, was an important (negative) influence on UNIX, and the inspiration for its name). Even within the UNIX sphere of influence, the MERT system had all the basic qualities of a layered microkernel-based system in the 1970s (Lycklama and Bayer [140]).

Continuing into the 1980s, research and other advanced operating systems continued to address the kernel size issue, usually by moving key functionality out into “server” processes. In most cases, this was driven more by the development of distributed computing than shared memory multiprocessors. Prime examples include the V system (Cheriton [44]),, Mach (Accetta *et al.* [1]), Amoeba (Mullender *et al.* [155]), and QNX (Hildebrand [108]).

The major barrier to "moving things out of the kernel" is performance: lots of things can run faster in the kernel, by avoiding the overhead of protection barrier crossings, directly accessing internal data structures, enjoying favorable scheduling, etc. This remains true today; the most enduring movement of functionality from the kernel seems to reflect the needs of network distribution or other functional requirements rather than a sense of design purity. The most common elements removed from the kernel today are network file systems and graphical display servers. Some recent systems, most notably Chorus (Rozier *et al.* [172]) and Lipto (Druschel *et al.* [69]), have had some success with the ability to move modules in or out of the kernel's protection domain to achieve different configuration and performance tradeoffs.

Small size isn't the only characteristic of a microkernel; exporting a set of simple yet powerful facilities is also of key importance. Of course there are many different ways to do this, but two common features stand out:

(1) *Dis-integration of the process.* The traditional UNIX process includes many attributes, including address space, flow of control, file descriptors, signal handlers, `umask` value, etc. Most recently-designed systems have taken the approach of decoupling at least some of these attributes. Some provide a flexible system, where several different resource types may be selected at thread creation time for sharing or copying (see, e.g., Barton and Wagner [17], Aral *et al.* [9], and Pike *et al.* [164]). Most attention has fallen on the separation of address space from control flow; the result is called *kernel supported threads*, *kernel-mode threads*, or *kernel implemented threads*.

(2) *Enhanced communication between protection domains.* In principle, a microkernel might not provide any communication facility other than shared memory, but all microkernels to date have incorporated a carefully designed high performance messaging mechanism. This is a natural approach because distribution across loosely coupled machines is normally one of the goals of microkernel efforts. The facility must be carefully designed and optimized because it is heavily used (Bershad, *et al.* [26]).

Both of these are old ideas: even IBM's OS/VS2 system (MacKinnon [141]), later known as MVS, provided kernel support for multiple threads of control in a shared address space, and Brinch Hansen's RC 4000 system [39] was based on a small kernel providing fast IPC in the 1960s.

The Mach project at CMU (Baron *et al.* [16], Accetta *et al.* [1], Tevanian *et al.* [193], Young *et al.* [214]; also see section 2.3.9) took a hybrid approach to multiprocessor OS evolution, in that many of the features of the Accent network operating system (Rashid and Robertson [166]) were integrated into a traditional UNIX kernel (4.3 BSD). The result, together with some new features to support parallel applications, has been used to support many different parallel or distributed OS research projects and several commercial offerings. Subsequent versions of Mach pushed most of the UNIX heritage outside of the kernel, and greatly promoted the microkernel concept. The first effort, described by Golub *et al.* [89], put most UNIX functionality into a single big multi-threaded server, the so-called *single server* system. This work and the subsequent OSF/1 AD system (Zajcew *et al.* [216]), were driven more to meet the needs of distributed (non shared memory) systems than the kinds of machines in our target class. More recent work has emphasized the need for careful implementation to overcome the negative performance impact of implementing key OS services outside the kernel. Julin *et al.* [124] describe the Mach *multi server* system, wherein a set of separate servers supply generalized system services and an emulation library linked into each application program provides a compatible UNIX interface. Ford and Lepreau [83] modified an OSF/1 single server Mach system to implement remote procedure call more efficiently and dramatically improving performance. Another testimonial to the importance and difficulty of getting good performance from a Mach system with a UNIX emulator was given by Wallach [183].

2.2.3. Threads

Programming with multiple threads of control within a single address space has become the most popular way of writing parallel applications for MIMD shared memory systems. A number of *threads packages* have been written, mostly for explicit use by programmers writing parallel applications in C or C++; they give the programmer some measure of portability as well as taking care of some of the details of thread management. These include Cthreads (Cooper and Draves [68]), Brown threads (Doeppner [67]), PCR threads (Weiser *et al.* [201]), Ultrix threads (Conde *et al.* [48]), Firefly/Modula-2 threads (Birrell *et al.* [32]), FastThreads (Anderson *et al.* [8]), QuickThreads (Keppel [127]), and others. In addition, the POSIX set of standards includes a threads interface, known generally as Pthreads [114], which most parallel system vendors can be expected to support.

There are two extreme approaches to implementing threads: in the kernel or outside the kernel. The former is potentially more robust and better integrated with the overall system, and the latter has potentially better performance. As a result, some effort has gone into the quest for the best of both worlds, generally by implementing threads in user-mode with some extra assistance from the kernel. Edler *et al.* [76, 75] argued for this approach in justifying the lack of kernel threads in Symunix-2; Anderson *et al.* [7] and Marsh *et al.* [144] also described systems along similar lines (see section 2.3.11 and section 2.3.12, respectively).

2.2.4. Synchronization

Synchronization between two or more processes can be achieved in two different ways:

- *Busy-waiting*, also called *spinning*, in which a process continues to consume processor cycles whenever it must wait.
- *Context-switching*, also called *blocking*, in which a waiting process relinquishes its processor while waiting.

It's also possible for a system to select between these two alternatives dynamically, as described by Ousterhout [160], who called it "two-phase blocking;" we discuss this hybrid approach further in section 7.3.3.

Busy-waiting synchronization can be achieved with no special hardware support beyond atomic *load* and *store* operations on shared memory (Dijkstra [58], Lamport [131]), but most practical work is based on the use of additional atomic *read-modify-write* operations. The most common such operations are probably *Test&Set* and *Compare&Swap* (Arnold *et al.* [10], MacKinnon [141]).

An alternative to read-modify-write operations for systems with hardware cache coherence is the load-linked/store-conditional mechanism of Jensen *et al.* [118], which is supported by several commercial microprocessors.[14] A load-linked operation is semantically equivalent to an ordinary load, but a store-conditional instruction has no effect unless the cache coherence protocol is certain that the target memory location is unchanged since the matching load-locked instruction was executed by the same processor. The combined effect of load-linked and store-conditional is similar to ordinary load and Compare&Swap, but is more powerful since the store-conditional succeeds only if the target location is *unmodified*, whereas Compare&Swap succeeds if the location merely has the *same value*.

Herlihy showed how to construct "non-blocking" and "wait-free" data structures using Compare&Swap [104] and load-linked/store-conditional [106]. Such data structures have desirable properties that become apparent when a participating processor is arbitrarily delayed or halted:

- With a non-blocking data structure, some other processor is guaranteed to make progress after a finite number of steps.
- With a wait-free data structure, *all* other processors are guaranteed to make progress after a finite number of steps.

These are clearly desirable properties, but they do not imply a performance improvement as the number of processors increases (in particular, updates are generally serialized).

Busy-Waiting

By their very nature, busy-waiting synchronization algorithms will sometimes require a processor to wait (by repeatedly reading a memory location) for it to change. Much work has focused on efficient busy-waiting synchronization for machines with hardware cache coherence or locally-accessible shared memory. The goal is to reduce serialization caused by contention for shared memory.

Rudolph and Segall [173] proposed the use of a "test and test and set" loop for locking on systems with hardware for cache coherence, the goal being to have processors do most spinning by referencing only their local cache. Unfortunately, this doesn't avoid the necessity of referencing shared memory when the lock is finally released, which can require time linear or quadratic in the number of waiting processors (depending on cache coherence protocol details).

A variety of *backoff* techniques have been applied to reduce memory contention while locking, and many of these were examined quantitatively by Anderson [6] and Graunke and Thakkar [95]. In a backoff scheme, explicit delays are dynamically calculated to respond to the level of contention experienced. There are many variations on this approach, with generally reasonable performance. Compared to a simple busy-waiting lock, backoff-based locking typically sacrifices fairness, since, in most such schemes, newly arriving processors poll at a higher frequency than processors that have already been waiting a long time.

The QOSB primitive proposed by Goodman *et al.* [90] builds a queue of waiting processors using the memory of the cache lines located at each processor. A variant of Test&Set fails when the issuing processor is not at the head of the queue; as a result, spinning is almost completely local.

---

[14] For example, the MIPS R4000 series [151] the DEC Alpha [57], and IBM PowerPC [117].

Anderson [6] gave a software-only lock algorithm with similar behavior on machines with a hardware cache coherency mechanism. Each waiting processor selects a unique spinning location within an array by using Fetch&Increment (below). Each spinning location is marked to indicate either "wait" or "proceed"; all but the first entry is initially marked to wait. Releasing the lock is simply a matter of marking the owner's array entry to wait and the circularly next array entry to proceed. By arranging the array elements to fall in different cache lines, spin-waiting memory traffic is almost completely eliminated.

Graunke and Thakkar [95] documented the effect of a variety of locking algorithms on the Sequent Symmetry and describe an algorithm similar to Anderson's, in which a separate array element is permanently associated with each participating processor. An atomic exchange (Fetch&Store) is used to determine the location associated with the previous lock holder; this is the location for spinning. Releasing the lock is a matter of changing the state of one's own permanent location.

The busy-waiting algorithms of Mellor-Crummey and Scott [147, 149] use a linked list to represent the spinning locations, so each processor spins on a location of its own choosing. This allows spinning to be local even when the machine has no cache coherency mechanism in hardware.

A problem with all of these local-spinning synchronization methods is that they impose some queuing order on processors as they begin to synchronize. While this has the advantage of ensuring fairness, a desirable property, it also makes each waiting processor suffer from every delay experienced by each predecessor in the queue. Such delays might include shared memory contention, interrupt, exception, and page fault handling, and preemptive scheduling in multiprogramming environments. Wisniewski *et al.* [206] proposed lock algorithms that accommodate preemptive scheduling by skipping over a preempted waiter (and thus partially sacrifice fairness).

An alternate method of busy-waiting, which avoids congesting the processor-memory interconnection bus or network with polling traffic, is to stall the processor until the synchronization condition is satisfied. The full/empty bits on memory words of the Denelcor HEP (Smith [182]) are an example of this kind of synchronization, as is Harrison's Add-and-Lambda proposal [101], based on a modification to the basic NYU Ultracomputer design.

Fetch&Φ

The class of Fetch&Φ operations was introduced by Gottlieb and Kruskal [92] as an improvement and generalization of the Replace-Add operation, upon which the NYU Ultracomputer design had been based up to that point (Gottlieb *et al.* [94, 93], Rudolph [174]). Each Fetch&Φ operation refers to a specific binary operator, Φ, which must be associative. The action of Fetch&$\Phi(v,e)$ is to atomically replace the variable v with the value $\Phi(v,e)$ and return the old value of v. The most used and studied Fetch&Φ function is Fetch&Add, for which $\Phi(v,e)$ is $v+e$. The older operation, Replace-Add, differs only in that it returns the new (updated) value of v. Other notable Fetch&Φ operations include Fetch&Store, also called swap, for which $\Phi(v,e)$ is e, and Fetch&Or, which is similar to Test&Set (using a bitwise-or for Φ). As a special case, a simple load operation can be accomplished with operations such as Fetch&Add(v,0) or Fetch&Or(v,0); a simple store can be accomplished by discarding the result of Fetch&Store. Other notable special cases include Fetch&Increment (the same as Fetch&Add(v,1)) and Fetch&Decrement (the same as Fetch&Add(v,–1)), which are more easily implemented on some machines.

One of the most interesting things about Fetch&Φ operations is the way in which they can be *combined* by hardware, as in the NYU Ultracomputer (Gottlieb *et al.* [91]). In this way, it is possible for many such operations to take place in the time normally required for just one, even when they are directed at the same memory location. Although this desirable property requires hardware not available on most computers, a growing number of machines support some non-combining variant of Fetch&Add, including the Intel 486 and Pentium microprocessors [115] and the Cray Research T3D multiprocessor.

2.2.5. Bottleneck-Free Algorithms

In addition to the possibility of combining, Fetch&Phi operations are valuable because they have been used to construct bottleneck-free solutions for many standard algorithm design problems. An algorithm is *bottleneck-free* if it has no heavily contended serial critical sections when run on a suitable idealized parallel computer. A critical section is heavily contended if the expected waiting time to enter it grows with the number of participating processors. The idealized parallel computer in this case is Schwartz's Paracomputer [176], augmented with the necessary Fetch&Φ instructions.[15]

Critical sections are a concept normally associated with software, but hardware can have critical sections too. In particular, read-modify-write memory cycles constitute small critical sections, as do operations on shared buses. The technique of combining Fetch&Φ operations in hardware can avoid serialization caused by such hardware critical sections, and, when used together with a bottleneck-free algorithm, can eliminate hardware bottlenecks in some cases.

Quite a few bottleneck-free Fetch&Add-based synchronization algorithms have been devised. Gottlieb *et al.* [94, 93] presented algorithms for semaphores and readers/writers locks, the latter having the property that no critical section is used in the absence of writers. Bottleneck-free queues were also presented, utilizing sequential memory allocation and a variety of methods for synchronizing access to the individual cells of the queues. Ordinary (FIFO) queues, priority queues, and multi-queues were all presented. Rudolph [174] extended these results to include a fair semaphore, non-sequential queue allocation, a stack (LIFO), and a barrier.[16]

Wilson [205] provided interior access and removal operations for bottleneck-free queues, using the same basic internal structure as Rudolph's non-sequential queues. He also gave several variations, such as an unordered list, and outlined a number of abstract OS operations, such as task creation, destruction, scheduling and coordination, and gave several memory allocation algorithms.

Dimitrovsky [65] introduced parallel hash tables, new queue algorithms (using rather different data structures from those of Rudolph and Wilson), and the group lock [66], which, among other things, allows easy construction of bottleneck-free algorithms from some serial algorithms.

---

[15] The Paracomputer is essentially equivalent to Borodin and Hopcroft's WRAM [38], and Snir's CRCW version [184] of Fortune and Wyllie's PRAM [84].

[16] Barriers were introduced by Jordan [122].

The restrictive special cases of Fetch&Add(v,±1), called Fetch&Increment and Fetch&Decrement, are of special practical interest, since they can be implemented more cheaply than general Fetch&Add on some machines. Freudenthal and Gottlieb [86] gave several algorithms based on these operations, including a readers/writers lock, a group lock, and a multiqueue.

Fetch&Φ-based algorithms, although attractive tools for constructing bottleneck-free systems, are not quite the only game in town. The organization of some software systems can be altered to adjust the tradeoff between frequency of coordination (low is good for performance) and "load balance" (better balance is good, but can require more frequent coordination). For example, Anderson *et al.* [8] studied the tradeoffs involved in choosing the arrangement of queues used by a thread scheduler, with some shared and some private to each processor. Another approach is to use algorithms that naturally avoid *hot spot* contention (Pfister and Norton [163]) by distributing computations over multiple memory locations and combining them in a logarithmic fashion in software with low-contention synchronization, e.g., pairwise. Such *software combining algorithms* have been proposed and studied by Yew *et al.* [213], Goodman *et al.* [90], Gupta and Hill [98], and Mellor-Crummey and Scott [148, 147, 149].

2.2.6. Scheduling

Scheduling in parallel systems is more challenging than for uniprocessors. Some important additional issues are:

- *Interdependencies* between processes.[17] Closely cooperating processes should be scheduled together as much as possible. This is an issue for general-purpose systems, where unrelated parallel jobs are multiprogrammed on a single parallel machine.
- *Load balancing*. This issue doesn't arise at all for uniprocessors. For multiprocessors, it is important to avoid having some processors idle while others are busy and also have a backlog of work to do.
- *Management of common resources*, particularly memory. In general, processes require other resources than just processors, and ignoring those requirements when making scheduling decisions can lead to some processors being unable to make progress for lack of those other resources. For example, two processes cannot run together, or possibly will thrash and make poor progress, if their combined need for physical memory exceeds the total memory size of the machine. This problem is more severe on multiprocessors than on uniprocessors.
- *Memory locality*. On NUMA machines, the access time to memory will not be the same for all processor/memory module pairs. Optimally scheduling the placement of many cooperating processes and the memory they will use isn't always practical, especially when access patterns aren't fully known in advance. (Memory locality issues are less severe for the target class of machines we consider in this dissertation.)
- *Cache affinity*. Most recent designs include a relatively large cache memory at each processor. A running process can build up a substantial investment in such a cache, which, if forfeited unnecessarily at context-switch time, can hurt performance.

---

[17] We use the term *process* here, although terminology for schedulable entities varies from one system to another. Thread, task, activity, activation, etc., may be substituted freely in this context.

The effect of multiprogramming (sharing a single machine among unrelated jobs) on synchronization and scheduling has been extensively studied. Ousterhout [160] presented the *co-scheduling* policy, whereby the scheduler attempts to run related processes concurrently, and the *two-phase blocking* synchronization technique for combining the advantages of busy-waiting and context-switching. Zahorjan, *et al.* [215], used analytical methods and simulation to study the serious detrimental effect of frequent "spinning" synchronization in a multiprogrammed environment, along with the benefits of several synchronization-aware scheduling policies. Edler *et al.* [76] presented some features of Symunix-2 designed to allow scheduling, synchronization, and management of threads in user-mode. These include a method whereby a process can usually avoid preemption while executing in a critical section, one of the effective scheduling policies identified by Zahorjan *et al*. Anderson *et al.* [7] and Marsh *et al.* [144] also presented systems in which thread management and scheduling take place largely in user-mode. In the former, the state of a preempted thread is passed to another cooperating processor, which can run it or enqueue it for later execution. In the latter, threads are given advance warning (via an asynchronous interrupt or a pollable flag) of impending preemption; this allows them to save state and arrange for quick resumption on another cooperating processor. In all three of these systems, the actual scheduling policies, queueing arrangements, etc., are under control of the user and may be tailored to meet the needs of the application.

Leutenegger [136] concluded from simulations that co-scheduling schemes work well and that, in general, medium and long-term allocation of processors to jobs is at least as important as interprocess synchronization or choice of preemption frequency.

Markatos *et al.* [143] and Crovella *et al.* [52] performed experiments on a BBN Butterfly to evaluate co-scheduling and relatively long-term hardware partitioning of the machine, dividing the processors among unrelated jobs; they concluded that partitioning was a generally superior approach.

Black [34] studied scheduling in Mach and developed a processor allocation mechanism to allow applications to be allocated processors on a long-term basis.

Tucker and Gupta [198] implemented a central server process to perform long-term processor allocation among parallel jobs. Each job periodically examines its processor needs, polls the server to determine it's fair allocation, and then adjusts the number of underlying processors it will use for running threads during the next period. The interactions with the server are handled transparently to the application by the thread package, based on Brown threads [67]. Tucker and Gupta's work also includes a low overhead mechanism for preventing preemption, but unlike the Symunix-2 mechanism, it doesn't provide any protection against abuse or accidental misuse.

Burger *et al.* [42] examined issues related to scheduling with a demand-paged virtual memory system on certain NUMA machines; they got best results with gang scheduling and busy-waiting, and even decided that context-switching on a page fault was not worthwhile.

Squillante and Lazowska [185] studied the effects of several scheduling policies designed to improve performance by recognizing the value of a processor's cache contents to processes that recently ran there (this is called *cache affinity*). The idea is to avoid process migration as much as possible while still employing it where necessary to maintain a good load balance.

2.3. Overview of Selected Systems

A number of operating systems are especially notable and reviewed in more detail here. In several cases an important general concept, problem, or solution in parallel systems is discussed in the context of a particular operating system. This should not be taken to imply that the situation is not relevant to other systems.

2.3.1. Burroughs B5000/B6000 Series

The operating system for Burroughs[18] computers is known as the Master Control Program, or MCP. The overall design of this family of machines is heavily oriented towards the execution of programs written in an Algol-like block structured language (Organick [159], Lonergan and King [138]). Just as the resulting static and dynamic structures of nested activation records provide a clean model for data sharing within a serial program, they also provide a simple extension to a model of parallel execution. Burroughs extended its implementation of the Algol-60 language to include tasks (which are like procedures, but execute concurrently with their caller) and events. Waiting for an event that has not happened causes a task to be suspended and placed on a waiting queue associated with the event. Signaling an event causes all tasks on the waiting queue to be moved onto the system-wide ready queue.

When a task is created, its new activation record is linked into the stack of activation records in essentially the same way as for an ordinary procedure. This creates an extended stack known as a saguaro or cactus stack (Hauck and Dent [102]) because of its branching structure. The sharability of variables between tasks is defined according to the nesting structure and the hierarchical relationships among tasks, in a natural extension to the way accessibility is defined for variables in the serial execution environment.

2.3.2. IBM 370/OS/VS2

Following somewhat in the footsteps of earlier IBM multiprocessors such as the 9020A (IBM [116]) and the System/360 models 65/67, IBM introduced the System/370 models 158/168 MP (MacKinnon [141]) in 1974. These machines were available in several configurations, but basically consisted of 2 System/370 processors and memory. Each CPU had its own complement of I/O channels, and control units could be configured to be accessible from two channels, one on each CPU. The instruction set was extended to include several new instructions including *Compare and Swap*, used along with the earlier *Test and Set* for synchronization. Other extensions to the basic 370 processor model to support multiprocessing included interconnections for interprocessor signaling and error handling.

The Compare and Swap instruction compares the contents of a register with the contents of a memory word, and if equal, stores a specified value into the memory word, otherwise it loads the contents of the memory word into the register. Because this instruction operates atomically, it can be used, like Test and Set, to implement various mutual exclusion schemes. It has an advantage over Test and Set in that a whole word can be set, so useful information can be stored in the memory word (e.g. task identifier). Also, it is possible in some cases to update data structures with Compare and Swap directly, thus avoiding the need for a lock. For example, items can be added to or deleted from a linked list safely,

[18] Burroughs subsequently merged with Sperry Univac to become Unisys.

because the new value will not be stored into memory if the old value is changed unexpectedly by another processor; if so, the Compare and Swap can be re-executed.

The overall structure of the operating system, OS/VS2 release 2 (later MVS), was symmetric. Only one copy of the operating system resided in memory. Almost all of available main memory was maintained in a common pool, and could be allocated by either processor. A single set of queues was accessible to either processor. Either processor could perform I/O, as long as it had a channel connected to the desired control unit. The system could recover from the failure of a either processor.

One of the more interesting aspects of this system is the way in which the operating system was extended to provide mutual exclusion (locking) and avoid deadlock. A set of locks was defined and arranged in a hierarchy. Each lock was used to serialize operations related to a particular set of data structures. For example, locks were assigned to the dispatcher, memory allocation, and input/output functions. Some locks were very static, and others were more dynamic. For example, there was a single lock for dispatcher functions, but a separate lock is created with each instance of several different kinds of dynamically-allocated control blocks (user, channel, device, etc.). In the latter case, all locks of the same type were assigned the same position in the hierarchy. A set of rules governed when a process could request a lock:

- A process cannot obtain more than one lock at a given level in the hierarchy at a time.
- A process can obtain an additional lock provided that it is higher in the hierarchy than any lock already held.
- A process need not obtain locks at all levels up to the highest level desired, but the desired locks must be obtained in hierarchical order.

The hierarchy was arranged so that most of the time the locking order was natural, corresponding to typical function calling sequences, but in certain cases lower level locks had to be obtained in advance of actual need in order to comply with the above three rules for deadlock avoidance.

Several types of locks were supported, the major distinction being between *spin locks* and *suspend locks*, as in other operating systems. When a process attempts to obtain an already locked spin lock, the processor is not relinquished as it would be for a suspend lock (the name *spin lock* probably refers to to the notion that the processor is “spinning its wheels”, not doing anything useful while waiting for the lock). Often a processor’s interrupts must be masked prior to obtaining a spin lock and not unmasked until after the lock is released.

In order to minimize the time interrupts are masked, OS/VS2 release 2 went to great effort to provide a “window” of enabled interrupts periodically while waiting for a lock with interrupts masked. This was important because recovery from failure of one processor was a major requirement, and otherwise a processor looping with interrupts masked would never receive the special interrupt generated when the other processor failed, and recovery would be impossible. Other steps taken to enable recovery from processor failures included data structures to record which locks were held by each processor.

The basic facility for parallel applications is multitasking within a single address space. The tasks are recognized and scheduled by the operating system, and can use a variant of suspend locks for synchronization.

The system described here has evolved over the years, and is similar in many respects to the current IBM MVS operating system available on current members of the 370 family (Tucker [199]).

2.3.3. CMU C.mmp/HYDRA

The C.mmp computer (Wulf and Bell [208]) was designed and built at Carnegie-Mellon University starting in 1971. It consisted of 16 DEC PDP-11 processors, each with 8 Kilobytes of private memory and shared access to 16 memory modules, via a 16-way crossbar switch. Each processor was further augmented with a writable control store, to support the addition of special instructions for such purposes as synchronization. The basic synchronization mechanism was provided by the memory controller's implementation of atomic read and write operations on words, together with a read-modify-write memory cycle. A cache memory was also planned for each processor, to reduce contention to shared memory, but never implemented. The physical I/O connections were asymmetric, since peripherals were connected to individual processors.

The operating system for C.mmp was HYDRA (Wulf *et al.* [209, 211, 210]). Except for certain processor-specific data structures and frequently executed kernel code in local memory, all processors executed a single copy of the kernel. The most important feature of HYDRA was its adoption of the object model. Objects were variable-sized and contained two parts: a data part and a capability part. Capabilities are unforgeable pointers to objects, together with a set of access rights (Fabry [78], Lampson [133]). Possession of a capability for an object implies the right to perform operations on the object according to the associated access rights. Some object types and operations were primitive and defined by the kernel, but others could be defined by users. The primitive object types included *process* and *procedure*. The primitive operations on procedures included *CALL* and *RETURN*; *CALL*ing a procedure established a new local name space, defining the objects accessible within the procedure, similar to but more general than the creation of a new activation record in the Burroughs B5000/B6000 series MCP. Another primitive operation on objects was normal memory access to the data part. The address space of a process is defined by the set of capabilities that it has; that address space is graph-structured, since objects contain capabilities. The cost of the kernel *CALL* function prohibited the realistic use of procedures as a replacement for ordinary subroutine calls, and procedures are more properly regarded as a sort of *module*. The HYDRA object system allowed for shared memory and the definition of various synchronization facilities such as semaphores, messages, and monitors.

Four types of synchronization were commonly used:

- kernel locks
- spin locks
- kernel semaphores (K-sems)[19]
- policy semaphores (P-sems)

Kernel locks and semaphores were used in the HYDRA kernel, while spin locks and policy semaphores were used in user programs. The two kinds of semaphores provided for rescheduling of the processor, while the locks did not.

[19] Not to be confused with the Ksems of section 7.3.4.

Kernel locks used a special interprocessor interrupt facility. Each lock was a data structure that included a bit mask indicating the processors waiting for this lock. When a processor encountered a locked lock, it would set its bit in the mask, and then execute a "wait" instruction (this instruction does nothing but wait for an interrupt; no memory references are made). When a processor unlocked a kernel lock, if any bits were set in the mask, it would send an interrupt to the indicated processors. A carefully timed sequence of instructions was executed to insure that the unlocking processor did not read the bitmask before the waiting processors had time to set the appropriate bit.

The interprocessor interrupt facility was only available within the HYDRA kernel, so another facility was needed that could by used by applications. Spin locks were much like those used in IBM's OS/VS2; in contrast to the HYDRA kernel locks, memory bandwidth was consumed while a processor waited for a spin lock.

Kernel semaphores could be used within the HYDRA kernel when expected waiting times were relatively long, so that the expense of tying up a processor for such a period of time would be considered too great. Similar to the suspend locks of IBM's OS/VS2, a processor encountering a locked K-sem would place the current task on a queue associated with the semaphore and switch to another task.

Policy semaphores were originally intended to address the synchronization needs outside the kernel, but proved too expensive when expected waiting time was low, resulting in the development and use of spin locks. P-sems are like K-sems in that the processor switches to another task when a locked semaphore is encountered, but more complicated because of their interaction with specially-designated *policy modules*. If a task is suspended for more than a certain pre-determined amount of time, its memory becomes eligible for swapping to secondary storage, and a policy module is notified, to make long-term scheduling decisions. An additional time-out feature was planned but not implemented, which would cause an attempt to lock a semaphore to fail after waiting too long, rather than just waiting forever, as a way of dealing with certain kinds of deadlocks.

2.3.4. CMU Cm*/StarOS/Medusa

A second highly influential multiprocessor system from CMU was the Cm* computer (Swan *et al.* [189, 188], Jones and Schwarz [120]). The project began in 1975, the machine became operational in 1976. Whereas the processor/memory relationship was symmetrical in C.mmp, Cm* had a hierarchical relationship. Each processor (computer module, hence the name Cm*) was interfaced by a switch (called the *Slocal*) to a bus connecting all processors of a cluster and a special microcoded processor called a *Kmap*, responsible for address mapping and request routing. Each Kmap was connected to an intercluster bus, so that the hierarchy has three levels: the processor, the cluster, and the whole machine. Since Cm* was a shared memory machine, the time to access memory depended on how "far away" the memory was. The closest memory was local to a processor, the next closest memory would be connected to a different processor in the same cluster, and the most distant memory was connected to a processor in a different cluster. This difference in memory access times had a major influence on the design of software for the machine. In addition to processing normal memory requests, the Kmaps were microprogrammed to perform various special synchronization and frequently-executed operating system functions.

Two operating systems were built for Cm*: StarOS (Jones *et al.* [119]) and Medusa (Ousterhout *et al.* [161]). While StarOS concentrated on providing facilities to make the power of the machine available to the user in a relatively convenient form, Medusa concentrated on issues of structure, and rather than trying to hide or compensate for the hierarchical structure of the machine, it provided facilities that mirrored the underlying hardware. There were many similarities between them, as well as with HYDRA. Both new systems were object oriented, relied on capabilities, and supported a programming structure known as a *task force*. But since StarOS tried to hide the underlying architectural structure more than Medusa, its model of a task force was more general. A task force was a collection of schedulable *activities*,[20] known to the operating system, cooperating on a single task. An attempt was made to schedule the activities of a task force concurrently onto different processors.

Both systems were structured around multiple task forces, each implementing a particular system facility, such as memory management, I/O, or file system. In addition, application programs took the form of task forces. Message passing was the primary means of interaction between task forces, although shared access to objects as memory was provided as well.

The expense of accessing non-local memory dominated many of the design decisions of these systems. For example, the activities of Medusa were restricted to run on processors containing their code in local memory. While StarOS was less restrictive in theory, the drive to achieve adequate performance often drove it to adopt similar restrictions in practice. A further difficulty was the lack of sufficient memory in each computer module to support representative activities of all system tasks. This resulted in the need to pass control from processor to processor when executing operating system requests.

2.3.5. MUNIX

Possibly the earliest project to modify the UNIX operating system for a multiprocessor was made at the U.S. Naval postgraduate school (Hawley and Meyer [103]). The machine in question was constructed from two DEC PDP-11 computers by providing a portion of memory dual-ported to each processor's bus. In addition, each processor had a certain amount of private memory and a set of peripherals. The system was asymmetric in that each peripheral was connected to only one processor. This hardware asymmetry, together with the presence of private memory created difficulties; for example, the contents of private memory might not be accessible to the swapping disk.

The operating system, dubbed MUNIX, was fully symmetric except for I/O. Only a single copy of the kernel resided in memory, and controlled the entire machine. Semaphores were used for synchronization; the implementation was similar to the kernel locks of HYDRA. MUNIX used a fairly small number of semaphores to serialize access to important system data structures; only 7 different locks were used: two to protect the process table, and one each for scheduler data, the disk buffer pool, the character buffer pool, the free memory pool, and the swap device. Note that while the system had a symmetric structure, little attempt was made to avoid serialization of unrelated operations of the same type. In contrast, other systems (such as IBM's OS/VS2 and those described below) allocate separate locks to certain individual data structures, rather than a single lock for the entire collection of similar

[20] Not to be confused with the Symunix activities of section 4.3.

structures. The coarse-grained approach to locking taken in MUNIX was not much more complicated than a simple master/slave scheme, and allowed a certain amount of concurrent operation in the operating system kernel.

Apparently no facilities were developed to make shared memory available to the programmer for parallel applications.

2.3.6. Purdue Dual VAX/UNIX

The Purdue University school of electrical engineering designed a simple scheme for interconnecting the buses of two DEC VAX-11/780 computers, together with an equally simple modification to the UNIX operating system to support it (Goble and Marsh [88]). The operating system was a classic "master/slave" organization, so only the peripherals of one processor could be used.

A master/slave organization is the simplest structure for an operating system with a single scheduler. Only one processor, the master, is allowed to execute most operating system functions. All other processors (in the Purdue dual VAX, there is only one other) are limited to running processes in user-mode only. While this restriction can be a severe bottleneck, it may not be, especially if there are only a few processors and if the workload is not operating system intensive. The chief attraction of a master/slave multiprocessor operating system is, however, that it is relatively easy to produce by modifying a uniprocessor operating system. In the case of the UNIX operating system, few modifications are needed other than to provide mutual exclusion for the process ready queue and some mechanism to mark processes that cannot be run on a slave.

There are several similarities between the approach taken by this project and the DEC VMS operating system for the VAX-11/782. There were slight differences between the DEC and Purdue hardware, but they were minor enough that Purdue UNIX was easily modified to run on the 782 also. Actually, DEC's VMS was even more master/slave oriented than Purdue's UNIX, since the latter allowed the slave to execute special code to select a process to run, whereas VMS required the master to perform process selection for the slave. On the other hand, Purdue did not develop facilities to support parallel applications, while VMS already had such features as shared memory and synchronization mailboxes (even on uniprocessor VAX computers).

2.3.7. AT&T 3B20A/UNIX

Following the already-mentioned mainframe UNIX systems, which were layered on top of another operating system, Bell Laboratories modified a version of UNIX to run directly on the dual-processor 3B20A computer. The hardware of this machine is asymmetric, in that one of the processors cannot do I/O, but the operating system, described by Bach and Buroff [14], is much more general and applicable to configurations with symmetric I/O capabilities and more processors. Each processor is comparable in power to the DEC VAX-11/780.

Internally, the kernel was changed to use semaphores for synchronization; the basic form of semaphore is the task-switching counting semaphore, but a variation of the regular semaphore P operation, called *conditional-P*, is also provided. When a conditional-P encounters a locked semaphore, it returns immediately with an indication of failure instead of suspending the process and rescheduling the processor. By calling conditional-P in a loop, a simple spin lock can be simulated.

Deadlock is avoided through an ordering of lock types, similar in concept to the lock ordering used in OS/VS2. When a process needs to hold more than one lock at a time, it must obtain them in the proper order. In some cases, this is not possible, and unlike the situation in OS/VS2, it is often not possible to request locks in advance of need because the out-of-order lock is associated with one of many identical data structures and its exact identity is not known until other locks have been obtained. The solution adopted is to rely on the conditional-P operation: when requesting an out-of-order lock, conditional-P is used; if it fails, the process must conclude that deadlock is possible and execute recovery code. The recovery code typically backs out any partially completed work, unlocks semaphores already held, and starts over at the beginning of the semaphore locking sequence. The cost of recovery has to be balanced against the benefits of such an optimistic approach when a conflict does not occur, which is, hopefully, most of the time. In fact, when a collection of data structures is distributed over a set of locks by some method such as hashing, lock clashes can be expected to be extremely rare except in the case where multiple processes are concurrently operating on the very same data structures.

In contrast to MUNIX or the Purdue University Dual VAX UNIX system, the modifications to the 3B20A kernel were far more extensive, involving not only the addition of semaphores, but the significant modification of whole data structures and algorithms to minimize contention for locks. Nonetheless, many traditional features of the traditional UNIX kernel (e.g. single monolithic kernel, processes with user/kernel dual existence, and reliance on self-service) remain.

This operating system is a version of UNIX System V, so parallel applications can be supported through the standard shared memory and semaphore facilities. There are not, however, extensive software tools available to assist in the construction of parallel applications.

2.3.8. Sequent Balance 21000/DYNIX

Sequent Computer Systems was one of the first companies to offer commercially a general purpose symmetric multiprocessor with more than 4 processors. The Balance 8000 (Beck and Kasten [19]) consisted of as many as 12 processors based on the National Semiconductor 32032 microprocessor interconnected by a very high performance bus to several memory modules and peripherals. The system was later extended to accommodate 30 processors with the introduction of the Balance 21000. Cache memory is included with each processor to reduce bus contention, and bus-watching logic associated with each cache keeps them consistent. Custom VLSI bus interface chips provide a variety of services, including distributing interrupts to the least busy processors and providing a special low-overhead mutual exclusion mechanism.

The operating system for the Balance 21000 is a version of UNIX called DYNIX. It is a fully symmetric operating system, including fully symmetric I/O. The implementation is similar to the 3B20A UNIX system, although it is based on a different version of UNIX (4.2BSD). Unlike the 3B20A operating system, DYNIX provides some software tools to support parallel applications (Beck and Olien [20], Sequent [179]). There are two special hardware synchronization facilities, one called *gates* and one called Atomic Lock Memory (ALM); the former is used exclusively in the DYNIX kernel, the latter by user programs (since the Balance 21000 is a multiuser machine, the kernel provides a special pseudo-device driver to control the allocation of the ALM). This apparent duplication of hardware facility and awkward programmer interface probably indicates that the designers did not realize the potential attraction users would have for parallel programming until late in the design cycle.

The kernel uses spin locks and process-switching counting semaphores, both multiplexed on top of gates. As in the 3B20A UNIX, a conditional form of semaphore is provided, and a similar approach to deadlock avoidance is used.

2.3.9. Mach

The Mach project at Carnegie Mellon University began as a marriage of the Berkeley UNIX kernel and the Accent message-oriented operating system (Rashid and Robertson [166]). The interprocess communication features of Accent were adapted to the UNIX kernel, together with a new design for the separation of processes into *tasks* and *threads*, and a new virtual memory system designed to provide more flexible sharing along with improved portability and efficiency (Rashid *et al.* [165]). Two trends emerged for Mach development:

- The first was the gradual improvement of the system as the new and old components became better integrated. This trend culminated with Mach 2.5 and variants adopted by or developed for a number of significant commercial and research systems.
- The second was the move to *kernelize* Mach, moving as much traditional UNIX functionality as possible into one or more user-mode servers. The culmination of this trend is the Mach 3 microkernel, which has also been used to support a number of significant projects.

Mach includes a significant amount of machinery for distributed, loosely-coupled, computing. It is fundamentally server-based and object-oriented. Messages are typed objects and message ports are protected by capabilities, which may be included in messages. Tasks, threads, and memory objects are all referenced by and controlled with ports and messages.

The Mach implementation of message passing is highly interdependent with that of shared memory. In some circumstances, each is implemented in terms of the other. Passing of large messages is implemented with copy-on-write techniques whenever possible, and memory can be shared between tasks, even on different machines, using *external pagers*, which are implemented with message passing.

2.3.10. IBM RP3

The IBM Research Parallel Processing Prototype (RP3), described initially by Pfister *et al.* [162], shares a number of attributes with the NYU Ultracomputer, and indeed there was significant cooperation and cross-fertilization between the two projects. Both machines have potentially many processors and support combinable Fetch&Φ operations, but the RP3 design calls for two separate processor/memory interconnection networks, one with combining and one without, and sports a unique variable interleaving feature for memory addressing. A 64 processor subset, without the combining network, was constructed and used for over two years.

The initial operating system work for RP3 was carried out jointly between researchers at IBM and at NYU; many basic design features of Symunix-2 took shape during that period (for the NYU perspective on the design during this period, see Edler *et al.* [74]). Eventually, IBM decided to modify a version of Mach for the RP3, and this was the system that was actually used. See Bryant *et al.* [41] for a retrospective view of the Mach effort and the RP3 project in general. The version of Mach used for the RP3 work included UNIX compatibility in the kernel, mostly serialized to run on only one of the processors; this proved to be a bottleneck, but few of the workloads studied on the machine were OS-intensive, so it didn't seem to matter much.

The RP3 was run as a single-user system, so the full time-sharing and multiprogramming capabilities of the operating system were not utilized. To avoid OS scheduling overheads, typical usage tended to allocate enough kernel threads to match the number of processors being used and perform further scheduling at the user level with the minimum OS intervention possible, i.e., as in a user-mode thread system.

Since Mach was initially developed with uniprocessors and small symmetric multiprocessors in mind, the RP3 was one of the first NUMA adventures for Mach, and certainly the largest until that time. A lot of emphasis was placed on using the variable interleaving feature of the hardware, but only in its two extreme modes: a page could be interleaved across all memory modules or fully contained in one memory module; none of the intermediate degrees of interleaving were used.

2.3.11. DEC Firefly/Topaz

Firefly is a shared memory multiprocessor developed at the DEC Systems Research Center (Thacker *et al.* [194]). It usually includes five VAX microprocessors on a bus with caches kept coherent by hardware, but a few systems with up to nine processors have been built. The primary operating system for the Firefly is called Topaz, and is written primarily in Modula-2+. It consists of a small portion, called the *nub*, which executes in kernel-mode and provides virtual memory, thread scheduling (multiple threads can run within a single address space), device drivers, and remote procedure calls. The remainder of the operating system, including the file system, Ultrix emulator,[21] and window system, runs with several threads in a separate address space. Essentially all OS services are distributed across the collection of Fireflys on a local area network, because of the fundamental use of remote procedure calls throughout the system software.

Programs written in Modula-2+ are able to take advantage of a threads module, which includes fork and join operations, a LOCK statement to provide critical sections, and condition variables (Birrell *et al.* [32]).

Anderson *et al.* [7] used the Firefly and modified Topaz to implement their proposal for *scheduler activations*. Scheduler activations are execution contexts for user level threads, implemented with a threads packages such as FastThreads (see Anderson *et al.* [8]). Unlike ordinary kernel threads, scheduler activations come and go, being created and destroyed by the kernel as needed to provide overall processor allocation to multiple competing jobs. This is made possible by the use of upcalls to notify the user level thread manager of each event affecting the allocation of processors to the application. These events include:

- *Allocation of a new processor*, in the form of another scheduler activation. In this case, the upcall is essentially just an entry point for the user level thread manager, which can simply begin running a thread.
- *Preemption of a processor*. In this case, the upcall is performed on another processor allocated to the same job, or on the first processor reassigned to the job in the case where the only processor currently allocated to a job is preempted. The upcall includes the machine state of the preempted processor as a parameter; this allows the thread manager to save

[21] Ultrix is a variant of UNIX offered commercially by DEC.

the state for later execution, according to whatever user level scheduling policy is in force.

- *An activation has blocked* in the kernel. In this case, a new activation is created and run to take the place of the blocked one, and the upcall is basically the same as the *allocation of a new processor* upcall.
- *An activation has unblocked* in the kernel. This upcall is similar to the upcall for preemption, except there are two interrupted machine state parameters: one for the thread that was running before the upcall, and one for the thread that unblocked.[22]

Just as the kernel informs the user of certain events, the user-mode thread manager is expected to inform the kernel as its needs change. There are two such conditions, essentially corresponding to the need for more or less processors in response to varying application demand.

The differences between scheduler activations and the activities of Symunix-2, as we will describe in section 4.3,[23] are more a matter of concept than of implementation. The virtual machine model is somewhat different. With scheduler activations, you get a shared address space "virtual multiprocessor" with occasional notices (upcalls) from the kernel when processors are allocated or taken away. With the Symunix-2 approach, the virtual multiprocessor is made up of somewhat more independent processes, and you get signals for asynchronous event notification (e.g., preemption, asynchronous I/O completion, or page fault) and return codes for synchronous ones (e.g., blocking in the kernel). In general, we can expect implementation details to dominate any performance differences between the two systems.

2.3.12. Psyche

Psyche (Scott *et al.* [178, 177], Marsh *et al.* [144]) is an operating system developed at the University of Rochester for the BBN Butterfly Plus [18]. The primary goals of the project were to efficiently utilize the NUMA hardware of the Butterfly and to provide as much flexibility as possible to the users, i.e., not to hide or bury the flexibility inherent in the architecture.

Threads, if required in a particular programming model, are implemented in user-mode, with low-level interactions with the kernel designed to minimize overhead and maximize user level control. The overall flavor is close to that of the scheduler activations work and the Symunix-2 work. Like Symunix-2, Psyche uses shared memory to provide some very low overhead communication between the user and kernel. In particular, Psyche provides advance notice of preemption to the user, in a manner similar to the Symunix-2 `SIGPREEMPT` mechanism; this notice is also available without interrupt, via polling a shared memory location. Also like Symunix-2, Psyche provides asynchronous system calls, but notification of blocking is via upcall, i.e., more in the style of scheduler activations.

---

[22] This event really occurs only when the scheduler activation is ready to return to user-mode; thus it could block and run several times entirely within the kernel without any notification to the user level.

[23] For comparability, we also need to consider the Symunix-2 temporary non-preemption mechanism, in section 4.6.4, and the `SIGPREEMPT` signal, in section 4.6.5.

Unlike Symunix-2, Psyche is not a very UNIX-like system. The model of protection and addressing is very much more elaborate and flexible, including the following:

- A uniform address space divided into nestable *realms*, which are a data abstraction mechanisms.
- Realms exist within potentially overlapping protection domains; intra-domain realm operations are much cheaper than inter-domain ones.
- Protection provided by keys and access lists.
- Virtual processors, called *activations*, which are the execution vehicle for *processes*, which move about from protection domain to protection domain.

2.4. Chapter Summary

We have seen a sampling of the number and variety of operating systems designed for shared memory MIMD computers. A number of trends show themselves to be popular, but in some cases two or more conflicting models vie for dominance, with no real proof of which is best. Some of these trends are now reviewed:

Modular Design

: All systems strive for this, for basic software engineering reasons. Nonetheless, some systems enforce modularity at the hardware, firmware, or operating system kernel level, while other systems leave this issue unenforced, or to be enforced by compilers. Most of the message passing, object-oriented, and capability-based systems fall into the former category, while the UNIX systems tend to fall into the latter.

Lightweight vs. Heavyweight Processes

: The term "lightweight process" is not generally well defined, but usually refers to an implementation which supports very fast context switching from one process to another, normally by running both processes in the same address space. Tasks within OS/VS2 are examples of a type of lightweight processes that is recognized and scheduled by the operating system scheduler. A number of UNIX systems have been extended along these lines, such as UNIX for the HEP and Cray X-MP, and the Mach operating system. A yet lighter approach is to manage these lightweight processes at the user level, with only indirect support from the operating system. This is the approach of the Symunix and Psyche operating systems, and of Anderson's scheduler activations.

Server vs. Self-Service Model

: A design that adopts the self-service model allows processes to perform certain functions directly on shared data rather than requiring them to send messages to special server processes. The primary arguments in favor of messages over shared data are that message passing encourages modularity, improves isolation, and is a more structured form of synchronization. In addition, message passing is naturally extended to more loosely-coupled environments without shared memory. Arguments against message passing are generally that it is easier to achieve high performance with direct sharing of data and explicit synchronization (e.g. via semaphores). For multiprocessors, the key point is that a single server can become a bottleneck in a large system. For this reason, many server-oriented multiprocessor systems allow multiple servers of each type, so called multi-threaded servers. The UNIX kernel has traditionally adopted the self-service approach. When a process makes a *system* *call*, it is transformed into a privileged process executing specified code within the kernel. While the lack of support for multiprocessors has long been regarded as a deficiency in UNIX, its basic self-service orientation is probably one reason why it has been seen as such an attractive vehicle for multiprocessor research.

Chapter 3: Some Important Primitives

We begin our study in a bottom-up fashion, by examining in detail the most fundamental software primitives used in the operating system kernel. In addition to laying the groundwork for understanding the material in the following chapters, these primitives (and obvious derivations thereof) are important for any use of the target architectures, independent of Symunix itself.

We make two general claims in presenting this material:

- That these primitives, with the algorithms and implementations we suggest, are key elements of a solution for many problems encountered in highly parallel software systems.
- That these primitives, viewed as modular interfaces, are key to the portability of such software.

Much of the material in this chapter is the result of our particular approach to highly parallel software development, which we can describe as a 7 step procedure:

(1) Start (at least conceptually) with a traditional serial UNIX kernel implementation.

(2) Apply rudimentary parallelization techniques.

(3) Identify performance bottlenecks.

(4) Design solutions for the bottlenecks.

(5) Attempt to redefine problems (i.e., design module interfaces) to accommodate highly parallel solutions, while still admitting serial or "lowly parallel" alternatives.

(6) Implement the highly parallel solutions.

(7) Repeat steps 3 through 6 until satisfied.

Although the resulting modular interface admits significant implementation flexibility, it cannot accommodate *all* unanticipated implementations (nor can any other modular interface). The experiments described in section 8.3 provide some evidence of the feasibility of our approach.

In the remainder of this chapter, we describe modular interfaces designed in the manner just described, together with the current implementation for the target architecture class. In most cases, we also suggest reasonable alternatives for machines outside that class.

It should be emphasized that most of the *details* presented in this chapter are entirely within the context of the Symunix-2 kernel. Chapter 7 covers many of the same topics in the context of the user-mode environment. Of course there are many other environments where the same general principles also apply, but we believe in fully exploiting as many as possible of the subtle factors that distinguish one environment from another. For example, our kernel environment runs in privileged mode, with access to all hardware capabilities and direct control over interrupts, scheduling decisions, etc. In contrast, code that runs in an unprivileged hardware environment must rely on a more privileged layer for any such control, and this may alter details (such as signal vs. interrupt masking) or general approaches (e.g., hybrid or "2-phase blocking" techniques may be appropriate (Ousterhout [160]) (Karlin *et al.* [125]); see section 7.3.3). As another example, the requirements of real-time systems differ in significant ways that would certainly impact both the interfaces and implementation choices described here. The choice of the C programming language also has enormous influence on our interface choices, as has the general UNIX model we have adopted. Each system has its own unique characteristics that must be accommodated and exploited wherever possible.

The remainder of this chapter is organized as follows. We begin by addressing the use of Fetch&Φ operations (§3.1), followed by some simple composite operations that are useful enough to be worth treating separately (§3.2). Sections 3.3 and 3.4 address the conceptually minor issues of instrumentation and interrupt masking, which are then used extensively in the remainder of the chapter. The real meat of the chapter is sections 3.5 and 3.7, which address busy-waiting and list structures, respectively. Sandwiched between these, in section 3.6, we briefly address the topic of interprocessor interrupts, as they depend on the facilities described in section 3.5 and are in turn depended upon by those in section 3.7.

3.1. Fetch&Φ Functions

One of the central features of the target architecture class (§1.3) is the availability of combinable Fetch&Φ operations. While the importance of Fetch&Add cannot be over-emphasized, many of our algorithms also make significant use of other Fetch&Φs, especially Fetch&And, Fetch&Or, and Fetch&Store.

Table 1, on the next page, lists all the Fetch&Φ functions used in Symunix-2. For example, the code

```
i = faa (&x, 1);
```

atomically copies (the old value of) `x` to `i` and adds 1 to `x`.

The choice of data type is difficult. We have used `int`, in keeping with the original definition of `int` as having the "natural size" for integers on a machine (Kernighan and Ritchie [128]). The way the industry is evolving, this may be a poor choice. For example, a 64 bit machine might very reasonably support Fetch&Φ only for 64 bit integers, but the `int` type may be 32 bits. Specifying `int` for the Fetch&Φ functions was probably naive; perhaps a derived type, such as `faa_t`, would be better. But what about machines with Fetch&Φ support for multiple integer sizes?

The functions `fai`, `fad`, and their variants require fewer keystrokes and can be slightly more efficient to execute than the more general `faa`, even on architectures that support Fetch&Add well. Furthermore, they are important special cases in their own right (see Freudenthal and Gottlieb [86]).

Most of the basic functions have variations that operate on unsigned integers. In each case the basic function name is prefixed with `u`. The primary advantage of these functions over their basic counterparts is in allowing the compiler to perform more type checking. On typical two's-complement machines, the underlying implementation will be the same as the signed variants.

Function	*Purpose*
`int faa(int *a, int i)`	Fetch&Add
`int fai(int *a)`	Fetch&Increment (`faa(a,1)`)
`int fad(int *a)`	Fetch&Decrement (`faa(a,-1)`)
`unsigned ufaa(unsigned *a, unsigned i)`	Unsigned Fetch&Add
`unsigned ufai(unsigned *a)`	Unsigned Fetch&Increment
`unsigned ufad(unsigned *a)`	Unsigned Fetch&Decrement
`void vfaa(int *a, int i)`	Add to memory (`(void)faa(a,i)`)
`void vfai(int *a)`	Increment memory (`vfaa(a,1)`)
`void vfad(int *a)`	Decrement memory (`vfaa(a,-1)`)
`void vufaa(unsigned *a, unsigned i)`	Unsigned add to memory
`void vufai(unsigned *a)`	Unsigned increment memory
`void vufad(unsigned *a)`	Unsigned decrement memory
`int fas(int *a, int i)`	Fetch&Store (a.k.a. swap: $\Phi(x,y)=y$)
`unsigned ufas(unsigned *a, unsigned i)`	Unsigned Fetch&Store
`void *pfas(void **a, void *i)`	Fetch&Store for generic pointers
`int fase0(int *a, int i)`	Fetch&Store if old value = 0
`int fasge0(int *a, int i)`	Fetch&Store if old value ≥ 0
`void *pfase0(void **a, void *i)`	`pfas(a,i)` if old value = `NULL`
`int faand(int *a, int i)`	Fetch&And
`int faor(int *a, int i)`	Fetch&Or
`unsigned ufaand(unsigned *a, unsigned i)`	Unsigned Fetch&And
`unsigned ufaor(unsigned *a, unsigned i)`	Unsigned Fetch&Or
`void vfaand(int *a, int i)`	And to memory (`(void)faand(a,i)`)
`void vfaor(int *a, int i)`	Or to memory (`(void)faor(a,i)`)
`void vufaand(unsigned *a, unsigned i)`	Unsigned and to memory
`void vufaor(unsigned *a, unsigned i)`	Unsigned or to memory

Table 1: Fetch&Φ Functions.

Most of the basic functions have variations that return no value at all; the function name is prefixed with `v`. Using one of these functions makes clear that only the side effect is desired, and allows a slightly more efficient implementation on some machines.

Fetch&Store for generic pointers, `pfas`, is another valuable variation. On architectures in which a generic pointer has the same size as an integer, `pfas` may actually be implemented identically to `fas`. Likewise, `pfase0` will be implemented identically to `fase0` on machines where the pointer and integer sizes match.[24]

On any given machine, some functions will be supported directly by hardware, and some won't. There is a compile-time symbol that can be checked for each function to see if it

[24] Technically, a different implementation could be required for machines where the representation of a "null pointer" in C is not a bit pattern of all zeros, but this is not a real consideration on most machines (ANSI C [5]).

is supported by hardware or emulated in software. This is important because emulated functions aren't atomic with respect to loads, stores, or other hardware-supported functions. Such compile-time symbols provide a brute force method of writing "machine-independent" code by providing alternative implementations.

The compile-time symbols are constructed by taking the name of the function, capitalizing it, and prepending the string `HARD_` (for example, `faa` is supported by hardware if `HARD_FAA` is *TRUE* (evaluates to 1)). When not supported directly by hardware, some of the functions are emulated by other functions, and some are directly emulated by software. For example, if `fai` isn't supported directly by hardware, it is emulated with `faa`. There are four cases:

`HARD_FAA && HARD_FAI`
: Both `faa` and `fai` are individually implemented using different hardware mechanisms, and it can be assumed that `fai(x)` is more efficient than `faa(x,1)`.

`HARD_FAA && !HARD_FAI`
: Only `faa` is implemented directly with hardware, and `fai(x)` has the same implementation as `faa(x,1)`.

`!HARD_FAA && HARD_FAI`
: Only `fai` is implemented directly with hardware; `faa` is emulated by software. As a result, `faa` will not be atomic with respect to loads, stores, `fai`, or other hardware-supported Fetch&Φ functions.

`!HARD_FAA && !HARD_FAI`
: Both functions are implemented by software emulation. The emulated functions are atomic with respect to each other, and to other emulated Fetch&Φ functions, but not to loads, stores, or hardware-supported Fetch&Φ functions.

Table 2, on the next page, shows the emulation relationships for all the Fetch&Φ functions. Full atomicity will be guaranteed over loads, stores, and all the functions in Table 1, if the nine functions in the first section of Table 2 are implemented by hardware.

While the `HARD_` symbols are helpful for writing code that is portable between machines with differing architectural support for Fetch&Φ operations, it doesn't indicate whether or not concurrent accesses to the same memory location are combined. Another set of compile-time symbols is provided for that. Each function in Table 1 has an associated symbol constructed by taking the function name, capitalizing it, and prepending the string `COMBINE_`. The symbol `COMBINE_`*FOO* evaluates to *TRUE* if `HARD_`*FOO* does, and if combining of concurrent *foo* operations on the same variable is supported. An example of the use of `HARD_UFAS` and `COMBINE_UFAS` can be seen in section 3.7.4 on page 119. In addition to `COMBINE_` symbols for the functions in Table 1, load and store operations have symbols to indicate their combinability too (`COMBINE_LOAD` and `COMBINE_STORE`). An example of using `COMBINE_STORE` appears in section 7.3.5 on page 304. Of course, if loads aren't combinable, but Fetch&Add is, the programmer can use `faa(`*addr*`,0)` for a combinable load. Likewise, `(void)fas(`*addr*`,`*value*`)` can be used for a store.

3.2. Test-Decrement-Retest

Gottlieb, *et al.* [94] identified the *Test-Decrement-Retest* and *Test-Increment-Retest* paradigms; we make use of the former. The basic idea is to decrement a variable (atomically, using

Function	*Emulated by*
faa	
ufaa	
fas	
pfas	
fase0	software
fasge0	
pfase0	
faor	
faand	
fai	
fad	
vfaa	faa
vfai	
vfad	

Function	*Emulated by*
ufai	
ufad	
vufaa	ufaa
vufai	
vufad	
ufas	fas
ufaor	
vfaor	faor
vufaor	
ufaand	
ufaand	faand
vufaand	

Table 2: Fetch&Φ function emulations.

Fetch&Add), but to undo the effect if the variable goes negative. To avoid livelock when performing this operation in a loop, the variable is first tested, so the decrement is avoided altogether if success seems unlikely. We have two boolean functions[25] for this, `tdr` and `tdr1`:

```
int
tdr (volatile int *a, int i)
{
    if (*a < i)
        return 0;
    if (faa (a, -i) < i) {
        vfaa(a,i);
        return 0;
    }
    return 1;
}
```

```
int
tdr1 (volatile int *a)
{
    if (*a <= 0)
        return 0;
    if (fad(a) <= 0) {
        vfai(a);
        return 0;
    }
    return 1;
}
```

3.3. Histograms

Histograms, as described here, are not needed for correct system operation. In fact, they are completely eliminated at compile time unless the symbol `INSTRUMENT` is defined to the C preprocessor. Even when not disabled they are inexpensive, and are widely used in the Symunix-2 kernel. Two types of histograms are supported:

[25] Normally inlined.

Function	*Purpose*
`hgraminit(histgram *h, unsigned nbuck, char *name)`	Allocate buckets and initialize
`ehgraminit(ehistgram *h, char *name)`	Initialize
`hgramabandon(histgram *h)`	Stop using `histgram`
`ehgramabandon(histgram *h)`	Stop using `ehistgram`
`hgram(histgram *h, unsigned v)`	Increment bucket `v`
`ehgram(ehistgram *h, unsigned v)`	Increment bucket $1 + \lfloor \log \mathtt{v} \rfloor$
`qhgram(histgram *h, unsigned v)`	Quicker `hgram`
`qehgram(ehistgram *h, unsigned logv)`	Increment bucket `logv`

Table 3: Histogram Support Functions.

`histgram`
: These have a variable number of buckets, dynamically allocated when the histogram is initialized. They are very simple, and suitable for a wide range of purposes.

`ehistgram`
: These are *exponential histograms*, and have a fixed number of buckets, `NEXPHIST`. They are generally used to count values whose frequencies are expected to decrease approximately exponentially. Examples of exponential histograms usage are given in section 3.5.1 and in section 3.7.2.

The functions provided to deal with histograms are given in Table 3, above. Each histogram includes a reference count, incremented by the initialization function with Fetch&Increment so that only the first call matters for each histogram. As the histograms are initialized, they are added to a simple linked list. Histograms are never deallocated, but may be *abandoned*, which just decrements the reference count.

Output is done externally, with a program using the `/dev/kmem` interface to read all the histograms from the linked list. A separate program can then format the results for printing or process it in various ways. The name and reference counts have no purpose other than helping to identify and interpret the data.

The buckets may be incremented in two ways. The "normal" way, using the `hgram` or `ehgram` functions, checks for too-large values, and redirects them to the final bucket. The "quick" way, using the `qhgram` or `qehgram` functions, does no such checking, and, in the case of `qehgram`, assumes the logarithm has already been computed.

3.4. Interrupt Masking

The traditional UNIX kernel mechanism for masking interrupts is embodied in a set of routines `spl`*n*, originating from the PDP-11 "spl" instruction,[26] to set the processor's "priority level" to n, where $0 \le n \le 7$; level 0 masks no interrupts, level 7 masks all. More recently, each UNIX implementation has adjusted the semantics of these routines to suit its own

[26] Spl stands for "Set Priority Level".

purposes in various ways, but the notion of a priority-based scheme is still in general use.

Most MIMD designs provide a hardware interrupt mechanism on a per-processor basis. This means that each processor keeps track independently of which kinds of interrupts are pending or masked, and a device interrupt request must wait for the attention of a particular processor. This does not preclude a more sophisticated approach to distribute device interrupt requests to processors in order to achieve certain goals, such as reduced interrupt latency or meeting real-time constraints. For example, the Sequent Balance [19, 80] and Symmetry [139] machines have special hardware to distribute interrupt requests to the processor running with the least priority. A similar facility is available for multiprocessors built using some recent Intel Pentium processors [200].

Interface: Overview

The set of functions to manipulate interrupts and their masks in Symunix-2 is summarized in Table 4, below. The key operations are sending a soft interrupt (`setsoft`*t*), masking interrupt categories (*f*`spl`*c*), and unmasking interrupt categories (*f*`spl0`, *f*`splx`). This core set of functions is slightly complicated by as many as three variant *flavors* of each masking function; the purpose of the variants is to improve efficiency through specialization and to

Function	*Purpose*
`setsoft`*t*`()`	Issue soft interrupt of type *t*

f`spl`*c*
`spl`*c*`()`
 	
`vspl`*c*`()`
`qspl`*c*`()`
	Mask interrupts of category *c* (Table 5)
	Plain: disallow unmasking side effects
	and return previous mask
	Void: return nothing
	Quick: allow unmasking in some cases
	and return nothing

f`spl0`
`vspl0()`
`qspl0()`
	Unmask all interrupts
	Void: return nothing
	Quick: omit soft interrupt check
	and return nothing

f`splx`
`vsplx()`
`qsplx()`
	Restore previous interrupt mask
	Void: return nothing
	Quick: return nothing
	soft interrupt check optional

`rpl()`	Return current interrupt mask

`splcheck`*x*
`splcheck0()`
`splcheckf(`*m*`)`
	Check functions (§3.5/p50)
	Momentarily allow any interrupt
	Generate custom check function for mask *m*

`issplgeq(`*m1*`,`*m2*`)`	Does *m1* mask a superset of interrupts masked by *m2*?
`fakespl`*c*`()`	Fabricate interrupt mask for category *c*

Table 4: Interrupt Manipulations.

improve system debuggability by allowing more extensive run-time checks for correct usage.[27] For example, by using a "quick" function, `qspl`*c*, in certain places instead of a "plain" one (`spl`*c*) or "void" one (`vspl`*c*), we can avoid unnecessary run-time checks designed to prevent unmasking side effects.

Interface: Details

The following semantics have been chosen for Symunix-2 to define the limits of portability, clarify behavior in certain situations, and provide rules for improved run-time checking of correct usage. While not every detail is fundamental to all highly parallel operating systems, clearly defined rules of usage are an important tool for construction of correct software in any environment with interrupts.

General. *Interrupt masks* are per-activity. They define the categories of interrupts masked when the activity runs. A single device request for an interrupt eventually generates an interrupt on a single processor.

Hard and Soft Interrupts. There are two general kinds of interrupts: "hard" and "soft". Hard interrupts are the ordinary kind, generally requested by an I/O device, or clock. Soft interrupts are requested by software (presumably because that specific kind of interrupt may be masked, otherwise a function call would suffice). Soft interrupts are per-processor, in the sense that there is no direct machine-independent way to request a soft interrupt for another processor.

The main purpose of soft interrupts is to reduce hard interrupt latency, by reducing the amount of time that hard interrupts are masked. Only the most time-critical functions should be performed by hard interrupt handlers, deferring as much as possible to soft interrupt handlers.

Interrupt Categories. The machine-independent part of the kernel "knows about" the interrupt categories given in Table 5, on the next page. There is a partial ordering assumed among these classes:

`resched` ≤ {other soft interrupts} ≤ `soft` ≤ {hard interrupts} ≤ `all`.

So masking any of the hard interrupts is guaranteed to mask all of the soft ones, as will masking the entire soft category. The `resched` category must be the lowest; masking any other interrupt category also masks rescheduling interrupts (§4.6.1). Other than that, there is no guarantee that any particular relationship holds between two soft or two hard categories. This gives an appropriate degree of flexibility to the implementation, allowing such possibilities as a single priority system, a dual priority system (with one priority each for hard and soft interrupts), or unordered independently maskable interrupt types within each of the hard and soft groups.

[27] The run-time checks are a kernel compile-time option.

Category	*Meaning*
`all`	all interrupts
`clock`	hard clock interrupts
`imp`	hard network interrupts
`apc`	hard asynchronous procedure calls
`soft`	all soft interrupts
`sclock`	soft clock interrupts
`sapc`	soft asynchronous procedure calls
`srbc`	soft random broadcasts
`tty`	soft tty interrupts
`bio`	soft block I/O interrupts
`net`	soft network interrupts
`log`	soft log device interrupts
`resched`	soft rescheduling interrupts

Table 5: Interrupt Mask Categories.

Choosing a Category to Mask. When locking is involved, it is best in principle to mask the lowest (smallest) interrupt category consistent with a deadlock-free solution. Deadlock can result when an interrupt handler tries to acquire a lock already held by the processor prior to the interrupt or, more generally, when an interrupt handler forges the last link in a waits-for chain among processors, leading eventually back to the same interrupted processor. The standard solution is to mask the relevant interrupt before acquiring the lock in the main-line code and not to unmask it until the lock is released. Masking a category larger than necessary can cause a needless increase in interrupt latency.

Although, in principle, this position is correct, there are often conflicting considerations.

- Slow but low priority interrupts can increase lock holding times; this increases lock latency, damaging performance in a critical area.
- Keeping track of many different interrupt categories is hard, leading to software bugs.

Since the creation of soft interrupts is partially aimed at separating handler functionality that is more tolerant of interrupt latency from that which is more needful of low latency, the cost of masking a soft category is effectively quite low. For these reasons, the entire soft interrupt category is generally used when masking is required for the code sections presented in this dissertation. As already suggested, a system in which all soft interrupt categories are assigned the same masking priority may be quite sensible.

Check Functions. The *check function* `splcheck0()` momentarily unmasks all interrupts. The *check function generator* `splcheckf(`*m*`)` takes an interrupt mask as returned by a plain spl routine or `rpl`, and returns a pointer to a check function that momentarily restores the interrupt mask. The use of check functions will be described in section 3.5 on page 50.

Mask Comparisons. A data type representing an interrupt mask, `spl_t` is returned by the `splc`, `rpl`, and `fakesplc` functions (see Table 4). The `issplgeq(`*m1*`,`*m2*`)` boolean function examines two interrupt masks and returns *TRUE* if *m1* masks a superset of the interrupt categories masked by *m2*. The fakespl*c*() functions return the same interrupt mask as the following code would,

```
{
      qspl0();
      vsplc();
      return rpl();
}
```

but without actually modifying the processor's real interrupt mask.[28] On most machines, they should evaluate to a constant of type `spl_t` that reflects the "absolute" priority of the interrupt category *c*.

Several useful constructs are handled simply by these routines. As an example, consider a function designed to be called with a certain category of interrupts masked. A useful practice is to place an *assertion* to that effect at the beginning of the function; this provides a valuable "living comment", as well as a run-time check that can be disabled at compile time when one is satisfied with its validity. An assertion tests an expression, causing a descriptive message and a controlled system crash ("panic") if it turns out to be false. In this case,

```
assert (issplgeq (rpl(), fakesplc()));
```

is appropriate, as it claims that all interrupts of category *c* are masked by the current interrupt mask.

Another, more essential, use of `rpl`, `fakesplc`, and `issplgeq` will be illustrated in section 3.7.3 on page 107.

3.5. Busy-Waiting Synchronization

There are two general kinds of explicit synchronization used within the kernel: *busy-waiting* and *context-switching*. The distinction arises when the semantics of synchronization require that a processor be delayed. In the busy-waiting case, such a processor continually re-tests the required condition until it is satisfied; in the context-switching case, a *context-switch* is made so that the processor can proceed with other useful work in the meantime. Busy-waiting methods are always more efficient if waiting times are short. Busy-waiting is also the primitive mode: the context-switching methods all employ some busy-waiting techniques in their implementation.

Interface

Table 6, on the next page, lists the kinds of busy-waiting synchronization employed in the Symunix-2 kernel. Among the semaphore and lock types, a certain uniformity of names, types, parameters and instrumentation was sought. All the functions begin with `bw`, to clearly distinguish them from context-switching functions (§4.5), and one or two characters to identify the specific type of synchronization (e.g. `l` for binary semaphore (i.e., plain "lock") and `rw` for readers/writers lock). Each semaphore or lock has similarly named functions for initialization and destruction, and common function variants have similar naming conventions. All initialization functions are declared to return a boolean value, giving the

[28] This is why it is called "fake".

Mechanism	*Data Type*	*See page*	*Key Functions*
delay loops	–	51	BUSY_WAIT, FBUSY_WAIT
binary semaphores	bwlock	58	bwl_wait, bwl_signal
counting semaphores	bwsem	54	bws_wait, bws_signal
readers/writers locks	bwrwlock	59	bwrw_rlock, bwrw_wlock, bwrw_runlock, bwrw_wunlock, bwrw_rtow, bwrw_wtor
readers/readers locks	bwrrlock	64	bwrr_xlock, bwrr_ylock, bwrr_xunlock, bwrr_yunlock
group locks	bwglock	68	bwg_lock, bwg_sync, bwg_unlock

Table 6: Busy-Waiting Synchronization Mechanisms.

implementation the freedom to perform some additional dynamic resource allocation at that time (at least in principle).[29] Instrumentation features, such as histogramming of wait times (§3.3), are completely suppressed if the preprocessor symbol `BW_INSTRUMENT` is undefined or defined to 0. For convenience, the macro `bw_instrument(`*x*`)` evaluates to *x* if `BW_INSTRUMENT` is defined to a non-zero value, and evaluates to nothing if `BW_INSTRUMENT` is undefined or defined to 0. Using a construction such as

```
        bw_instrument(qehgram(&some_hg);)
```

is less disruptive to the textual appearance of code than the equivalent

```
#if BW_INSTRUMENT
        qehgram(&some_hg);
#endif
```

especially when buried inside loops and conditionals.

There is a strict separation of interface and implementation. The most efficient algorithms and/or data structures for a given machine can be used. Taking advantage of this freedom of implementation is one of the reasons for the richness of the defined interface (number of synchronization types and function variants): by providing a richer interface, and writing the machine-independent parts of the kernel to use the least powerful mechanism in each situation, portability and efficiency are both improved.

Notably missing from the interface is any kind of barrier (Jordan [122]). To date there has been no need for general barriers within the kernel, but the *group lock* mechanism includes barrier as a fundamental operation (§3.5.6).

None of the interface functions are designed to mask interrupts or block preemption as side effects; the caller must generally also use appropriate functions from section 3.4. A general problem with busy-waiting is that interrupts must be masked *before* beginning to wait for a lock, possibly leading to unnecessarily high interrupt latency. To reduce this impact,

[29] Unfortunately, making use of dynamic allocation also requires the ability to tolerate its failure.

the busy-waiting functions of Symunix-2 are all designed to take a parameter specifying a *check function* to be called (with no arguments) periodically while waiting. A typical locking situation, where no interrupts were previously masked, might be written as

```
vsplsoft();                // mask interrupts
bwl_wait(lock,splcheck0); // get lock
 critical section
bwl_signal(lock);          // release lock
vspl0();
```

In this example, all "soft" interrupts are masked (including preemption, §4.6.1); other possibilities are listed in Table 5, in section 3.4 on page 47. The check function `splcheck0` is called repeatedly while waiting within `bwl_wait`, and does nothing but briefly enable all interrupts, so even if the processor must wait for a long time, interrupt latency will not be severely affected.[30] When the previous interrupt mask is not known to be zero, the typical locking code looks like

```
spl_t s = splsoft();
bwl_wait(lock,splcheckf(s));
 critical section
bwl_signal(lock);
vsplx(s);
```

The function `splcheckf` is a *check function generator*; it returns a pointer to a function (with no arguments) that will briefly restore the interrupt mask `s`. Obviously, the implementation of `splcheckf` is machine-dependent (as are all *spl* functions). In addition to `splcheck0` and `splcheckf`, a `nullf` function is available to be used when no purposeful check function is desired.

Beyond factors specific to the particular algorithms chosen for busy-waiting synchronization, there are two generic issues in minimizing the cost of the common case where no delays are necessary: subroutine call overhead and unnecessary evaluation of check function generators, such as `splcheckf`. Subroutine call overhead is avoided by adopting a *two-stage* approach, where the first stage synchronizes in the absence of contention without any calls, invoking the second stage otherwise. The first stage can be implemented as either a macro or an *inline function*,[31] and the second stage as an ordinary function. To avoid unnecessary evaluation of a check function generator without specific compiler optimizations, macros are used for the first stage, instead of inline functions. In this way, overhead is avoided in the common case, and "macro bloat" is minimized because the second stage contains the delay loops. We use the convention of constructing the name of the second stage function by prepending _ to the first stage macro name.

---

[30] As described in section 2.3.2 on page 27, IBM's OS/VS2 release 2 operating system employed a similar technique while busy-waiting, although the motivation was at least partially for fault tolerance (MacKinnon [141]).

[31] The details of inline functions differ among compilers that support them, since they are not part of standard C [5], but we have developed conventions that allow for reasonable portability, and also allow suppression of inlining when desired for debugging or for compilers that can't support it.

An alternative to check function generators would be to provide the old interrupt mask itself instead of a check function. In essence, this would elevate `splcheckf` to the status of implied universal check function. Although the specific motivation for check function parameters is to handle interrupt masking, they are more general and, in principle, could be used for other purposes as well. By using macros to avoid needless evaluation of `splcheckf`, the cost of the more general mechanism is the same as the less general alternative.

Implementation

For the target class of machine, the algorithms chosen for semaphores and locks are optimized for the common case of no contention. Specifically, this optimization consists of using an algorithm based on the *Test-Decrement-Retest* paradigm (Gottlieb *et al.* [94]), but with the initial test of the first iteration (performed in the first stage) thrown out.

Most of the busy-waiting algorithms to be presented in the following subsections are not immune to starvation, but are protected against livelock. The exception is group lock, in section 3.5.6, which is ticket-based and thus starvation-free. The others would be *probabilistically fair* if we removed the exponential backoff feature (to be described in section 3.5.1 on page 53).

We have accepted potentially unfair algorithms for Symunix because we assume that, most of the time, no delay will occur. The justification for this assumption is based primarily on the fact that higher level algorithms, described throughout this dissertation, really bend over backwards to avoid lock contention.[32] We believe this rationale is acceptable for general purpose computing, but not for certain other environments, e.g., real-time.

Rudolph [174] gives fair versions of semaphores and readers/writers locks, but fairness is sometimes gained only at the cost of additional complexity and overhead.

3.5.1. Delay Loops

Busy-waiting is fundamental to many higher level algorithms for multiprocessors. While the abstract synchronization types, such as semaphores and readers/writers locks, are commonly useful, they are ill-matched to the needs of some algorithms. Sometimes what is needed is the ability to busy-wait until some condition is satisfied. A set of common macros is useful to ensure that each such instance of busy-waiting is performed in a reasonably efficient manner.

Interface

In Symunix-2, the basic busy-waiting macro is BUSY_WAIT (*cond*, *check*, *hg*) which busy-waits until the expression *cond* evaluates to true; *check* is a pointer to a check function (§3.5/p50), and *hg* points to an *exponential histogram* structure (§3.3), used to count the number of times synchronization was achieved after a delay of {1, 2, 4, 8, ..., `NEXPHIST` or more} time units.[33] Often, *cond* is a simple expression that tests the value of a memory location, but

---

[32] Chapter 8 provides some supporting empirical data.

[33] The unit of delay is machine-dependent, and need not be precisely defined as long as the histogram indicates approximate waiting time.

it can be any integer expression. Clearly, a macro must be used rather than a function (even an inline one), because otherwise the condition would be evaluated only once.

Several variants of `BUSY_WAIT` are also useful:

`FBUSY_WAIT(`*cond*`,`*check*`,`*hg*`)`
: Like `BUSY_WAIT`, but `cond` is evaluated *before* entering the loop ("`F`" is for "first"). This is suitable for a one-stage approach, whereas `BUSY_WAIT` is best suited for two-stage situations (where the condition has already have been checked once). Since `FBUSY_WAIT` is designed to handle the first condition test, it also handles histogramming for the zero-delay case.

`XBUSY_WAIT(`*cond*`,`*check*`,`*hg*`,`*iter*`,`*p*`)`
: This version supports situations where multiple conditions must be met successively before synchronization is achieved. In such cases, the total waiting time histogrammed should be the sum of the individual waiting times. `XBUSY_WAIT` meets these needs by exposing two internal variables as parameters: *iter* counts overall waiting time for histogram purposes, and *p* is an index into the histogram buckets; both integer variables should be initialized to 0 before the first call to `XBUSY_WAIT`. Because of `XBUSY_WAIT`'s support for multiple delay loops, it can't increment the histogram bucket itself; that must be done separately after the final delay loop by calling `XBUSY_FINAL`, below. (For examples of the use of `XBUSY_WAIT`, see §3.5.4/p63 and §3.5.5/p67.)

`FXBUSY_WAIT(`*cond*`,`*check*`,`*hg*`,`*iter*`,`*p*`)`
: This is like a combination of `FBUSY_WAIT` and `XBUSY_WAIT`. The condition is checked before the loop, and multiple delay loops may be used in sequence. The zero-delay case is handled, and `XBUSY_FINAL`, below, must be called after the final delay loop. It is okay to mix calls to `XBUSY_WAIT` and `FXBUSY_WAIT`.

`XBUSY_FINAL(`*hg*`,`*iter*`,`*p*`)`
: This macro simply records the histogram value accumulated by the previous sequence of `XBUSY_WAIT` or `FXBUSY_WAIT` loops.

`BUSY_WAIT_NO_INST(`*cond*`,`*check*`)`
: This is like `BUSY_WAIT`, but with no instrumentation support.

`FBUSY_WAIT_NO_INST(`*cond*`,`*check*`)`
: This is like `FBUSY_WAIT`, but with no instrumentation support. The condition is checked before the first delay.

As with all other primitives described in section 3.5, `BUSY_WAIT` and its variations will avoid instrumentation altogether if the preprocessor symbol `BW_INSTRUMENT` is undefined or defined to 0. (In such a case, the `_NO_INST` variants are substituted for the others.)

Implementation

There are a variety of ways to implement this kind of busy-waiting. Momentarily ignoring instrumentation, the simplest is probably something like

```
while (!cond)
        check();
```

But such a tight loop will flood the memory system of some machines, reducing performance by slowing down other processors (including one or more that will eventually make the condition true). This is certainly the case for machines in which all shared memory accesses appear on a single bus, but inserting a delay into the loop will reduce the trouble. The situation is more complicated for machines with cache memories, snoopy caches, directory-based cache coherence, multiple buses, multistage interconnection networks, or local access to shared memory. Anderson [6] investigated the effects of several delay alternatives for several possible machine organizations. Some backoff strategies are tightly integrated into the synchronization algorithm itself; delay macros such as our `BUSY_WAIT` are poorly suited to those situations.

The instrumentation side effect of the `BUSY_WAIT` macros can be accomplished in various ways. Among the most obvious are referencing a high-precision hardware clock and counting loop iterations.

The current implementation of `BUSY_WAIT` for the Ultracomputer prototypes is based on the notion of *exponential backoff*, inspired by the collision handling mechanism of Ethernet (Metcalf and Boggs [150]), together with loop iteration counting for instrumentation. At any given time, the assumption is that the expected additional waiting time is directly proportional to the amount of time already waited, up to some limit. The motivation for exponential backoff is the assumption that the condition to be waited for involves a shared memory access which will cause memory contention for other processors.

```
#define BUSY_WAIT(cond,check,hg)                       \
     do { int _iter = 0;                               \
          unsigned int _p = 0;                         \
          while (1) {                                  \
               (check)();                              \
               if (_iter < (1<<NEXPHIST))              \
                    _iter++; /* avoid overflow */      \
               else                                    \
                    _iter = (1<<(NEXPHIST-1)) + 1;     \
               if ((_iter&(_iter-1)) == 0 &&           \
                   _p < NEXPHIST-1)                    \
                    _p++;                              \
               if (((_iter&(_iter-1)) == 0 ||          \
                    (_iter&(MAX_BW_DELAY-1)) == 0) &&  \
                   (cond)) {                           \
                    qehgram(hg,_p);                    \
                    break;                             \
               }                                       \
          }                                            \
     } while(0) /* require semicolon */
```

It is assumed that `MAX_BW_DELAY`, used to limit the exponential backoff, is a power of two.[34] The choice of the value `(1<<NEXPHIST)` to prevent overflow allows us to count high enough

[34] We have used the value of 128, but this was not the result of extensive testing.

to reach the final bucket of the histogram structure, yet still avoid overflow[35] and maintain the maximum delay period.[36] The expression `_iter&(_iter-1)) == 0` tests whether `_iter` is a power of 2 or not. The function `qehgram` (§3.3) actually increments the histogram bucket indicated by its second argument. The shortest waiting time recordable in this manner by `BUSY_WAIT` is 1; a delay of 0 must be recorded by the caller using `qehgram(`*hg*`,0)`.

The macros for `FBUSY_WAIT`, `XBUSY_WAIT`, `FXBUSY_WAIT`, `BUSY_WAIT_NO_INST`, and `FBUSY_WAIT_NO_INST`, are all defined in a similar style.

3.5.2. Counting Semaphores

Counting semaphores are more general than the somewhat more common binary semaphores (§3.5.3). A common use for counting semaphores in Symunix-2 is in solving *producer/consumer* problems: by setting the semaphore value to *n*, it represents *n* available units of some resource (e.g. buffers). A *wait* (or *P*) operation performs a unit allocation by waiting until the value is positive, then decrementing it. A *signal* (or *V*) operation performs a unit release by incrementing the value.

Interface

The data type `bwsem` is operated upon by the functions listed in Table 7, below. The initialization function is implemented as a macro: `BWS_INIT(s,v,h,n)`, where `s` points to the `bwsem` structure to be initialized, `v` is a non-negative initialization value, `h` points to an exponential histogram structure for instrumentation purposes (§3.3), and `n` points to a string to identify the histogram structure.[37] The reason for the macro is to eliminate the `h` and `n` arguments when instrumentation features are *not* desired, while reducing the need for `#ifdefs` whenever counting semaphores are initialized. The macro simply evaluates to a function call, with either two or four arguments, depending on whether or not instrumentation is desired. `BWS_INIT` is the only part of the interface affected by instrumentation. This is

Function	*Purpose*
`BWS_INIT`	Initialize
`bws_destroy`	Un-initialize
`bws_wait`	Semaphore P operation
`bws_trywait`	Conditional P: success/failure, no delay
`bws_qwait`	Quick P: no contention, no delay, no failure
`bws_signal`	Semaphore V operation
`bws_value`	Return semaphore value

Table 7: Busy-Waiting Counting Semaphore Functions.

---

[35] Assuming `(1<<NEXPHIST)` is representable as a positive integer.

[36] Assuming `MAX_BW_DELAY` is no greater than `(1<<(NEXPHIST-1))`.

[37] The string is printed out with the histogram values when finally extracted from the kernel by a utility program using the `/dev/kmem` interface.

important, because initialization is typically done in fewer places than key operational functions like `bws_wait` and `bws_signal`.

The function `bws_trywait` returns a boolean result, indicating whether the semaphore was decremented successfully after a single attempt. The function `bws_qwait` also makes only a single attempt, but assumes that failure is impossible (i.e., logic dictates that no other processor knows about the lock, or that the semaphore value is greater than the maximum number of processors that could possibly call `bws_wait`, `bws_trywait`, or `bws_qwait`). The implementation should cause a system crash ("panic") if this assumption proves false (and the kernel was compiled with `DEBUG` defined). The only function in this interface that will actually delay execution is `bws_wait`; it should have a two-stage implementation (§3.5/p50).

The "current" semaphore value can be obtained by the function `bws_value`, although it may have changed even before the function returns. This unsynchronized result is most useful for assertion statements and error messages. As originally defined by Dijkstra [59], counting semaphores cannot have negative values, but in our interface, negative values are allowed,[38] with the same interpretation as 0 (i.e., no resource units available).

Implementation: Overview

The current implementation for the target class of machines is based on an algorithm of Gottlieb, *et al.* [94], which can be succinctly expressed in pseudo-code using *Test-Decrement-Retest* and Fetch&Add:

```
void
P(int *sem)
{
        while (!tdr1(sem))
                continue;
}
```

```
void
V(int *sem)
{
        fai(sem);
}
```

Our implementation has been extended with several relatively minor enhancements and stylistic embellishments, but the basic algorithm is only slightly modified. The major enhancement is to make P *optimistic*: the initial test of the first *Test-Decrement-Retest* is eliminated:

[38] Dijkstra also allowed negative semaphore values in a later paper [60].

```
void
P(int *sem)
{
        if (fad(sem)<1) {
                fai(sem);
                while (!tdr1(sem))
                        continue;
        }
}
```

The modified algorithm requires only a single shared memory access in the absence of lock contention.

Implementation: Details

To begin with, a typedef is needed to provide some separation between interface and implementation, and a structure allows the integer semaphore to be accompanied by optional instrumentation information:

```
typedef struct {
     volatile int val;
#if BW_INSTRUMENT
     ehistgram *hg;
#endif
} bwsem;
```

The initialization macro, `BWS_INIT`, evaluates a function with either two or four arguments, depending on instrumentation:

```
#if BW_INSTRUMENT
#define BWS_INIT(s,v,h,n) _bws_init(s,v,h,n)
#else
#define BWS_INIT(s,v,h,n) _bws_init(s,v)
#endif
```

The internal function `_bws_init`, has two alternative definitions:

```
#if BW_INSTRUMENT
int
_bws_init (bwsem *s, int v, ehistgram *h, const char *n)
{
     s->val = v;
     s->hg = h;
     return ehgraminit(h,n);
}
```

```
#else   // !BW_INSTRUMENT
int
_bws_init (bwsem *s, int v)
{
     s->val = v;
     return 1;     // always succeed
}
#endif
```

As described in section 3.3, reinitialization of a histogram is benign; this allows one to be shared between multiple semaphores (or perhaps other busy-waiting synchronization mechanisms as well), while avoiding the inconvenience of having to call both `ehgraminit` and `BWS_INIT`. There is no dynamic resource allocation in `_bws_init` to be undone, so the current implementation of `bws_destroy` only needs to call `ehgramabandon` (§3.3) if `BW_INSTRUMENT` is defined.

The almost trivial P algorithm on page 55 becomes fairly complicated when recast into a two-stage implementation with support for check functions and instrumentation.

```
#define bws_wait(s,f)   /* first stage semaphore P */  \
     do {                                              \
          if (fad(&(s)->val) > 0)                      \
               bw_instrument(qehgram((s)->hg,0));      \
          else                                         \
               _bws_wait(s,f);                         \
     } while (0)        /* require semicolon */
```

Note that, in the favorable case where no delay is required, the complete operation requires only a single Fetch&Decrement (plus histogram overhead, if applicable). It is important for `bws_wait` to be a macro, not an inline function; this avoids evaluation of `f` until the certainty of delay has been determined (`f` is a function pointer expression, e.g. `splcheckf`, a not-quite-trivial function). The call to `qehgram` records a no-delay wait in the histogram. The second stage function is `_bws_wait`, and handles everything when we aren't lucky:

```
void
_bws_wait (bwsem *s, void f(void))
{
     vfai(&s->val);     // undo failed first stage decrement
     BUSY_WAIT (tdr1(&s->val), f, s->hg);
}
```

Note that all optimism has vanished by the time we get to the second stage function; the essence of the original algorithm is preserved by using `tdr1` exclusively from then onward. The remaining embellishments to the original algorithm (check functions, instrumentation, and exponential backoff polling) are bundled up in the `BUSY_WAIT` macro (§3.5.1).

Our implementation of `bws_signal` is almost literally the same as the V algorithm (p55), but we have not yet finished with our counting semaphore extensions. First, a version that only tries once: `bws_trywait`. Such a function is very occasionally useful, e.g. as part of a deadlock avoidance scheme. In all likelihood, such a function will not be used to simulate the general P operation, as in

while (!bws_trywait(*s*))
  ... code of no consequence ...

but for safety, and because we don't really expect the cost of `bws_trywait` to be a critical system performance factor, we choose to implement it faithfully, as a single iteration of the TDR-based P algorithm on page 55.[39]

```
int
bws_trywait (bwsem *s)
{
      return tdr1(&s->val);
}
```

The final variation on the semaphore P operation is an occasional convenience item. There are times when higher level logic guarantees that `bws_trywait` could never fail. In such a case, `bws_qwait` is appropriate: it can simply use the most efficient method of decrementing the semaphore value (even a non-atomic one would be acceptable).

```
void
bws_qwait (bwsem *s)
{
      vfad (&s->val);
      assert (s->val >= 0);
}
```

The assertion is present as a simple debugging aid.

3.5.3. Binary Semaphores

Binary semaphores are simple locks, providing mutual exclusion. In any direct sense, binary semaphores are not useful for developing software that is free of serial critical sections. They are, however, still valuable for highly parallel software because they can be very efficient when lock contention is low; all the examples of binary semaphore usage in this dissertation are excerpted from precisely such situations.

Interface

The data type `bwlock` is operated upon by the functions listed in Table 8, on the next page. The same general characteristics apply to these functions as for counting semaphores (§3.5.2/p54). The initialization value passed to `BWL_INIT` must be either 0 or 1, and the function to return the current lock status is called `bwl_islocked` rather than `bwl_value`.

Implementation

The current implementation for the target class of machines is just a special case variant of counting semaphores (§3.5.2/p56), but other possibilities are quite reasonable. The inclusion of both counting and binary semaphores in the interface allows more flexibility of

[39] As pointed out by Dijkstra [61] and Gottlieb, *et al.* [94], such a P algorithm is subject to livelock without the initial test of the *Test-Decrement-Retest*.

Function	*Purpose*
`BWL_INIT`	Initialize
`bwl_destroy`	Un-initialize
`bwl_wait`	Obtain lock
`bwl_trywait`	Conditional P: obtain lock only if no delay
`bwl_qwait`	Obtain lock (guaranteed)
`bwl_signal`	Release lock
`bwl_islocked`	Return lock status

Table 8: Busy-Waiting Lock Functions.

implementation. For example, many machines (especially small ones) efficiently support an operation such as Test-and-Set or Compare-and-Swap; it is almost trivial to implement binary semaphores with either of these operations, but counting semaphores are harder.

3.5.4. Readers/Writers Locks

Designed to solve the Readers/Writers problem put forward by Courtois *et al.* [50], this kind of lock may be held exclusively (a "write lock") or shared (a "read lock").

Interface

The supported functions are listed in Table 9, below. The initialization function, `BWRW_INIT`, is actually a macro, for the same reason as `BWS_INIT` (§3.5.2/p54): to allow conditional elimination of instrumentation parameters at compile time with a minimum of fuss. For readers/writers locks, two histogram structures are used to separately account for reader and

Function	*Purpose*
`BWRW_INIT`	Initialize
`bwrw_destroy`	Un-Initialize
`bwrw_rlock`	Obtain read lock
`bwrw_tryrlock`	Obtain read lock only if immediate
`bwrw_qrlock`	Obtain read lock (quick−guaranteed)
`bwrw_isrlocked`	Return read lock status
`bwrw_runlock`	Release read lock
`bwrw_wlock`	Obtain write lock
`bwrw_trywlock`	Obtain write lock only if immediate
`bwrw_qwlock`	Obtain write lock (quick−guaranteed)
`bwrw_iswlocked`	Return write lock status
`bwrw_wunlock`	Release write lock
`bwrw_rtow`	Upgrade read to write lock
`bwrw_wtor`	Downgrade write to read lock

Table 9: Busy-Waiting Readers/Writers Lock Functions.

writer waiting time.

The basic lock acquisition functions are of the form `bwrw_`*rw*`lock`, where *rw* is "`r`" for a read lock or "`w`" for a write lock. A lock is released by the corresponding `bwrw_`*rw*`unlock` function. The `bwrw_try`*rw*`lock` functions return a boolean result to indicate if the lock was obtained or not in a single attempt; they never delay execution. The `bwrw_q`*rw*`lock` functions assume the lock is available; failure or delay "can't happen". The `bwrw_is`*rw*`locked` functions return a boolean result the opposite of what the corresponding `bwrw_try`*rw*`lock` function would, if it were called instead.

Upgrading a read lock to a write lock is sometimes useful. The semantics adopted here are designed to allow higher-level algorithms to avoid unnecessary serialization:

- A reader may attempt to upgrade by calling `bwrw_rtow`, which returns a boolean value.
- In the presence of pending writers, an attempt to upgrade will fail, and `bwrw_rtow` will immediately return false, without releasing the read lock.
- If, in the absence of writers, a single reader attempts to upgrade, it will block further readers and wait for the last remaining reader to unlock before seizing the write lock and returning true.
- If, in the absence of writers, many readers attempt to upgrade, exactly one will succeed, as above, while the others fail, without releasing their locks.

The intention is that after failing to upgrade, a reader will release its lock and then immediately attempt to obtain it again, on the optimistic assumption that the writer is a collaborator and will perform actions relieving the reader's need for an exclusive lock.[40]

The downgrade function, `bwrw_wtor`, immediately converts a write lock into a read lock, but if other writers are waiting, an implementation with writer priority will not allow any other readers to join this one.

In order to achieve highly parallel operation, readers/writers locks are used in situations where write lock attempts are expected to be rare, but read lock attempts may be extremely common. For this reason, fairness is less important than preventing readers from completely starving writers. An implementation that merely achieves the latter is said to have *writer priority*, an adequate property for the Symunix-2 kernel.

Implementation: Overview

The implementation for the target class of machines is somewhat different from the readers/writers algorithm of Gottlieb *et al.* [94]; we use an algorithmic improvement due to Freudenthal [85]. In either case, the heart of the algorithm is essentially the same:

- An integer is used to maintain a counter initialized to the maximum number of readers (in our case, `NUMPES`, the maximum number of processors).
- Readers treat the counter as a counting semaphore (§3.5.2/p56), i.e., they try `tdr1` until it succeeds.
- Writers try to decrement the counter by `NUMPES` with `tdr` until it succeeds.

---

[40] Examples: §3.7.6/p129, §3.7.6/p134, §3.7.6/p137, §6.1.1/p254, §6.1.1/p255, and §7.3.4/p294.

Unlike the readers/writers algorithm of Gottlieb *et al.* [94], an additional counter is not required to ensure writer priority. Here is the pseudo-code:

```
rlock(int *rw)
{
        if (fad(rw)<1) {
                fai(rw);
                while (!tdr1(rw))
                        continue;
        }
}
```

```
wlock(int *rw)
  {
        int old=faa(rw,-NUMPES);
        while (old<NUMPES) {
                if (old>0) {
                        // first writer
                          while (*rw!=0)
                             continue;
                     return;
                  }
                  // not first writer
                  faa(rw,NUMPES);
                  while (*rw<=0)
                          continue;
                  old=faa(rw,-NUMPES);
         }
  }
```

```
runlock(int *rw)
{
        fai(rw);
}
```

```
wunlock(int *rw)
  {
        faa(rw,NUMPES);
  }
```

Note that while the initial Fetch&Decrement of rlock is only an optimistic improvement over the simpler

```
while (!tdr1(rw))
        continue;
```

loop, the initial Fetch&Add of wlock is needed to maintain any semblance of writer priority. In fact, the writer priority is somewhat weaker in this algorithm than in the readers/writers algorithm of Gottlieb *et al.* [94], as it is possible for a reader to sneak in between the last two Fetch&Adds of wlock. Such a case is only possible when there are multiple writers contending, and it is still impossible for readers to starve all writers completely.

Implementation: Details The data structure contains the counter and a couple of histograms:

```
typedef struct {
     volatile int c;   // counter
#if BW_INSTRUMENT
     ehistgram *hgr, *hgw;
#endif
} bwrwlock;
```

Of course BWRW_INIT initializes c to NUMPES.

Readers treat the counter exactly the same as a counting semaphore (§3.5.2/p57): they obtain the lock when tdr1 succeeds. The first stage of bwrw_rlock, implemented as a macro to avoid unnecessary evaluation of the check function, looks like this:

```
#define bwrw_rlock(rw,f)                                \
      do {                                              \
           if (fad(&(rw)->c) > 0)                       \
                bw_instrument(qehgram((rw)->hgr,0));    \
           else                                         \
                _bwrw_rlock(rw,f);                      \
      } while(0)   /* require semicolon */
```

The parameters rw and f point to the lock and the check function, respectively. The second stage function, _bwrw_rlock, looks like this:

```
void
_bwrw_rlock (bwrwlock *rw, void f(void))
{
      vfai(&rw->c);  // undo failed first stage decrement
      BUSY_WAIT (tdr1(&rw->c), f, rw->hgr);
}
```

Writers obtain their lock when tdr(a,NUMPES) succeeds. The first stage looks like this:

```
#define bwrw_wlock(rw,f)                                \
      do {                                              \
           int _x = faa(&(rw)->c,-NUMPES);              \
           if (_x == NUMPES)                            \
                bw_instrument(qehgram((rw)->hgw,0));    \
           else                                         \
                _bwrw_wlock(rw,_x,f);                   \
      } while(0) /* require semicolon */
```

and the second stage looks like this:

```
void
_bwrw_wlock (bwrwlock *rw, int oldc, void f(void))
{
     register int iter = 0;
     unsigned int p = 0;
     while (1) {
          if (oldc == NUMPES)
               break;
          else if (oldc > 0) {
               // first writer, wait for readers to finish
               XBUSY_WAIT (rw->c == 0, f, rw->hgw, iter, p);
               break;  // done; lock obtained
          }
          else {
               // wait for another writer
               vfaa (&rw->c, NUMPES); // undo failed attempt
               XBUSY_WAIT (rw->c > 0, f, rw->hgw, iter, p);
               oldc = faa (&rw->c, -NUMPES); // try again
          }
     }
     XBUSY_FINAL(rw->hgw,iter,p);
}
```

Note the use of XBUSY_WAIT, as described in section 3.5.1 on page 52, to handle waiting and instrumentation.

In the absence of lock contention, either type of lock can be granted with only a single shared memory reference; the cost is no higher than a simple lock. One may ask, then, why support simple locks? The answer is portability and efficiency: machines without efficient Fetch&Add operations may have another, more efficient, algorithm for mutual exclusion, and we want to take advantage of the most efficient implementation possible for each machine.

Like the other operations capable of delaying, the upgrade operation is also suitable for a two-stage implementation. Here is the first stage:

```
int
bwrw_rtow (bwrwlock *rw)
{
     int oldc;
     oldc = faa (&rw->c, -(NUMPES-1));
     if (oldc < 0) {             // other writers ... give up
          vfaa (&rw->c, NUMPES-1);
          return 0;
     }
     if (oldc < NUMPES-1)        // other readers ...
          return _bwrw_rtow(rw); // succeed after delay
     bw_instrument(qehgram(rw->hgw,0));
     return 1;
}
```

The key step is to subtract NUMPES−1 from the counter; recall that it was already decremented by 1 when the read lock was obtained. The second stage simply waits for the other readers to release their own locks:

```
int
_bwrw_rtow (bwrwlock *rw)
{
      BUSY_WAIT (rw->c == 0, nullf, rw->hgw);
      return 1;
}
```

Reversing the process, downgrading from a write lock to a read lock, is accomplished by simply adding `NUMPES-1` to the counter; this is the `bwrw_wtor` operation. If there are no other writers waiting, this action allows other waiting readers to complete their second stage functions.

The `bwrw_tryrlock` operation is accomplished with a single `tdr1(&rw->c)` together with some debugging assertions. The corresponding function for writers, `bwrw_trywlock`, is similar: `tdr(&rw->c,NUMPES)`. As with the the corresponding semaphore operation, `bws_trywait` (§3.5.2/p58), we chose not to omit the initial test of the *Test-Decrement-Retest*; it is unlikely to create a performance bottleneck, and prevents livelock when used in a loop. Of course, livelock isn't a consideration for `bwrw_qrlock` and `bwrw_qwlock`, which never require waiting (as we saw in section 3.5.2 on page 55), so those functions consist of a single Fetch&Decrement and Fetch&Add, respectively.

3.5.5. Readers/Readers Locks

The name of this lock is inspired by readers/writers locks, but instead of enforcing either concurrent (reader) access or exclusive (writer) access, we provide two kinds of concurrent access (Edler [70]). Processors making accesses of type *X* may execute together, as may those making type *Y* accesses, but *X*s and *Y*s may never execute together.

Interface

The supported operations are listed in Table 10, on the next page. The flavor of supported operations is much the same as readers/writers locks (§3.5.4). The implementation may either be fair, or give priority to *X* lock requests.

Implementation: Overview

The current implementation for the target class of machines gives priority to *X* lock requests.[41] In the absence of dissimilar lock requests, no serialization is required.

The basic idea is to maintain separate counters of type *X* and *Y* readers. Since we allow for *X* priority, the algorithm is asymmetric:

[41] Fair implementations exist; a simple one can be built with the group lock mechanism (§3.5.6), as can an extended version to support *N* different mutually exclusive classes of access. Special case implementations are also possible (Freudenthal and Peze [87]).

Function	*Purpose*
`BWRR_INIT`	Initialize
`bwrr_destroy`	Un-initialize
`bwrr_xlock`	Obtain X lock
`bwrr_tryxlock`	Obtain X lock only if immediate
`bwrr_qxlock`	Obtain X lock (quick−guaranteed)
`bwrr_isxlocked`	Return X lock status
`bwrr_xunlock`	Release X lock
`bwrr_ylock`	Obtain Y lock
`bwrr_tryylock`	Obtain Y lock only if immediate
`bwrr_qylock`	Obtain Y lock (quick−guaranteed)
`bwrr_isylocked`	Return Y lock status
`bwrr_yunlock`	Release Y lock

Table 10: Busy-Waiting Readers/Readers Lock Functions.

```
struct rrlock {
        int x;
        int y;
};
xlock (struct rrlock *xy)
{
        vfai(&xy->x);
        while (xy->y != 0)
                continue;
}
```

```
ylock (struct rrlock *xy)
    {
  retry:
          while (xy->x != 0)
                  continue;
            vfai (&xy->y);
            if (xy->x != 0) {
                    vfad (&xy->y);
                    goto retry;
            }
    }
```

The actual code, given on page 66, includes additional details to support a two-stage implementation and a more optimistic approach to obtaining a *Y* lock.

Implementation: Details

The data structure is simple:

```
typedef struct {
     volatile int x, y;
#if BW_INSTRUMENT
     ehistgram *hgx, *hgy;
#endif
} bwrrlock;
```

Within the `bwrrlock` structure, the number of *X* and *Y* locks held (or requested) is maintained in the `x` and `y` fields, respectively. Because this implementation always gives priority to *X* lockers, they need only declare themselves, by incrementing `x`, and wait for any *Y* lockers to unlock. *Y* lockers, in contrast, must first wait for all *X* lockers to unlock before even declaring themselves by incrementing `y`; in addition they must yield to any *X* lockers (decrement `y` and start over) whenever there is a race with an *X* locker. This approach is similar to TDR, and to Freudenthal and Gottlieb's Fetch&Increment-based readers/writers algorithm [86].

Initialization and destruction are trivial: `x` and `y` are initialized to zero by `BWRR_INIT`, and `bwrr_destroy` is an empty function. As usual for Symunix-2, the operations with potential delays are implemented in two stages.

```
#define bwrr_xlock(rr,f)  /* first stage X lock */      \
     do {                                               \
          vfai(&(rr)->x);                               \
          if ((rr)->y == 0)                             \
               bw_instrument(qehgram((rr)->hgx,0));     \
          else                                          \
               _bwrr_xlock(rr,f);                       \
     } while (0)           /* require semicolon */
```

In the absence of any *Y* lock requests, an *X* request can be satisfied with only two shared memory accesses (`vfai` and testing `(rr)->y == 0`), not counting instrumentation costs. Any real difficulties are handled by the second stage function:

```
void                        // second stage X lock
_bwrr_xlock (bwrrlock *rr, void f(void))
{
     BUSY_WAIT (rr->y==0, f, rr->hgx);
}
```

The complementary functions for `Y` locks are more optimistic, since they treat the initial attempt differently from subsequent attempts. The first stage macro `bwrr_ylock` is identical to `bwrr_xlock`, except for the swapping of "`x`" and "`y`":

```
#define bwrr_ylock(rr,f)  /* first stage Y lock */      \
     do {                                               \
          vfai(&(rr)->y);                               \
          if ((rr)->x == 0)                             \
               bw_instrument(qehgram((rr)->hgy,0));     \
          else                                          \
               _bwrr_ylock(rr,f);                       \
     } while (0)           /* require semicolon */
```

If this fully symmetrical relationship between *X* and *Y* lock requests were allowed to continue, a livelock situation could easily develop. The problem is resolved in the second stage:

```
void                        // second stage Y lock
_bwrr_ylock (bwrrlock *rr, void f(void))
{
#if BW_INSTRUMENT
     int iter = 0;
     ehistgram *hg = rr->hgy;
     unsigned p = 0;
#endif
     do {
          vfad(&rr->y);      // undo failed attempt
          XBUSY_WAIT (rr->x==0, f, hg, iter, p);
          vfai(&rr->y);
     } while (rr->x);
     XBUSY_FINAL(hg,iter,p);
}
```

Note the use of XBUSY_WAIT to perform instrumentation across potentially many delays, as described in section 3.5.1 on page 52.

Remaining functions of interest are rather straightforward:

```
void
bwrr_xunlock (bwrrlock *rr)
{
     vfad(&rr->x);
}

int
bwrr_tryxlock (bwrrlock *rr)
{
     vfai(&rr->x);
     if (rr->y == 0)
          return 1;
     vfad(&rr->x);
     return 0;
}

void
bwrr_qxlock (bwrrlock *rr)
{
     vfai(&rr->x);
     assert (rr->y == 0);
}
```

```
void
bwrr_yunlock (bwrrlock *rr)
{
     vfad(&rr->y);
}

int
bwrr_tryylock (bwrrlock *rr)
{
     if (rr->x)
          return 0;
     vfai(&rr->y);
     if (rr->x == 0)
          return 1;
     vfad(&rr->y);
     return 0;
}

void
bwrr_qylock (bwrrlock *rr)
{
     vfai(&rr->y);
     assert (rr->x == 0);
}
```

```
int
bwrr_isxlocked (bwrrlock *rr)
{
     return rr->y;
}
```

```
int
bwrr_isylocked (bwrrlock *rr)
{
     return rr->x;
}
```

3.5.6. Group Locks

Dimitrovsky's group lock [64, 65, 66] isn't really a "lock" in the same sense as a binary semaphore, or readers/writers lock. The primary operations are named *lock* and *unlock*, but the number of processors allowed access, and the manner of their accesses, are not controlled the same way as with other kinds of locks. The group *lock* operation delays the caller, if necessary, until a new group can be formed. A group is a collection of processors that have not yet executed the *unlock* operation. Only one group at a time is allowed to execute; any other processor invoking the *lock* operation must wait to join a later group after the current one completes.[42] In addition, the group lock provides operations for *barrier synchronization* of all the processors in a group; these are said to divide the group lock into *phases*.

Interface

The supported operations[43] are listed in Table 11, below. `BWG_INIT` is a macro, for the same

Function	*Purpose*
`BWG_INIT`	Initialize
`bwg_destroy`	Un-initialize
`bwg_lock`	Join group
`bwg_sync`	Barrier within group
`bwg_qsync`	Barrier (quick−simplified)
`bwg_fsync1`	Barrier (fuzzy−part 1)
`bwg_fsync2`	Barrier (fuzzy−part 2)
`bwg_unlock`	Leave group
`bwg_size`	Return group size
`bwg_relock`	Leave and rejoin next group

Table 11: Busy-Waiting Group Lock Functions.

[42] This is somewhat like the N-Step SCAN disk scheduling algorithm [192], in that processors (disk requests) arriving after a group has formed (sweep has begun) are forced to wait for the next group (sweep).

[43] There have been several minor variations in the definition of the basic group lock operations. The differences concern the way group members may obtain *index values* within a phase. In Dimtrovsky's original version [64], no special provision was made, but group members could provide their own indices by using Fetch&Add. In a subsequent paper [66], he defined the *gindex* operation to provide indices in a more general manner. Our version provides indices as a side effect of group lock entry and barrier operations. In addition, we added several barrier variants and the *relock* operation.

reason as `BWS_INIT` (see section 3.5.2 on page 54). The `bwg_lock` and `bwg_sync` operations return a unique number in {0, ..., (group size)-1}; this number is the *index* of the processor within the subsequent *phase* of the group lock. The implementation must allow some processors to call `bwg_unlock` while others may be calling `bwg_sync`, but there is another operation, `bwg_qsync` that isn't required to support this *early exit* property, and may actually be slightly cheaper than `bwg_sync`. To round out the set of barrier operations, a *fuzzy* barrier is provided (fuzzy barriers were introduced by Gupta [97]). A fuzzy barrier is accomplished by making two separate calls: no processor may proceed beyond `bwg_fsync2` before all others in the group have called `bwg_fsync1`.[44] An example of using these functions can be found in section 3.7.4 on page 118.

The number of processors in the group may be determined at any time by calling `bwg_size`, but this number is indeterminate if any processors are calling `bwg_unlock` concurrently.

An additional operation, `bwg_relock`, allows a processor to leave the active group and join the very next one, with no possibility of intervening groups. An example of such usage will be given in section 3.7.3 on page 99; the goal is to reduce the average time spent in `bwg_lock`.

Implementation: Overview

The algorithm used in the current implementation is based on that of Freudenthal and Gottlieb [86]. There are two parts to the algorithm: group formation and barrier synchronization. Group formation is based on three shared variables, `ticket`, `group`, and `exit`, all initialized to 0. At the beginning of `bwg_lock`, a ticket is issued to each processor by incrementing `ticket` with Fetch&Increment. As a group is being formed, the `group` variable is the ticket number of the *group leader*, the processor responsible for determining the maximum ticket value admitted to the group. All processors with tickets less than `group` have already been admitted to previous groups. When processors leave their group (via `bwg_unlock`), the `exit` variable is incremented; this allows the next group leader to know when the previous group has finished (when `exit` equals `group`, which also equals the group leader's ticket). When this happens, the group leader copies `ticket` to `group`, thus defining the size of the group and the next group leader. All non-leaders simply wait until their ticket is less than `group`.

In section 7.3.5 on page 307, we will describe another group lock algorithm with a different approach to group formation.

The barrier portion of the algorithm is based on the size of the group, determined by the difference `(group-exit)`, and two variables, `barcount` and `barphase`, the latter restricted to values 0 and 1. Before any processor reaches a barrier, `barcount` is 0. Each processor reaching a barrier makes a private copy of `barphase`, and increments `barcount` with Fetch&Increment, getting a number that will eventually be returned to the caller as the *index* of the processor for the next phase of the group lock. Meanwhile, this index is also used to identify the last processor reaching the barrier, which occurs when the index is one

[44] We take a somewhat different approach for a context-switching group lock in section 7.3.5 on page 304.

less than the current group size. The last processor to reach the barrier resets `barcount` to 0, making it ready for the next barrier, and complements `barphase`; the other processors simply wait until `barphase` differs from their private copy.

Implementation: Details

Here is the data structure:

```
typedef struct {
      volatile unsigned int ticket,
                            group,
                            exit,
                            barcount,
                            barphase;
#ifdef BW_INSTRUMENT
      ehistgram *hg;
#endif
} bwglock;
```

The first three variables are unsigned to avoid overflow problems. When they overflow, the ANSI C standard [5] guarantees the result will be computed by discarding high order bits, and no exception will be raised. Here is the first stage code for the lock operation:

```
int
bwg_lock (bwglock *g)
{
      unsigned int myt = ufai(&g->ticket);
      int x = _bwg_trylock(g,myt);
      if (x < 0)
           x = _bwg_lock(g,myt);
      else
           bw_instrument(qehgram(g->hg,0));
      return x;
}
```

Unlike the other busy-waiting mechanisms in section 3.5 on page 49, a check function is not provided. This is because the processor is committed to a group by the initial Fetch&Increment of `g->ticket`; accepting an extra delay while waiting to begin group execution is just as harmful to group barrier latency as at any other point before leaving the group. Since the primary use of check functions is to allow interrupts, we simply omitted the feature in this case.

The basic test condition for completion of `bwg_lock` is encapsulated in `_bwg_trylock`, an inline function that is strictly "internal" to the implementation. If the result of `_bwg_trylock` is non-negative, the processor has successfully entered a group, and the result is the processor's index for the first phase. The `else` statement is responsible only for instrumentation in the case of no delay. In the case that `_bwg_trylock` failed, the second stage function, `_bwg_lock`, is called:

```
int
_bwg_lock (bwglock *g, unsigned int myt)
{
      int x;
      BUSY_WAIT ((x=_bwg_trylock(g,myt)) >= 0, nullf, g->hg);
      return x;
}
```

The parameter `myt` is the all-important ticket of the processor. As soon as `_bwg_trylock` returns a non-negative result, we are done.

The condition for completion of `bwg_lock` is that all members of the previous group have called `bwg_unlock`: this is tested by `_bwg_trylock`:

```
int
_bwg_trylock (bwglock *g, unsigned int myt)
{
      unsigned int myg = g->group;

      if (myg == myt) {
            if (g->exit != myt)
                  return -1;
            else {
                  myg = g->ticket;
                  g->group = myg;
                  return (int)(myg - myt - 1);
            }
      }
      else if (udiff(myg,myt) > 0)
            return (int)(myg - myt - 1);
      else
            return -1;
}
```

The initial test determines if the caller is a group leader or not. If it is, we can proceed if every member of the previous group has incremented `exit`; we then set `group` to define the group size and the following group leader. If the caller isn't a group leader, it may proceed when the group leader advances `group` to exceed the caller's ticket. This latter test is performed by the function `udiff` (for "unsigned difference"), which performs subtraction modulo $2^{wordsize}$, returning the difference as a signed integer. The result produced by `udiff` is correct even if its parameters have already "wrapped around", providing that $-2^{wordsize-1} < myg - myt < 2^{wordsize-1}$, where *myg* and *myt* are the values of `myg` and `myt` that would result if the algorithm were run with arbitrary precision integers. For the group lock algorithm, this simply requires that the largest possible group size $< 2^{wordsize-1}$. The following suffices for conventional implementations of C on 2's complement machines:[45]

[45] This solution is not "strictly conforming" to the ANSI C standard [5].

```
#define udiff(a,b) ((int)((a)-(b)))
```

The two-stage implementation strategy doesn't work as well with group lock as it does with regular kinds of locks. As can be seen from the description so far, it is unlikely that a processor can enter the group without calling the second stage function, except for the first one to enter when the lock was previously "open" (`group == ticket == exit`). The two-stage approach is probably still worthwhile for uses where the group lock is often available immediately and a group size of 1 is not uncommon. Of course, the caller can recognize when the group size is 1, and use a purely serial algorithm without barriers to reduce overhead within the semi-critical section controlled by the group lock (see section 3.7.3 on page 101 for an example).

The barrier operation comes in several flavors, depending on whether or not *early exit* must be supported and on the choice of normal or fuzzy barriers.

```
int  // stage 1, ordinary group barrier
bwg_sync (bwglock *g)
{
     unsigned int gsize = g->group - g->exit;
     if (gsize == 1) {
          bw_instrument(qehgram(g->hg,0));
          return 0;
     }
     // not worth inlining check for barrier reached (?)
     return _bwg_sync (g, gsize);
}
```

A two-stage implementation is employed simply to check for and optimize the case of a group of only one processor. The second stage function contains the entire barrier algorithm:

```
int  // stage 2, ordinary group barrier
_bwg_sync (bwglock *g, unsigned int gsize)
{
     unsigned int phase = g->barphase;
     unsigned int count = ufai(&g->barcount);

     FBUSY_WAIT (_bwg_trysync(g,gsize,count,phase) ||
                 ((gsize=g->group-g->exit),0), nullf, g->hg);
     return count;
}
```

The group size must be repeatedly recalculated to allow for early exit; this can be avoided in `bwg_qsync`, which, for the sake of brevity, we do not present. The barrier completion test is encapsulated into an inline function, `_bwg_trysync`, so that it may easily be incorporated into the fuzzy barrier as well; here it is:

```
int
_bwg_trysync (bwglock *g, unsigned int gsize,
              unsigned int count, unsigned int phase)
{
      if (gsize == count + 1) {
            // last one to reach barrier
            g->barcount = 0;
            g->barphase = !phase;
            return 1;
      }
      if (phase != g->barphase)
            return 1;
      return 0;
}
```

Note that `count` and `phase` are only set at the beginning of the barrier.

Fuzzy barriers are accomplished by a pair of functions, `bwg_fsync1` and `bwg_fsync2`. The first part of the fuzzy barrier needs to communicate some state to the second part; we abstract this as a "cookie", which must be allocated by the caller. The cookie is an `int`, and we use it to encode both the group size and phase (`gsize` from `bwg_sync` and `phase` from `_bwg_sync`). We omit the details from this presentation for the sake of brevity.

Group exit and size operations are short enough to be implemented with macros:

```
#define bwg_unlock(g) vufai(&(g)->exit)
#define bwg_size(g)    ((g)->group - (g)->exit)
```

Of course the true group size is indeterminate if `bwg_unlock` executes concurrently with `bwg_size`.

There are times, such as in the example in section 3.7.3 on page 99, when performance can be improved by overlapping final computations of one group with the early computations of the following group. This effect can be achieved with the `bwg_relock` function, which behaves as a combined `bwg_unlock` and `bwg_lock`, but guarantees no intervening groups:

```
int
bwg_relock (bwglock *g)
{
      unsigned int myt = ufai(&g->ticket);
      vufai(&g->exit);
      int x = _bwg_trylock(g,myt);
      if (x < 0)
            x = _bwg_lock(g,myt);
      else
            bw_instrument(qehgram(g->hg,0));
      return x;
}
```

This is the same as `bwg_unlock` and `bwg_lock`, except we have postponed the action of `bwg_unlock` until after the ticket for `bwg_lock` is obtained.

3.6. Interprocessor Interrupts

The need for interprocessor interrupts can arise in many situations, including processor scheduling, device management, TLB management, signal delivery, etc. Symunix-1 runs only on the NYU Ultra-2 prototype (§8.1), which lacks a direct hardware interprocessor interrupt mechanism, so it handles such situations in the worst case by relying on a regular 16Hz clock interrupt on each processor. One of these situations arises in support of asynchronous serial communication ports, which are directly connected to processors. Continuous input and output on these ports is interrupt driven, but a process running on the wrong processor can't physically access the device; this poses a problem especially for starting an output stream. The Symunix-1 solution is to use an extended version of the traditional UNIX kernel `timeout` mechanism, which provides a per-processor time-ordered list of functions to call in the future. By specifying a time that is 0 clock ticks in the future, a function can be called on another processor as soon as possible (§6.3/p265).

In Symunix-2, which is intended to be portable to a variety of machines, we replace this peculiar use of the `timeout` facility with a new abstraction called *Asynchronous Procedure Calls* (*APCs*). We also introduce a new abstraction, *Random BroadCasts* (*RBCs*), which extends the APC mechanism to act on a specified number of randomly selected processors, thus providing a low overhead way of parallelizing certain fine granularity operations in the kernel.

APCs are not like *Remote Procedure Calls*, since the calls are asynchronous; the initiator does not wait for the call to complete, nor is any mechanism built-in for a result to be passed back. While APCs and RBCs are suitable for many purposes within an operating system kernel, they are not intended as a general-purpose parallel or distributed computing paradigm.

APCs are abstract in two ways:

(1) The implementation hides hardware details. The physical mechanism for interprocessor interrupts is machine-dependent.

(2) A hidden dispatcher is provided to allow each interprocessor interrupt to call a different procedure. This is important for kernel modularity, since APCs are a central facility used by totally unrelated parts of the kernel.

RBCs share these two abstraction properties, and add two more:

(3) The selection of target processors is hidden as a machine detail. Depending on hardware support, the choice may be static or dynamic, and may (but need not) vary from call to call.

(4) The actual number of interrupts generated by a single RBC call depends on overhead costs and the number of physical processors running in the machine. The programmer specifies only the total amount of work to be done and a rough estimate of the granularity (see page 76).

APC Interface

The basic interface supplied by the APC facility consists of two functions, corresponding to one or two interrupt levels:

`int apc(int penum, void (*f)(genarg_t), genarg_t arg)`

The function `f` will be called with generic argument `arg` on the processor indicated by `penum` as soon as possible. (`genarg_t` is a typedef for a union of common scalar types.) On the target processor, the call will be made in the context of an interrupt handler, and may be masked by calling `splapc()` (§3.4). The return value indicates success or failure, to be discussed further below.

`int softapc(int penum, void (*f)(genarg_t), genarg_t arg)`
: This is the same as `apc`, but the call is executed in the context of a soft interrupt handler on the target processor, and may be masked by calling `splsapc()`. Recall that the priority of this interrupt must be no higher than any hard interrupt, and that the hard APC interrupt priority must be no lower priority than any soft interrupt (§3.4).

In both cases, if the target processor is the calling processor and the appropriate interrupt is unmasked, the call is performed at once and without incurring interrupt overhead. There is no built-in mechanism provided for the caller to synchronize with an asynchronous procedure call or to obtain status information from it; any such synchronization must be programmed directly.

On machines with hardware support for only a single interprocessor interrupt, it may be best used for `apc`, leaving `softapc` to be implemented indirectly by using `apc` to call a procedure that calls `setsoftapc` to send the soft interrupt, locally, to the target processor.

The functions `apc` and `softapc` dynamically allocate a structure to record the (`f`, `arg`) pair and keep track of each pending call. They also arrange for automatic cleanup of the structure, making them easy to use but subject to failure if the dynamic allocation fails (this is the significance of the return value, a boolean). Another disadvantage of this automatic structure management is that it involves some additional overhead in cases where a procedure is to be called asynchronously many times. To remedy both of these difficulties, an alternate interface is provided to allow explicit structure management:

`apc_pair *apcregister(apc_pair *, void (*f)(genarg_t), genarg_t arg)`
: Initialize an `apc_pair` structure for calling `f(arg)` asynchronously. The caller may allocate the `apc_pair` structure itself, e.g. as part of some larger data structure, or it may pass `NULL` to let `apcregister` allocate one dynamically, returning the resulting pointer (or `NULL`, if the allocation fails). A call to `apcregister` can't fail if the caller has allocated the structure beforehand.

`void apcreregister(apc_pair *, void (*f)(genarg_t), genarg_t arg)`
: Given an `apc_pair` structure that has already been initialized by `apcregister`, this allows changing just the (`f`, `arg`) pair, without reinitializing other data internal to the structure.

`void apcunregister(apc_pair *)`
: Uninitialize the indicated `apc_pair` structure, and, if it was dynamically allocated by passing `NULL` to `apcregister`, deallocate it.[46]

`void r_apc(int penum, apc_pair *)`
: Like `apc`, but using a preregistered (`f`, `arg`) pair.

[46] There is a marker inside each `apc_pair` to indicate if it was allocated by `apcregister` or not.

`void r_softapc(int penum, apc_pair *)`

Like `softapc`, but using a preregistered (`f`, `arg`) pair.

RBC Interface

Random broadcasts are very similar to asynchronous procedure calls, but cause a procedure call to be executed on a specified number of processors, rather than on a specific one. The implementation has a fair amount of latitude in how this is done, which is important since most current machines don't provide direct hardware support (see section 3.8 on page 142). Only a single interrupt level is assumed for random broadcasts (a soft one), although extending the system to include another level would not be difficult. The basic interface is modeled after that of `softapc`:

`int softrbc(int howmany, void (*f)(genarg_t, int), genarg_t arg, int chunksize)`

Cause the procedure `f` to be executed on an appropriate number of processors as soon as possible. The processors may be randomly selected, and may or may not include the calling processor. Each target processor calls `f(arg,`*n*`)`, where the sum of all the parameters *n* is `howmany`. The number of target processors actually used isn't directly controlled by the caller of `softrbc`, but the `chunksize` parameter is supposed to reflect the relative cost of `f`, and generally indicates that ⌈`howmany/chunksize`⌉ processors can be productively used.

As with APCs, there is an alternate interface for RBCs to provide explicit control over data structure allocation and initialization:

`rbc_pair *rbcregister(rbc_pair *, void f(genarg_t, int), genarg_t arg, int smallthresh, int chunksize)`

Like `apcregister`, but the structure type operated upon is `rbc_pair` rather than `apc_pair`. Actually, in the case of RBCs, the structure describes four items rather than a pair: the procedure, `f`, the argument, `arg`, and two numbers to control the overhead, `smallthresh` and `chunksize`. The role of `chunksize` has already been described for `softrbc`, but `smallthresh` is a parameter that is only available through the alternate interface. As its name suggests, it specifies the threshold for "small" RBC requests. A call to `r_softrbc` (p77) with `howmany` less than `smallthresh` will immediately be fully executed on the calling processor, provided that the soft RBC interrupt is not masked.

`void rbcreregister(rbc_pair *, void f(genarg_t, int), genarg_t arg, int smallthresh, int chunksize)`

Like `apcreregister`, `rbcreregister` allows the caller to change only the `f`, `arg`, `smallthresh`, and `chunksize` fields of an already initialized `rbc_pair` structure.

`void rbcretune(rbc_pair *, int smallthresh, int chunksize)`

As its name suggests, this is like `rbcreregister`, but only allows one to *retune* the parameters `smallthresh` and `chunksize`, while leaving the procedure and argument alone.

`void rbcunregister(rbc_pair *)`

Uninitialize the indicated `rbc_pair` structure, and, if it was dynamically allocated by passing `NULL` to `rbcregister`, deallocate it.

`void r_softrbc(int howmany, rbc_pair *)`
: Like `softrbc`, but using a preregistered `rbc_pair` structure.

Joint APC/RBC Modular Implementation

The implementation of the APC and RBC facilities is largely machine-independent, although the key component that actually requests or emulates an interprocessor interrupt is machine-dependent. “Hooks” are provided in the form of calls to machine-dependent functions from the machine-independent portion of the implementation:

`void md_apc(int penum)`
: Send, or emulate, an interprocessor interrupt to the processor indicated by `penum`.

`void md_softapc(int penum)`
: Like `md_apc`, but for soft APC interrupts.

`void md_softrbc(int numpes)`
: Send, or emulate, soft RBC interrupts to `numpes` unspecified processors.

In addition, machine-dependent code must be provided to handle the interrupts (or simulated interrupts) thus generated, and call the machine-independent *worker* routines:

`void doapc(void)`
: This parameterless function is called, with hard APC interrupts masked, to do all pending hard APC work for the executing processor.

`void dosoftapc(void)`
: This is like `doapc`, but for soft APCs.

`void dosoftrbc(int nreq)`
: This function is called, with soft RBC interrupts masked, to do all pending soft RBC work for the executing processor. But the nature of RBCs and the desire to avoid serialization create some difficulty in determining just how much work is pending for each processor. The parameter `nreq` is supposed to be the number of RBC requests directed at the executing processor since the last call to `dosoftrbc`, but typical hardware doesn’t provide a way to know this number. The caller of `dosoftrbc` is machine-dependent, and so can provide an approximation if the exact number is not maintained by hardware.[47]

3.7. List Structures

Since the early work of Gottlieb, *et al.* [94], and Rudolph [174], much software attention for NYU Ultracomputer-style machines has focused on queues and other list-like structures. The tradition continues in the current work with some minor improvements to previous algorithms (§3.7.1) and some new algorithms for new purposes (§3.7.3, §3.7.4, §3.7.5).

---

[47] All the hardware needs to do is count the received rbc interrupts, and provide an atomic method of reading and clearing the counter.

3.7.1. Ordinary Lists

No single algorithm for parallel list access will ever be truly universal in practice, even for a single machine. There are just too many important algorithmic properties to optimize, and they cannot be ignored without sacrificing significant functionality, performance, or scalability.[48] The following paragraphs describe the most significant issues that have arisen during our experience.

Ordering Discipline.

Probably the most obvious ordering discipline to consider is FIFO, but in our experience, two others are more valuable: completely unordered, and starvation free. The former is especially well suited for free lists of available objects, while the latter is good for most other purposes. With a starvation free discipline, no item can languish indefinitely on the list while an arbitrary number of other items are inserted and deleted. A probabilistic solution is generally adequate, where the probability of an item remaining on the list approaches zero as the number of other items deleted increases.

In many cases, an unordered list can be more cheaply implemented than either of the others. Often FIFO is preferable if it is no more expensive, or if guaranteed fairness is important (as, perhaps, in a real-time system).

Contention-Free Execution Time.

An important list algorithm performance measure is the cost of a single insert or delete, without any competing concurrent accesses. Some solutions achieve the ideal of a small constant time, but others require time that is logarithmic in the capacity or current size, or have different best, average, and worst case costs. There are usually tradeoffs between minimum execution time and other factors, such as worst-case time, concurrency, memory usage, etc.

Concurrency.

In general, concurrency control costs something. As an extreme example, the cheapest possible list algorithms are generally not safe for concurrent execution at all; such a serially-accessible list might be suitable for use on a uniprocessor, or in inherently serial situations (such as within a critical section), or when the list is private to a process or processor.[49] When concurrent access is a possibility, the shortest contention-free execution time is sometimes achieved by using a simple lock in conjunction with a serially-accessible list. This time-honored approach is, of course, not highly-parallel, but may still be appropriate if contention is expected to be very rare. Other techniques can be used to eliminate serialization, with varying costs in terms of expected execution time, memory usage, etc.; there are many possibilities.

Capacity.

Generally, lists with a fixed capacity have shorter access times than those without such a limit, but they usually also require an amount of memory that is proportional

---

[48] The same is, of course, also true for serial list algorithms and serial computers.

[49] Serially-accessible lists are acceptable for per-processor use only when preemptive scheduling is inapplicable or disabled.

to their capacity rather than their current size. Some data structures have no fixed capacity, but suffer worse expected execution times as the number of items grows beyond some point (e.g., an algorithm based on a fixed size hash table). Serialization may also increase as a list approaches its full or empty condition.

For our purposes, we generally prefer an insert or delete operation to return a failure indication rather than delaying for a full or empty list, respectively, but the opposite approach may be more appropriate in some environments.

Memory Usage.

The ideal list requires a constant amount of memory, independent of capacity or number of processors, plus a constant amount of memory for each item in the system. This is important because, in many cases, the number of lists and items are both proportional to the machine size (number of processors). Greater per-list or per-item memory requirements can quickly lead to an overall explosion of memory usage as the system is moved to larger machines (e.g., quadratic growth, if both sizes and number of lists are proportional to the machine size). Unfortunately, memory-frugal algorithms sometimes pay a premium in execution time compared to less scalable alternatives. It is therefore sensible to identify the situations where the number of lists grows sub-linearly in the machine size. Typical examples are free lists; often there is one of these for each of several kinds of objects in the system, not one per processor.[50] Such lists can “get away” with non-constant per-list memory requirements.

Multiplicities.

An insert with multiplicity m means that an item is inserted onto a list only once, but tagged in such a way that m consecutive deletes will return a pointer to it before it is actually removed from the list’s data structure. This usage is explained further in section 4.6.9. The primary difficulty in designing an algorithm for a list with multiplicities is arranging for high concurrency of deletes. In particular, we know of no algorithm that handles a mixture of multiplicities well, and is also competitive with non-multiplicity algorithms (for multiplicity 1).

Interior Removal.

The idea of interior removal is to remove a given item from a given list; the operation fails if the item is not, in fact, on the list. Algorithms that support interior removal are tricky because they must be able to arbitrate contention between an interior removal and an ordinary delete attempting to operate on the same item. Additional cost is generally required.

In our experience, it is usually valuable to keep track of *holes*, placeholders for items that have been removed. Thus a delete may return any of three kinds of value: a *list empty* failure indicator, a valid pointer to an item, or a special value to indicate a hole. Depending on the ordering discipline chosen, the holes may or may not need to maintain their position. Making holes visible allows the caller to assume the

---

[50] Sometimes per-processor free lists can be used to good advantage, however. A reasonable approach may be to combine the use of a small-capacity, private list per-processor with a single, highly parallel list.

number of items on the list is unaffected by interior removal. The importance of this can be seen in the readers/writers unlocking routines given in section 4.5.2.

Parallel-access queues with holes were first introduced by Wilson [205]. It is important to realize that a hole left behind by an interior removal operation will increase the cost of an ordinary delete, when it is eventually encountered. In the most extreme case, where all items leave the list via removal and none via deletion, the number of holes will grow without bound.[51]

Presence Test.
Testing to see if a given item is, at this instant, on a given list is occasionally useful. Note that this is quite different from any kind of search; the goal is not to locate anything, but to return a boolean result.

Item Count.
Providing the instantaneous number of items on a list is never an absolutely necessary part of a list algorithm, because it can be simulated by incrementing a count after each insert and decrementing it before each delete. Many algorithms maintain the count for internal purposes, however, in which case it costs “nothing” to support.

Deletion from Empty list.
When a delete operation encounters an empty list, there are basically two possible actions: return an indication of failure or block. We generally take the former approach, since it is easy to use a counting semaphore to achieve the latter, but some additional convenience and efficiency could be gained by taking a more integrated approach.

Architectural Support.
Many parallel architectures include special features that can be advantageous in crafting efficient list algorithms. Some examples include Test&Set, Fetch&Add, Compare&Swap, and full/empty bits; there are many others. Equally important may be implementation characteristics that are largely considered to be transparent, such as cache organization and coherence strategy and the ability (or lack thereof) to combine concurrent memory requests directed at the same location.

Interface

In the context of an operating system kernel, the potential number of practical list algorithms tends to favor a modular approach, in which the multitude of alternative algorithms and their data structures can be hidden behind generic operations on list and item types that are selectable at compile time as configuration choices. The item types contain no data; rather they are included as members in other structures of interest.[52] To allow objects to move from list to list even when the lists are different types, the list item types are designed to be unioned together in appropriate combinations, as needed for each object.

---

[51] If the holes occupy no real space, but are merely counted, this will “only” lead to integer overflow.

[52] In an object-oriented system, the real structures of interest would be derived from the list item types. In a plain ANSI C environment, the `offsetof` construct can be used to get a pointer to the enclosing structure from an item pointer.

Reinitialization functions are defined for use when moving from one list type to another. This allows an object to move among lists of different types with a per-object space overhead equal to the maximum of the item sizes used.

With this approach, the system may be tuned for a particular workload or machine configuration by choosing the best algorithm for each situation. Disregarding initialization issues, Table 12, below, gives the complete set of generic operations on ordinary lists used within the Symunix-2 kernel. The set of operations used is intentionally small, to provide the greatest implementation flexibility possible. In particular, note the absence of search or traversal operations (but see sections 3.7.3 and 3.7.6). In addition, not all operations are required for every list. In particular,

- We do not use both *put* and *mput* on the same list.
- We use the operations of *remove*, *puthole*, and *presence test* on a very small number of lists.

To get the best performance, one must choose, for each list, the least powerful semantics that meet the requirements.

Implementation

As a result of our highly tailorizable approach, very good solutions are possible for the target machine class as well as for other, more traditional classes, and there is ample room to experiment with alternatives. Table 13, on the next page, indicates the algorithms chosen initially for the Ultra-3 prototype, typical of our target class. Table 14, on page 83, shows the performance and memory usage of the same algorithms. Similar experimental results for related algorithms have been obtained by Wood [207].

Three algorithms listed in Table 13 have $O(P)$ memory consumption per list on a P processor machine: `pool`, `dafifo`, and `mqueue`. They are used in situations where the number of such lists is (nearly) independent of P; otherwise they would be unacceptable for our target class of machines:

- `pools` are used almost exclusively as lists of unallocated objects, e.g., process structures, file structures, etc. The number of different object types is constant, possibly varying only between different operating system versions.
- The primary use of `dafifos` is for waiting lists of interruptible counting semaphores and readers of readers/writers locks (§4.5.2). Traditional UNIX semantics only require

Operation	*Description*
put	Insert item on list
get	Delete and return ''next'' item from list
count	Return number of items on list
mput	Insert with multiplicity
remove	Remove indicated item from list
puthole	Insert a place marker (a ''hole'')
presence test	Determine if item is on list

Table 12: Generic List Operations.

List Description	*List Type*	*Item Type*	*Applicability*
Linked lists	`llist`	`llitem`	binary semaphore wait list r/w wait list for writers
Deluxe linked lists	`dllist`	`dllitem`	same as `llist`, but supports interior removal for interruptible synchronization
Un-ordered lists	`pool`	`poolitem`	free lists
Almost FIFO lists	`afifo`	`afitem`	counting semaphore wait list r/w wait list for readers event wait list
Deluxe almost FIFO lists	`dafifo`	`dafitem`	same as `afifo`, but supports interior removal for interruptible synchronization ordinary ready list
Almost FIFO multi-queues	`mqueue`	`mqitem`	ready list for spawning processes for awakened readers for awakened event waiters

Table 13: Ordinary List Types for Ultra-3.

interruptible semantics for system calls that access “slow” devices, like terminals; for computationally intensive workloads we don’t expect the number of such devices to grow with the size of the machine.[53]

- `mqueues` are useful primarily for the ready list. (§4.6.9).

The algorithms themselves are derivatives of previously developed ones; `llist` and `dllist` are implemented with fairly ordinary linked lists, singly- and doubly-linked, respectively; `afifo` is based on Dimitrovsky’s hash table queue algorithm [63], because of its simplicity, good average performance, and low space usage;[54] circular array structures are used for `pool`, `dafifo`, and `mqueue`, because they have advantages of simplicity, lower overhead, and they handle interior removals well (see Wilson [205] for variations of the circular array list theme).

---

[53] A few other system calls are also interruptible, primarily *wait* and *pause*. These do not require `dafifo`s.

[54] But only if a sufficiently large hash table can be statically allocated. See Wood [207] for a *dynamic* hash table queue algorithm that may be more appropriate; better alternatives will probably be discovered over time. Note also that Dimitrovsky’s hash table queue algorithm requires generation of queue identifiers; we handle this with a variation of the set algorithm to be presented in section 3.7.5 on page 124.

List	No Contention †			Parallel ‡			Memory *
	Put	Get	Remove	Put	Get	Remove	
`llist`	7	7	n/a	$O(N)$	$O(N)$	n/a	$16L + 4I$
`dllist`	9	11	10	$O(N)$	$O(N)$	$O(N)$	$20L + 12I$
`pool`	8	9	n/a	$O(1)$	$O(1)$	n/a	$(12 + 12P)L + 4I$
`afifo`	11	11	n/a	$O(1)$	$O(1)$	n/a	$12 + 16L + 8I + 16H$
`dafifo`	12	14	12	$O(1)$	$O(1)$	$O(1)$	$(12 + 24P)L + 16I$
`mqueue`	12	7,19•	n/a	$O(1)$	$O(1), O(N)$•	n/a	$(16 + 20P)L + 16I$

† Minimum number of shared memory references required for the indicated list operation. It is achieved when the operation succeeds and there is no contention.
‡ Time for 1 of N concurrent processors operating on the same list, assuming the list is neither full nor empty and all processors run at the same speed on a CRCW PRAM.
* Memory requirement in bytes for I items distributed across L lists on a machine with P processors. H specifies the number of buckets in a hash table, and is proportional to the maximum expected value of I.
• Time for *get* depends on average multiplicity; values given represent the extremes, assuming perfect combining.

Table 14: Ultra-3 Asymptotic List Performance.

A uniprocessor can take advantage of the tailorizability to use simple linked list algorithms (perhaps only one) in all cases. Small multiprocessors might take a similar approach by adding simple locks to the algorithm(s). Machines with hardware support for compare-and-swap or load-linked and store-conditional instructions will presumably benefit from list algorithms based on them. Very large machines with no support for combining might get the best performance from list algorithms that use software combining techniques. any of them can be used as long as they fit the generic operations given in Table 12.

3.7.2. LRU Lists

We have some ordinary list types, such as `afifo`, that maintain at least approximate FIFO ordering under the operations of insertion, deletion, and removal (§3.7.1/p82). It might seem that such lists are suitable for maintaining a list of items in Least Recently Used order, but we don't consider these lists practical for such usage because each successful removal increases the cost of a subsequent deletion, and ratio of removals to deletions may be quite large in LRU lists.

LRU lists are typically used to organize items that are subject to reallocation for other purposes. The classic application of LRU ordering is in virtual memory page replacement policies, although explicit lists such as we propose here are not normally used in practice for paging.

The operations we require for an LRU list are:

- *Getlru*: Delete and return the least recently used item. The item is then typically reassigned, e.g., for another cached object (disk block, etc.).
- *Putmru*: Insert an item as the most recently used. This makes the item eligible for reassignment, but with the expectation that it may be needed again soon.

- *Putlru*: Insert an item as the least recently used. This makes the item eligible for reassignment, with the expectation that it won't be needed again soon.
- *Remove*: Remove a specific item from the list, whatever its position. This operation returns *TRUE* if the removal is successful (i.e., the item was on the list), and *FALSE* otherwise (i.e., the item was not on the list). *Remove* is the mechanism by which a previously released item is reclaimed for use.[55]

To allow sufficient latitude for efficient implementation, we stipulate a requirement only for approximate LRU ordering.

On the face of it, there is nothing to distinguish these operations from the *get*, *put*, and *remove* operations in section 3.7.1 on page 81, other than allowing for insertion at the other end via *putlru*; in fact, we have simply defined an *output-restricted deque* (Knuth [130]). The significant difference lies in expected usage patterns: with LRU lists, it is *expected* that more items will come off the list via *remove* than via *getlru*. This difference has a significant impact on the choice of algorithm for use on our target class of machines, along with many other issues discussed on pages 78–80.

It is required that the LRU list operations be serializable: the outcome of any concurrent execution of these operations (*getlru*, *putmru*, *putlru*, and *remove*) must be the same as some serial ordering. In particular, it is not permitted to successfully *remove* an item that is concurrently returned by *getlru*.

Interface

Table 15, below, gives the functions for LRU lists. To increase the latitude allowed for implementation, we impose a restriction that wasn't present on the lists in section 3.7.1: an item is initialized to interact with a single LRU list; it may not move from list to list. This is reflected in the `lruiinit` function, which specifies the particular `lrulist`.

For the sake of simplicity, we choose not to define list or item un-initialization functions, although it could be done.

Function	*Purpose*
`int lruinit (lrulist *)`	Initialize LRU list
`int lruiinit (lruitem *, lrulist *)`	Initialize LRU item for given list
`lruitem *getlru (lrulist *)`	Return LRU item or `NULL`
`int lruremove (lrulist *, lruitem *)`	Remove item; return success/failure
`void putlru (lrulist *, lruitem *)`	Insert item as least recently used
`void putmru (lrulist *, lruitem *)`	Insert item as most recently used

Table 15: LRU List Functions.

The types `lrulist` and `lruitem` are `typedefs` for appropriate structure definitions.

---

[55] It is arguable that a more appropriate term for this operation is *use*, but our list-centric attitude inclines us more towards *remove*.

Implementation: Overview

There are several possible approaches to implementing LRU lists. For uniprocessors, multiprocessors with few processors, or workloads for which LRU operations are known to be very rare, a doubly linked list is appropriate, with a binary semaphore to protect it on multiprocessors.

One possible approach for larger machines or heavy LRU activity is to use group lock (§3.5.6), a linked list, and an algorithm based on recursive doubling. This requires $O(\log N)$ group lock phases when N processors are performing the same operation concurrently.

The approach we describe here is based on a circular array of linked lists, much as the parallel access queues of Rudolph [174] and Wilson [205]; these are also the basis for `dafifo` and `dqueue` lists (§3.7.1/p82). The central question in such a design is: how to handle removals? Wilson's queue algorithm records how many items have been removed between each remaining item in each sublist, skipping over them at deletion time to maintain FIFO ordering. Instead of keeping counts, we simply mark removed items and leave them on the list; we can do this because we gave up the requirement to allow items to move from list to list. When an item is inserted (via `putmru` or `putlru`) back on the list, we check to see if it's still linked into a sublist or if it has been deleted and skipped over. In the former case, we simply move the item to the proper end of its sublist, and in the latter case we insert it on a sublist chosen with Fetch&Increment. As a result, what we have is essentially a balanced collection of FIFO lists.

Implementation: Details

The items, which are to be included in larger objects of interest, look like this:

```
typedef struct _lruitem {
     struct _lruitem *mru;      // more recently used item
     struct _lruitem *lru;      // less recently used item
     int whichlist;             // which sublist item is on
} lruitem;
```

We use `whichlist` to locate an item's sublist for the removal operation, and also as a status indicator:

- If `whichlist` = 0, the item is on no sublist.
- If `whichlist` > 0, the item is on the sublist with index `whichlist` – 1 (and has not been removed).
- If `whichlist` < 0, the item is present on the sublist with index −`whichlist` – 1, but has been removed.

While this status can be determined atomically by examining `whichlist`, a non-zero value can change asynchronously unless the indicated sublist is locked.

The structure for an LRU list looks like this:

```
typedef struct {
     struct _lrusublist {
          bwlock lk;        // serialize access to linked list
          lruitem head;     // list head for items
     } sub[LRU_NSUBLISTS]; // sublists
     unsigned int dlist;   // sublist for next delete (getlru)
     unsigned int ilist;   // sublist for next insert (putmru, putlru)
     int n;                // current number of items
} lrulist;
```

Insertion of an item (with `whichlist==0`) via `putmru` or `putlru` chooses a sublist with `ufai` acting on the list's `ilist` field; the result is taken mod `LRU_NSUBLISTS` and overflow is ignored.[56] We require `LRU_NSUBLISTS` to be a power of 2. Likewise, deletion of an item via `getlru` chooses a sublist with `ufai` acting on the `dlist` field.

Here is the code for LRU list initialization; we begin with declarations of histograms for instrumentation:

[56] In practice, it may be worthwhile to dynamically allocate the array `sub`. We choose not to do so in this context for the sake of brevity. An example of our preferred manner for performing such dynamic allocation is given in section 3.7.4 on page 109.

```
bw_instrument (ehistgram _lru_hglk);
bw_instrument (ehistgram _getlru_hgwait);

int
lruinit (lrulist *lp)
{
      int i;
      for (i = 0;  i < LRU_NSUBLISTS;  i++) {
            if (!BWL_INIT (&lp->sub[i].lk, 1, &_lru_hglk,
                           "lrulist sublist lock")) {
                  // failed lock initialization
                  while (--i >= 0)
                        bwl_destroy (&lp->sub[i].lk);
                  return 0;
            }
            // empty list head points to self
            lp->sub[i].head.mru = &lp->sub[i].head;
            lp->sub[i].head.lru = &lp->sub[i].head;
            lp->sub[i].head.whichlist = i; // not used
      }
      lp->dlist = lp->ilist = 0;
      lp->n = 0;
      bw_instrument (ehgraminit (&_getlru_hgwait,
                     "empty sublist wait time"));
      return 1;
}
```

In an empty sublist, the two link pointers are set to point back to the sublist head itself, eliminating special cases of insertion into an empty sublist and deletion of the last item in a sublist. An exponential histogram (§3.3), `_lru_hglk`, is set up to collect lock waiting times. The only real complication in LRU list initialization is handling lock initialization failure.

Here is the code for item initialization:

```
int
lruiinit (lruitem *ip, lrulist *lp)
{
      ip->mru = ip->lru = NULL;
      ip->whichlist = 0;
      return 1;  // no failure possible
}
```

The `lp` parameter is ignored in this version, although we still rely on each item being dedicated to a single list; it could presumably be recorded and checked with an assertion in `getlru`, `putmru`, and `putlru` functions.

Here is the code for `getlru`:

```
bw_instrument (ehistgram _getlru_hgempties);
bw_instrument (ehistgram _getlru_hgholes);

lruitem *
getlru (lrulist *lp)
{
     lruitem *ip;
     spl_t s;
     int removed;
     bw_instrument (int empties = 0);
     bw_instrument (int holes = 0);

     ip = NULL;
     while (tdr1 (&lp->n)) {
          struct _lrusublist *list;
          list = &lp->sub[ufai(&lp->dlist) % LRU_NSUBLISTS];
          do { // loop until list is non-empty (optimistic)
               s = splall();  // reduce lock holding time
               bwl_wait (&list->lk, splcheckf(s));
               ip = list->head.mru;  // head.mru is LRU
               if (ip == &list->head) {
                    // list is empty; wait for non-empty
                    bwl_signal (&list->lk);
                    vsplx(s);
                    BUSY_WAIT(list->head.mru != &list->head,
                              nullf, &_getlru_hgwait);
                    bw_instrument (empties++);
               }
          } while (ip == &list->head);
          (list->head.mru = ip->mru)->lru = &list->head;
          removed = (ip->whichlist < 0);
          ip->whichlist = 0;
          bwl_signal (&list->lk);
          vsplx(s);
          if (!removed)
               break;
          // found hole -- must retry
          bw_instrument (holes++);
          ip = NULL;
     }

     bw_instrument (ehgram (&getlru_hgempties, empties));
     bw_instrument (ehgram (&getlru_hgholes, holes));
     return ip;
}
```

Histograms are used for measuring the time waiting for sublists to become non-empty and for measuring the number of "holes" (removed items) skipped over.

We can implement both `putmru` and `putlru` in a reasonably efficient manner with a single routine:

```
#define putmru(lp,ip) _lruput(lp,ip,1)
#define putlru(lp,ip) _lruput(lp,ip,0)

void   // insert as most recently used if most != 0
_lru_put (lrulist *lp, lruitem *ip, int most)
{
      int whlist;
      int doinsert;

      int mylist = ip->whichlist;
      assert (mylist <= 0);  // otherwise not yet removed
moved:
      int onlist = (mylist < 0);
      if (onlist)
           mylist = -mylist;
      else
           mylist = (ufai(&lp->ilist) % NUMPES) + 1;
      struct _lrusublist *list = &lp->sub[mylist-1];
      spl_t s = splall();     // reduce lock holding time
      bwl_wait (&list->lk, splcheckf(s));
```

```
    if (!onlist) {
         // ip->whichlist must still be 0
         doinsert = 1;
         whlist = 0;
         ip->whichlist = mylist;
    }
    else {
         whlist = ip->whichlist;
         if (whlist == 0) {  // no longer on sublist
              bwl_signal (&list->lk);
              vsplx(s);
              mylist = 0;
              goto moved; // do this at most once
         }
         assert (whlist == -mylist);
         ip->whichlist = mylist;
         // already on list; move to proper position
         lruitem *m = ip->mru;
         lruitem *l = ip->lru;
         if (&list->head == (most ? m : l))
              doinsert = 0; // already in proper position
         else {
              // unlink from interior of list
              m->lru = l;
              l->mru = m;
              doinsert = 1;
         }
    }

    if (doinsert) {
         if (most) { // insert as most recently used
              ip->mru = &list->head;
              (ip->lru = list->head.lru)->mru = ip;
              list->head.lru = ip; // head.lru is MRU
         }
         else {     // insert as least recently used
              ip->lru = &list->head;
              (ip->mru = list->head.mru)->lru = ip;
              list->head.mru = ip; // head.mru is LRU
         }
    }
    bwl_signal (&list->lk);
    vsplx(s);
    if (whlist == 0)  // we did a real insert
         vfai (&lp->n);
}
```

The main "trick" of `_lru_put` is to get a tentative sublist by examining `ip->whichlist`, and verify it after locking. Because we assume at most one processor will try to insert a particular item concurrently, we know the item won't be moved from list to list asynchronously.

The final LRU list operation is `lruremove`:

```
int
lruremove (lrulist *lp, lruitem *ip)
{
     int mylist = ip->whichlist;
     assert (mylist >= 0);  // else already removed
     if (mylist == 0)
          return 0;         // lost race with lruget
     struct _lrusublist *list = &lp->sub[mylist-1];
     spl_t s = splall();
     bwl_wait (&list->lk, splcheckf(s));
     int whlist = ip->whichlist;
     assert (whlist == 0 || whlist == mylist);
     if (whlist > 0)
          ip->whichlist = -whlist;
     bwl_signal (&list->lk);
     vsplx(s);
     return whlist != 0;
}
```

This straightforward routine checks to see if the item has already been skipped over by `getlru` and, if not, locks the sublist and checks again. If it has won the race, the item is marked as removed by negating the `whichlist` field.

3.7.3. Visit Lists

The lists described in section 3.7.1 are all similar in that the primary operations supported are:

- *insert* an item,
- *delete* and return the "next" item, and
- *remove* a specific item.

Of course, "next" may be defined differently for each list type, and *remove* isn't supported by all lists. In this section we describe a different kind of list, where the primary operations are:

- *insert* an item,
- *remove* a specific item, and
- *apply a function* to each item on the list.

We call such lists *visit lists*, and the third operation is called *visit*. Visit lists are used to maintain several kinds of process groups in Symunix-2, supporting signal delivery through the visit operation (§4.8.2).

The basic concept of visit lists is certainly not new; most ordinary list structures are easy to traverse in a serial computing environment, especially if there is no ordering requirement. But a highly parallel system can't afford the serial solution for two reasons:

- The insert and remove operations can become a performance bottleneck if one simply protects a serial list with a lock. This is true even if the visit operation is very rare.
- The visit operation may require work proportional to the size of the machine.[57] If performed serially, this can be a scalability problem even when the visit operation is rarely required, since it directly affects the latency of the higher level operation being performed. For example, the latency of signal delivery is affected.

Visit lists allow insert, remove, and visit operations to be performed concurrently. To strike a balance between utility and latitude for implementation, we have chosen the following rules:

(1) All items on a visit list for the entire duration of a visit operation will be visited at least once.

(2) An item might not be visited at all if it is inserted or removed while a visit operation is in progress.

(3) An item may be visited redundantly.

(4) The visit operation may be performed asynchronously, and on more than one processor.

(5) If one visit operation begins before another has completed, some items may be visited only once.

(6) The order in which items are visited is undefined.

(7) The function called for each member may be called from an interrupt handler, so it must not cause a context-switch.

It should be clear from rules 1, 2, 3, and 5 that each visit operation is “anonymous”. Higher level application-specific methods must be employed to keep track of the work to be done by each visit. There is also no built-in mechanism provided for a “visit initiating” processor to wait for, or be notified of, the visit’s completion, but it is simple for higher level software to do so, e.g., by using a reader/writer lock if necessary to prevent insertions or removals from interfering, and by counting the items on the list and the items visited with Fetch&Add; this determines when the work is done, at which point the initiator can be awakened. This, and other desirable features, may be worth incorporating in the visit list mechanism itself, but we assume a minimal specification for simplicity of presentation.

Interface

A visit list is represented by an abstract data type, `vislist`. The data type for a visit list item, `visitem`, is designed to be included as an element in some larger structure of interest. The function to call when visiting each item, which is set when the visit list is initialized, is called with two parameters: a pointer to the `vislist`, and a pointer to the `visitem`.

Table 16, on the next page, lists the functions that manipulate visit lists and items. Functions that may fail, e.g., due to resource shortage, return boolean values; the others return nothing. The most complicated function is `visinit`, but only because of the number of arguments. In addition to the list, and the function to call for each visit, `visinit`

---

[57] This will happen when the size of a particular visit list grows linearly with the size of the machine, a reasonable expectation for many anticipated uses.

Function	*Purpose*
`void visitinit(void)`	Overall initialization (boot time)
`int visinit (vislist *, void f(vislist *, visitem *), int chunksize)`	List initialization
`int visiinit (visitem *)`	Item initialization
`void visdestroy (vislist *)`	List un-initialization
`void visidestroy (visitem *)`	Item un-initialization
`int visput (vislist *, visitem *)`	Insert item on list
`void visremove (vislist *, visitem *)`	Remove item from list
`void visit(vislist *)`	Initiate visit operation

Table 16: Visit List Functions.

The function to call for each item during a `visit` operation is specified by the `f` parameter to `visinit`.

supports a single tuning parameter: `chunksize`. Practical considerations often make it preferable to visit items in *chunks*, reducing the overhead of distributing work by a constant factor. Of course, `chunksize` is only a hint; the implementation is free to ignore it.

Implementation: Overview

While less parallel machines can probably adopt the serial approach, the data structures and algorithms described here are designed for the target class. Figure 3, below, shows a logical view of how we use a stack to implement the three main visit list operations. The general characteristics of our solution are:

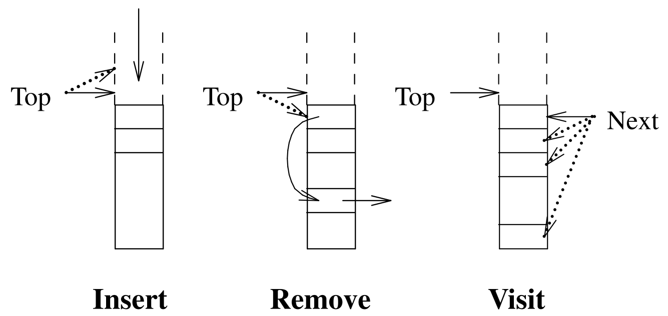


Figure 3: Use of Stack for Visit List Operations.

Insert is simply a push. Removal of the top item is simply a pop; an interior item is replaced by the top item. Fetch&Add allows visit to be performed without serialization.

- Work is performed on an appropriate number of processors by *random broadcast* interrupt handlers (§3.6).
- Each of the operations (insert, remove, and visit) must execute exclusively from the others; this could be called a *Readers/Readers/Readers* situation, generalizing from section 3.5.5.
- The insert and remove operations are each structured as *multi-phase* group lock algorithms (§3.5.6).
- A single group lock is used in a stylized manner to manage the coordination without designating fixed phases for each major operation (insert, remove, and visit).
- The conceptual data structure is that of a stack; removal of an item that isn't on top of the stack is accomplished by replacing it with the top item (see Figure 3).
- Visitation is performed by dynamically assigning "chunks" of items to each processor's interrupt handler. This is accomplished by keeping the stack index of the next item to be visited, starting at the top, and using Fetch&Add to perform assignment. To initiate a visit, one simply sets this index to the top, and interrupts the appropriate number of processors via `rbc`.
- The stack data structure for each visit list is organized as *d direct* item pointers to handle common small lists efficiently, plus *i indirect* pointers to fixed-size arrays of c item pointers allocated from a common pool. Whereas the purely contiguous array approach requires $L \times I$ pointers for L visit lists and I items, our simple strategy requires $L \times (d + i) + L \times c + I$ pointers (assuming c evenly divides I). (Cf. traditional UNIX file block mapping in i-nodes (Bach [15])).

Although we discuss further details below, the need for some of the rules given on page 92 should already be clear. Removal of one item can cause an already-visited item to be relocated on the stack so that it is visited again (redundantly). When one visit operation overlaps with another, the index of the next item to be visited is simply reset to the top, thus causing the items closer to the top to be twice visited and items closer to the bottom to be visited only once.

The success or failure of this design for visit lists depends primarily on two factors:

(1) The quality of the random broadcast mechanism, in terms of overhead, latency, and fairness. The first two factors determine the list size threshold for which the serial approach is preferred. Maximum latency restricts the degree to which visit lists can meet real-time requirements. Unfairness affects overhead and latency, and worsens apparent speed variations among processors in the machine.

(2) The relative priority (importance) of the "visit work", compared to the work interrupted on the random target processors. This can be controlled somewhat by careful interrupt masking, but excessive interrupt masking is detrimental to latency. Another way to sacrifice visit latency to reduce priority inversions is to use visit functions only to generate lower-priority soft interrupts (§3.4/p46) to perform time consuming work.

Implementation: Details

The `vislist` structure contains the following fields:

```
typedef struct _vislist {
     int      top;                // highest index used + 1
     int      next;               // next item to visit, counting down
     rbc_pair pair;               // used to send rbc
     bwglock  g;                  // for visput/visremove/visit
     int      op_pri;             // glock arbitration
     int      opcnt[VISIT_NUMOP]; // count visput/visremove/visit
     int      temp;               // for visremove
     visitem  *dir[NDIRVIS];      // direct item pointers
     visindir *indir[NINDIRVIS];  // indirect item pointers
     int      indirn;             // next indir for visput
                                  // function to call for each item
     void     (*f)(struct _vislist *, struct _visitem *);
} vislist;
```

As already mentioned, we use direct and indirect pointers to conserve space, because the indirect pointer blocks can be shared among all visit lists. The indirect pointer blocks are allocated from a pool (§3.7.1/p82):

```
pool visitfree;
typedef union {
     poolitem pi;
     visitem *ptr[NVISINDIR];
} visindir;
```

The constants `NDIRVIS`, `NINDIRVIS`, and `NVISINDIR` determine the maximum capacity of a visit list:

```
#define NDIRVIS      Number of direct pointers per list
#define NINDIRVIS    Number of indirect pointers per list
#define NVISINDIR    Number of pointers per indirect block
#define VISLIMIT     (NDIRVIS+NINDIRVIS*NVISINDIR)
```

For positions greater than `NDIRVIS`, two macros are convenient:

```
#define _VDIV(pos) (((pos)-NDIRVIS) / NVISINDIR)
#define _VMOD(pos) (((pos)-NDIRVIS) % NVISINDIR)
```

The items have a structure, too:

```
typedef struct _visitem {
     int index;            // position of item in vislist
     int mark;             // for use by visremove
} visitem;
```

Initialization. Visit lists and items must be explicitly initialized and un-initialized. Some fields deserve explicit mention now; the others will be described as needed.

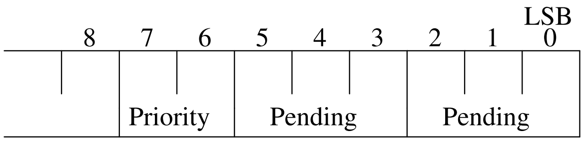
- *Visit item initialization.* The `mark` field is set to 0.
- *Visit list initialization.* The RBC parameters in `pair` are set for the internal function `_dovisit`, to be described on page 108, with an argument pointing to the visit list itself. Eventually, `_dovisit` will call the function pointed to by `f`, which is also set when the

visit list is initialized. The visit list field `top` is set to 0, to indicate an empty list, and `next` is set to −1, to indicate no visit operation in progress. The other integer fields (`op_pri`, `opcnt`, `temp`, and `indirn`, are all set to 0.

Arbitration. Coordination of the three main visit list operations, `visput`, `visremove`, and `visit`, is achieved with a group lock. The algorithms we use require quite a few group lock phases, so instead of assigning all the phases to one large sequence, we perform dynamic arbitration to control which of the three operations is being performed at any time. This is the purpose of the `vislist` fields `op_pri` and `opcnt`. We assign each of the three operations a number:

```
#define VISIT_PUT    0
#define VISIT_REMOVE 1
#define VISIT_VISIT  2
#define VISIT_NUMOP  3  // Number of operations
```

This number is used as an index into `opcnt`, and to indicate which operation has priority in `op_pri`. Each element of the `opcnt` array indicates how many processors are performing the corresponding operation. The `op_pri` field encodes several bit fields:



The low-order 6 bits indicate which of the 3 operations are pending; the same information is recorded twice in two 3 bit fields. The next two bits indicate which operation has priority in case of contention.

We define an internal function, `_visglock` to perform this arbitration. In two group lock phases, `_visglock` determines which operation may proceed; processors wishing to execute this operation return so they may perform it (with additional phases as needed), while processors that must wait use the `bwg_relock` function to guarantee a place in the very next group, at which time another arbitration decision is made. Figure 4, on the next page, shows a transition diagram for group lock phases used by visit operations.

Each of the functions `visput`, `visremove`, and `visit` (and the internal function `_dovisit`, acting on behalf of `visit`), begins with a call to `_visglock`. Here is the first phase of `_visglock`:

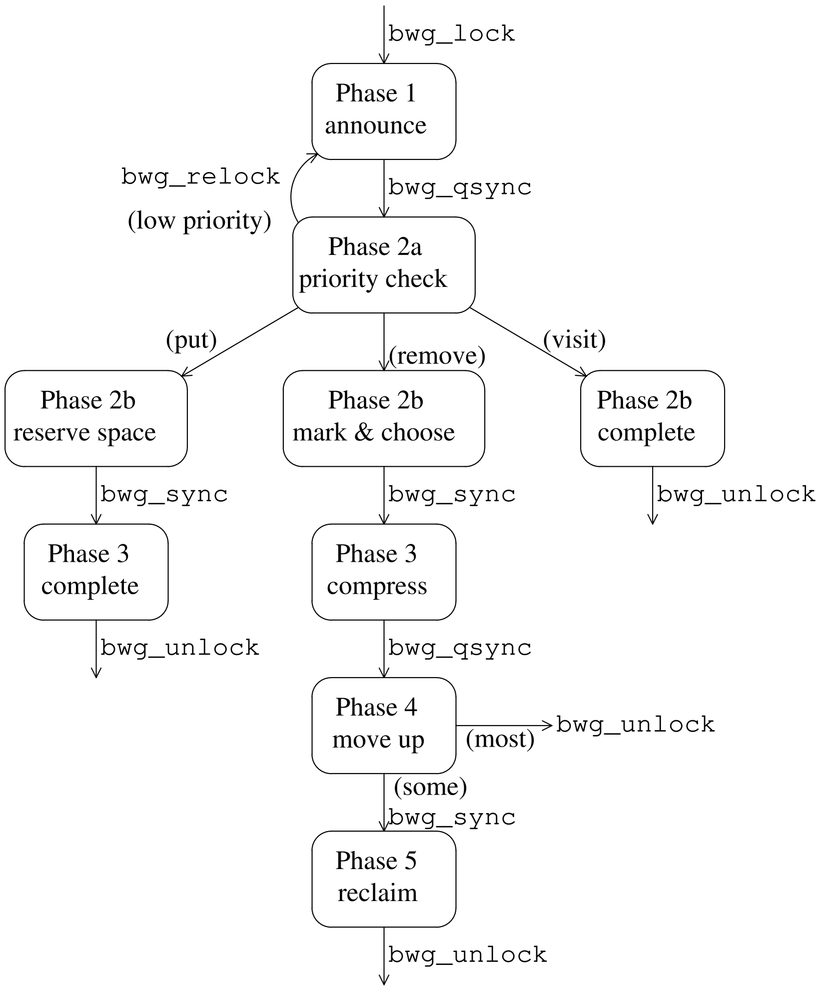


Figure 4: Group Lock Phase Transitions for Visit List Operations.

The first two levels of the diagram are performed by `_visglock`. Phases 2a and 2b are actually part of a single phase; only one of the alternate 2b phases is chosen for execution by each group.

```
          // interrupts are already masked
int       // op is VISIT_PUT, VISIT_REMOVE, or VISIT_VISIT
_visglock (vislist *v, const int op, spl_t s)
{
     // calculate my contribution to pending fields of op_pri
     int myopbits = (1<<op)|(1<<(op+VISIT_NUMOP));
     bwg_lock (&v->g, splcheckf(s));
     // PHASE 1: announce our presence

     int gs = bwg_size (&v->g);
p1again:
     if (gs == 1) {     // fast path; no contention
          v->opcnt[op] = 1;
          // set priority for next arbitration round
          v->op_pri = (((op+1)%VISIT_NUMOP) << (2*VISIT_NUMOP));
          return 0;
     }
     int pri = faor (&v->op_pri, myopbits) >> (2*VISIT_NUMOP);
     int r = fai(&v->opcnt[op]);    // return value and index
```

The pending bits of interest within `op_pri` depend only on the constant parameter `op`, so they are computed before entering the group lock. When there is no contention (the group size is 1) we avoid all further group lock overhead. Otherwise, we use `faor` to register our intent to perform our operation and also to retrieve the current operation priority value from `v->op_pri` (during phase 1 the priority bits in `v->op_pri` are stable, i.e., not changing). We also use `fai` to obtain a unique position within the ordering of all processors participating in our operation.

In the second phase, the pending bits within `op_pri` will be stable, and we will have enough information to know whether we may proceed or try again with the next group:

```
p2again:
      bwg_qsync (&v->g, gs);
      // PHASE 2: may we proceed?

      int myop = (myopbits >> pri) & ((1<<VISIT_NUMOP)-1);
      int op_pri = v->op_pri;  // stable in this phase
      if (((op_pri >> pri) & (myop-1)) != 0) {
            // we are unlucky this time
            pri = (pri + count_lsz(op_pri) + 1) % VISIT_NUMOP;
            bwg_relock(&v->g);
            // PHASE 1: all over again

            gs = bwg_size (&v->g);
            if (gs == 1)
                  goto p1again;
            goto p2again;
      }
      if (r == 0)    // set pri for next arbitration round
            v->op_pri = (((op+1)%VISIT_NUMOP) << (2*VISIT_NUMOP)) |
                         (op_pri & ~myopbits & ((1<<(2*VISIT_NUMOP))-1));
      return r;
}
```

The `if` expression tests whether there are any other operations with higher priority than ours and with active processors seeking to perform them; if so, then we must wait for the next group by calling `bwg_relock`. We don't need to reference `v->op_pri` within the next phase 1, since we compute the next value of `pri` locally. The function `count_lsz` returns the number of least significant zero bits in its argument, i.e., `count_lsz`$(x) = \log_2(((x \ XOR \ (x-1))/2)+1)$. After calling `bwg_relock`, we go back to re-execute phase 1 or phase 2; the reason for the distinction, based on group size, is to avoid repeating our original affect on `v->opcnt` or recomputing `pri` or `r`. If, on the other hand, we have priority to proceed, we must set the priority bits within `v->op_pri` for the next group; the test for `r == 0` ensures this is done only once.[58]

The cost of executing `_visglock` is quite low:

- The common case of no contention (only one processor executing `_visglock`) costs only 2 shared memory stores plus the cost of `bwg_lock` and `bwg_size`.
- When the caller's operation has priority, the cost is only 3 or 4 shared memory references, plus the cost of `bwg_lock`, `bwg_size`, and `bwg_qsync`.
- Each time the caller's operation doesn't have priority, the additional cost is 1 shared memory reference plus `bwg_relock`, `bwg_size`, and possibly `bwg_qsync`. This can only happen twice, since there are only three operation types and `bwg_relock` doesn't skip groups (§3.5.6/p73).

---

[58] The test for `r == 0` could be omitted, because each participating processor would compute the *same* new value for `v->op_pri`. The redundant stores, however, can increase memory traffic severely on machines without combining.

The return value of `_visglock` will be different for each processor executing the same operation concurrently; the processor receiving 0 is obligated to clear the appropriate element of `v->opcnt` before releasing the group lock.

The `visput` Operation. The management of direct and indirect `visitem` pointers complicates `visput`, forcing the use of an extra group lock phase. The initial phase determines placement of the new item and allocates storage, and the extra phase handles allocation failure and performs the insertion.

The first part of `visput` obtains the lock, determines group size, and checks for room (this check can be eliminated if `VISLIMIT` is large enough to be irrelevant):

```
int
visput(vislist *v, visitem *vitem)
{
     int   t,  // index for new item, derived from v->top
           r,  // return value
           gi, // group index, from [0,...,gsize-1]
           gs; // group size
     spl_t s;

     s = splrbc();        // visit uses random broadcast
     gi = _visglock (v, VISIT_PUT, s);
     // PHASE 2, continued: reserve space, allocate indir pointers

     gs = v->opcnt[VISIT_PUT];
     if (gi == 0)
          v->opcnt[VISIT_PUT] = 0;

     t = fai(&v->top);
     if (t >= VISLIMIT) { // check for room
          vfad(&v->end);
          r = 0;          // failed
          goto done;
     }
```

Note that we both read and write `v->opcnt[VISIT_PUT]` in the same group lock phase; this constitutes a race, but one which doesn't affect correctness: the value of `gs` is significant only to the extent that it is 1 or not 1. If the group size is 1, no race occurs. If the race occurs, it is possible for `gs` to falsely assume the value 0, but this has the same effect within `visput` as any other non-1 value. Tolerating this race is convenient, because it allows us to skip phase 3 in the error case (where we are out of room). Normally, we might avoid use of "early exit" for the sake of an error condition, since it prevents the use of "quick" barriers elsewhere, but we already have another instance of early exit from phase 2, when `_visglock` calls `bwg_relock`, so we might as well take advantage of it.

The "fast path" shows clearly what must be done to complete the `visput` operation, without concern for parallelization:

```
if (gs == 1) {        // fast path if no contention
      r = 1;
      if (t < NDIRVIS)
            v->dir[t] = vitem;
      else {    // use indirect pointers
            if (_VMOD(t) == 0) {
                  // allocate block of pointers first
                  // note v->indirn equals _VDIV(t)
                  visindir *x = poolget(&visitfree);
                  if (x == NULL) {
                        vfad(&v->top);
                        r = 0;     // failed
                        goto done;
                  }
                  poolidestroy (&x->pi);
                  v->indir[fai(&v->indirn)] = x;
            }
            v->indir[_VDIV(t)]->ptr[_VMOD(t)] = vitem;
      }
}
```

The tricky part of `visput` involves storage allocation and reclamation; the fast path case shows that `v->indirn` is equivalent to `_VDIV(t)` when things are serialized. The real need for `v->indirn` arises when we have to deal with concurrency:

```
else {  // concurrent visput operations
     if (t < NDIRVIS) {
          bwg_sync (&v->g);
          // PHASE 3: store item pointers

          r = 1;
          v->dir[t] = vitem;
     }
     else {    // use indirect pointers
          if (_VMOD(t) == 0) {
               // allocate block of pointers
               visindir *x = poolget(&visitfree);
               if (x != NULL) {
                    poolidestroy (&x->pi);
                    v->indir[fai(&v->indirn)] = x;
               }
          }
          bwg_sync (&v->g);
          // PHASE 3: store item pointers

          r = 1;
          if (v->indir[_VDIV(t)] == NULL) {
               // some allocation failed
               vfad(&v->top);
               if (_VMOD(t) == 0)
                    vfad(&v->indirn);
               r = 0;      // failed
               goto done;
          }
          v->indir[_VDIV(t)]->ptr[_VMOD(t)] = vitem;
     }
}
```

The direct pointer case is no more complicated than in the fast path, in fact the assignment to `v->dir[t]` could be moved to phase 2 without harm (but we put it in phase 3 to balance the greater amount of work performed in the indirect pointer case).

Consider the case where several indirect pointer blocks must be allocated concurrently; it is possible for some of the allocations to fail. This is the real purpose of `v->indirn`: to ensure that any allocation failures affect only the top of the stack, not some middle position. Thus, the `visput` operations which ultimately fail may not be the same ones that got `NULL` from `poolget`.

The final act of `visput` is to point each item back to its position on the visit list; this is for the benefit of `visremove`:

```
        vitem->index = t;
done:
        bwg_unlock (&v->g);
        vsplx(s);
        return r;
}
```

The* `visremove` *Operation. This is more complicated than `visput`, requiring 3 new phases; the last phase is dedicated to storage reclamation for indirect pointer blocks.

The basic strategy for removing an item is to replace it with an item popped from the stack. Since the algorithm supports parallel execution, many removals may occur concurrently. Assume we have d concurrent removals; each of the top d items will be replacements unless they are to be deleted. The essence of the algorithm is to identify the replacements so they can be moved to their new locations. This is done by using the `mark` field in the `visitem` structure. Figure 5, below, gives a simplified view of the situation at the end of each of the main `visremove` phases. The management of indirect pointer blocks is omitted from the diagrams for the sake of clarity, but not from the full code for `visremove`:

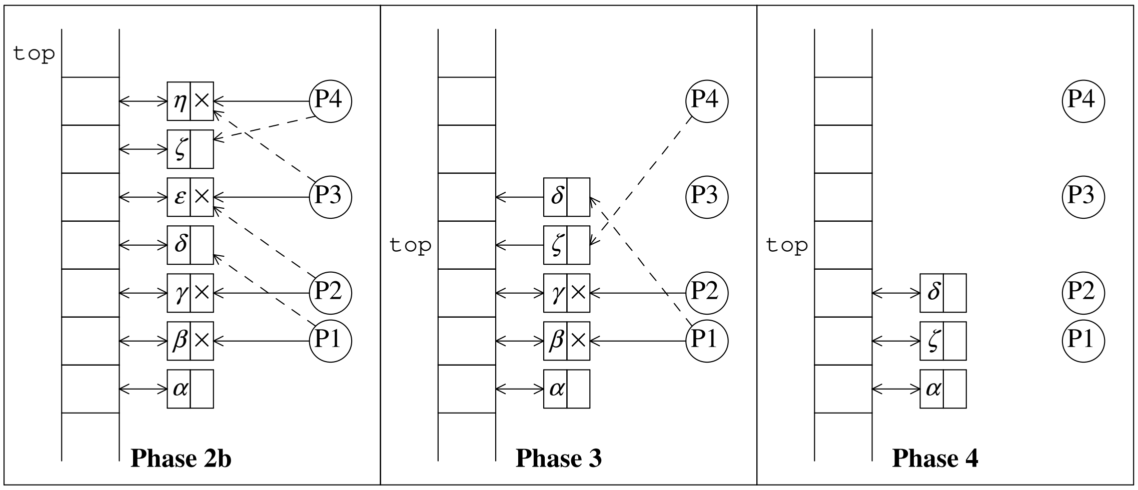


Figure 5: Three Phases of `visremove`.

Processors P1, P2, P3, and P4 are trying to delete items β, γ, ε, η. For simplicity, we pretend only direct item pointers are used. By the end of the first phase (phase 2b in Figure 4) each processor has marked its own item with × and chosen an item to be deleted or moved (indicated with dashed lines). By the end of the next phase `top` has been updated to account for all deletes, and items to be moved are now contiguous on the stack (but their order is not preserved, as shown in this example). By the end of the last phase the items to be moved have reached their final positions. (Use of indirect pointers force use of a phase 5 in the code on page 106.)

```
void
visremove(vislist *v, visitem *vitem)
{
      int t,      // early copy of v->top
          d,      // number removals
          gi,     // index of caller in group
          n;      // index of item being removed
      visitem *p; // possible item to move up
      visindir **xp = NULL;
      visindir *x;
      spl_t s;

      s = splrbc();
      gi = _visglock (v, VISIT_REMOVE, s);
      // PHASE 2, continued: mark items and get replacements

      d = v->opcnt[VISIT_REMOVE];
      t = v->top;
      n = vitem->index;
```

The value obtained for `d` would be the same as `bwg_size` if there were no other operations vying for priority in `_visglock`.

The "fast path", taken when there is no real concurrency to deal with, shows what must be done in the simplest terms:

```
      if (d == 1) {           // take fast path if no contention
           if (n < t-1) {
                // identify replacement for item being removed
                if (t-1 < NDIRVIS)
                     p = v->dir[t-1];
                else
                     p = v->indir[_VDIV(t-1)]->ptr[_VMOD(t-1)];
                // move up replacement item
                if (n < NDIRVIS)
                     v->dir[n] = p;
                else
                     v->indir[_VDIV(n)]->ptr[_VMOD(n)] = p;
                p->index = n;
           }
           if (t-1 >= NDIRVIS && _VMOD(t-1) == 0) {
                // reclaim storage
                xp = v->indir + _VDIV(t-1);
                v->indirn = _VDIV(t-1);
                x = *xp;
                *xp = NULL;
           }
           goto done;
      }
```

We now expand the code to handle concurrency. Items to remove are marked and potential replacements are identified:

```
vitem->mark = 1;
if (t-gi-1 < NDIRVIS)
      p = v->dir[t-gi-1];
else {
      p = v->indir[_VDIV(t-gi-1)]->ptr[_VMOD(t-gi-1)];
      // prepare for storage reclaim, phase 5
      if (_VMOD(t-gi-1) == 0) {
            xp = v->indir + _VDIV(t-gi-1);
            vfad(&v->indirn);
      }
}
```

Note that a potential replacement can't be used if it is itself slated for removal. We must solve a matching problem: removal items that are not replacement candidates must be paired up with replacement candidates that are not removal items. We do this in two phases: first we squeeze the removal items out of the set of possible replacements:

```
bwg_sync (&v->g);
// PHASE 3: compress unmarked replacements

v->top = t-d;
if (v->next >= t-d)  // watch for pending visit
      v->next = t-d-1;
if (!p->mark) {
      gi = fai(&v->temp);
      if (t-d+gi < NDIRVIS)
            v->dir[t-d+gi] = p;
      else
            v->indir[_VDIV(t-d+gi)]->ptr[_VMOD(t-d+gi)] = p;
}
```

(Note the check for `v->next`; this relates to the action of `_dovisit` (p108).) Once we have weeded out any removal items from the replacement candidates, it is simple to choose one for each case where needed. To do this, we use an extra variable, `v->temp`. This variable was set to 0 when the list was first initialized, then incremented for each true replacement item in phase 3. Finally, we decrement `v->temp` again as we allocate those replacement items in phase 4:

```
        bwg_qsync (&v->g, d);
        // PHASE 4: replace items not at top

        if (n < t-d) {
             // choose replacement
             gi = fad(&v->temp) - 1;
             if (t-d+gi < NDIRVIS)
                  p = v->dir[t-d+gi];
             else
                  p = v->indir[_VDIV(t-d+gi)]->ptr[_VMOD(t-d+gi)];
             // use replacement
             if (n < NDIRVIS)
                  v->dir[n] = p;
             else
                  v->indir[_VDIV(n)]->ptr[_VMOD(n)] = p;
             p->index = n;
        }
```

Thus, `v->temp` is the only additional storage required, and is self-reinitializing.

We need an extra phase for storage deallocation, since the replacements pointers are being read from that storage in phase 4. We used the local variable `xp` to identify the relevant pointer blocks in phase 2; now we set the elements of the `v->indir` array so we can release the group lock as soon as possible:

```
        if (xp != NULL) {
             bwg_sync (&v->g);
             // PHASE 5: reclaim storage
             x = *xp;
             *xp = NULL;
        }
```

Finally, any pointer blocks deallocated can be returned to the pool without holding the group lock:

```
  done:
        if (d == 0)
             v->opcnt[VISIT_REMOVE] = 0;
        bwg_unlock (&v->g);
        vsplx(s);
        if (xp) {
             pooliinit (&x->pi);
             poolput (&visitfree, &x->pi);
        }
        vitem->mark = 0;
  }
```

We take care to clear `v->opcnt` only once, although this is not necessary for correctness, since only the constant 0 is being stored.

The* `visit` *Operation. Triggering a `visit` operation isn't nearly as complicated as `visput` and `visremove`, but even though it works in only one group lock phase, it is potentially more expensive.

As soon as we pass `_visglock`, the size of the list is stable and equal to `top`; we can either do the visitations directly (e.g., if the list is small) or set `next` to `top-1`. Even if we don't do the visitations directly, they are scheduled at this point. We force an appropriate number of processors to work on the visitations via random broadcast, `r_rbc`, after releasing the group lock:

```
void
visit(vislist *v)
{
      int th = v->pair.smallthresh;
      spl_t s = splrbc();
      if (_visglock (v, VISIT_VISIT, s) == 0)
            v->opcnt[VISIT_VISIT] = 0;
      int top = v->top;

      // do small lists now, if interrupt not masked
      if (top < th && !issplgeq(s, fakesplrbc()))
            _visloop (v, top-1, 0);
      else
            v->next = top - 1;
      bwg_unlock (&v->g);
      vsplx(s);
      if (top >= th || issplgeq(s, fakesplrbc()))
            r_rbc (top, &v->pair);
}
```

We will see the `_visloop` function on the next page.

Note that `visit` changes `v->next` without concern for its previous value. Its value will be −1, unless visitations are pending. The worst that can happen is some items will be re-visited, which is allowed under the rules on page 92 (but don't think this was a coincidence).

The internal function `_dovisit` was registered with the random broadcast `rbc_pair` structure at `vislist` initialization time. It is called by the random broadcast interrupt handler with a generic argument (which we must convert into a `vislist` pointer) and a parameter telling how much work to do:

```
void
_dovisit(genarg_t vlistp, int howmany)
{
      vislist *v = (vislist *)vlistp.vd;
      if (v->next < 0)
            return;   // nothing to do
      if (_visglock (v, VISIT_VISIT, rpl()) == 0)
            v->opcnt[VISIT_VISIT] = 0;
      int next = faa (&v->next, -howmany);
      int stop = next-howmany;
      if (stop < 0)
            stop = 0;
      _visloop (v, next, stop);
      bwg_unlock (&v->g);
}
```

The items to visit are assigned to visiting processors by decrementing `v->next` with `faa`.

The work of visiting a "chunk" of items is performed by a simple internal routine, `_visloop`:

```
void
_visloop (vislist *v, int next, int stop)
{
      for (;  next >= stop;  next--) {
            if (next < NDIRVIS)
                  v->f(v, v->dir[next]);
            else
                  v->f(v, v->indir[_VDIV(next)]->ptr[_VMOD(next)]);
      }
}
```

Its only real complication is having to deal with direct and indirect item pointers.

3.7.4. Broadcast Trees

The subject of this section, a structure we call the *broadcast tree*, was conceived to solve part of a particular problem: efficient bottleneck-free delivery of signals to process groups (§4.8.2). We discuss it separately because the approach used is independent of, and more general than, the subject of signals, and because it is interesting in its own right.

The problem to be solved is to create a bottleneck-free structure that supports a limited form of broadcast to all members of a group, with well defined semantics even in the face of concurrent group membership changes. The information to be broadcast is restricted to a single integer selected from a predetermined range. The method of distribution is to place the information into a central structure (the broadcast tree), and to rely on group members to check for recently added data at appropriate times. (As we will describe in section 4.8.2, a visit list (§3.7.3), which in turn relies on random broadcast (§3.6/p76), is used to force polling and deal with processes that are currently unable to poll, e.g., because they are blocked or waiting to run. The primary function of the broadcast tree in this case is to provide the desired signal semantics.) Each member retains sufficient state to allow it to distinguish new information from old in the broadcast tree.

A suitable implementation for a small machine (outside the target class) might use a linked list of group members, and deliver the information serially, i.e., a special form of visit list (§3.7.3). In that case, a broadcast would be received by all group members, and membership changes couldn't be concurrent.

With a broadcast tree, a broadcast is guaranteed to be received by all processes that are group members *at the time of the broadcast*, but not until they poll for recent broadcasts. A member leaving the group performs the *resign* operation, which simply discards all broadcast information since the last poll; it is the final poll that matters. A process *joins* the group in one of two ways: directly, as when changing from one group to another, or in synchrony with a parent, in the case of `fork` or `spawn`. In the latter case, a new child effectively joins the group immediately after the parent's previous poll, thus making broadcasts atomic with respect to child creation. When a new member performs its first poll, it automatically ignores broadcasts from the time before it joined the group. By decoupling the broadcast initiation from the data delivery in this fashion, we not only allow for non-serial delivery, but also provide useful semantics (the traditional ones).

Interface

Table 17, on the next page, lists the function interface for broadcast trees. Because a broadcast tree is restricted to handle a range of integers, we provide a macro `BCT_DECL(range)` to produce a customized declaration suitable for the range [0,...,`range`-1]. This approach is useful since the amount of memory required may be a function of `range`, and there is no notion of a variable-sized structure in standard C. Traditionally, this kind of situation has been dealt with by (1) using a pointer in a fixed-size structure to access the variable-sized component as a dynamically allocated array, or by (2) putting a small (e.g., 1 element) array at the *end* of a fixed-size structure, and taking advantage of the lack of run-time subscript checking. The solution we adopt here (`BCT_DECL`) is more robust and more conducive to alternative implementations requiring different data structures than (2), and doesn't require the additional memory access to fetch the pointer, as in (1).

In order to provide a single set of functions to operate on different broadcast tree types, we use an implementation-defined *generic* broadcast tree declaration, `bct_generic;` a pointer to any tree declared with `BCT_DECL` may be converted into a generic one by using a `(bct_generic *)` cast, and the functions listed in Table 17 operate on such converted pointers. Of course, any broadcast tree must be initialized; this is accomplished by calling `bct_init`, and giving the same range provided in the `BCT_DECL` macro call. After initialization, the other broadcast tree functions must be able to determine the range from the generic structure.

For reasons we will describe in section 4.8.2, we must provide the ability to operate atomically on more than one broadcast tree. We do this by separating out the synchronization component of the broadcast tree into a separate abstract data type: `bct_sync`. This allows us to introduce broadcasting and polling functions that operate on more than one broadcast tree (`bct_put`*n* and `bct_get`*n*, for $1 \le n \le 3$).

Each member of a group to receive broadcasts is represented by a variable of the abstract type `bct_memb`. This structure contains whatever state is needed by members to make the correct decision of which broadcast data to ignore when one of the `bct_get`*n* functions is called. It is initialized by calling `bct_minit`.

Function	*Purpose*
`BCT_DECL`(*r*)	Tree declaration (macro)
`bct_init`(*t*,*r*)	Tree initialization
`bct_destroy`(*t*)	Tree un-initialization
`bct_sinit`(*s*)	Synchronization structure initialization
`bct_sdestroy`(*s*)	Synchronization un-initialization
`bct_minit`(*m*)	Member initialization
`bct_mdestroy`(*m*)	Member un-initialization
`bct_join1`(*s*,*t*,*m*)	Join 1 tree group
`bct_resign1`(*s*,*t*,*m*)	Leave 1 tree group
`bct_dup1`(*s*,*t*,*m*,*n*)	Proxy join for 1 tree (allow `bct_mcopy`)
`bct_mcopy`(*from*,*to*)	Copy member (target pre-initialized)
`bct_put1`(*s*,*t*,*i*)	Broadcast to 1 tree
`bct_put2`(*s*,*t1*,*t2*,*i*)	Broadcast to 2 trees
`bct_put3`(*s*,*t1*,*t2*,*t3*,*i*)	Broadcast to 3 trees
`bct_get1`(*s*,*t*,*m*,*b*)	Poll for broadcast data from 1 tree
`bct_get2`(*s*,*t1*,*t2*,*m1*,*m2*,*b*)	Poll for broadcast data from 2 trees
`bct_get3`(*s*,*t1*,*t2*,*t3*,*m1*,*m2*,*m3*,*b*)	Poll for broadcast data from 3 trees

r = `int` (the range of the broadcast tree)
t,*t1*,*t2*,*t3* = `bct_generic *` (broadcast tree structure)
s = `bct_sync *` (synchronization structure)
m,*m1*,*m2*,*m3* = `bct_memb *` (member structure)
n = `int` (number of member copies to be allowed)
i = `int` (item to be broadcast)
b = `int *` (bit string for result)

Table 17: Broadcast Tree Functions.

For the sake of generality, and the ability to change the underlying implementation, `bct_init`, `bct_sinit`, and `bct_minit` return boolean values. The primary reason is to allow them to fail if some needed resource (such as additional memory) is unavailable, although the current implementation doesn't admit the possibility of any such failure.

Polling is accomplished by calling the aforementioned `bct_get`*n* functions, which set bits in a caller-supplied bit string for each data item broadcast to the indicated tree group(s) since the last poll. Since each broadcast data item is represented by a single bit, no distinction is made between one and more than one such item received. The `bct_get`*n* functions return the number of bits set.

In a UNIX-like environment, where broadcast tree members are often processes, we must make special provision for process *fork* semantics. This is the purpose of the `bct_dup1` function; it provides for clean broadcast semantics in the face of `fork` and `spawn` system calls. After `bct_dup1` is called, direct copies of the parent's `bct_memb` can be made by calling `bct_mcopy`.

Implementation: Overview

The broadcast tree algorithm for the target class of machines is based on the group lock (§3.5.6). The basic idea is due to Eric Freudenthal: For a tree that can handle a range of integers from 0 to *range* − 1, build a binary tree with *range* leaves and record the time of last broadcast in each leaf. In each interior node, record the maximum of the broadcast times of its children. Thus, the root node gives the most recent broadcast time for any item. Broadcast times are only relative, i.e., they do not correspond to real time. Items broadcast concurrently have the same time value. The value 0 is used to represent an infinitely old time. Each member of the group always remembers the time associated with the last received item(s).

Any number of broadcast operations can be performed together by working from the leaves to the root and copying the current time value to each node on the path. Each concurrent broadcaster has the same notion of "current time", so by checking each node before updating it (or, more surely, by using Fetch&Store), a broadcaster may quit as soon as it finds a node that has already been updated.[59] A natural implementation of this algorithm uses a group lock to form groups of broadcasters. In parallel, the group can then

(1) Compute the current time as 1 + *root node time*.

(2) Perform barrier synchronization among themselves (§3.5.6/p69).

(3) Update nodes from leaf to root.

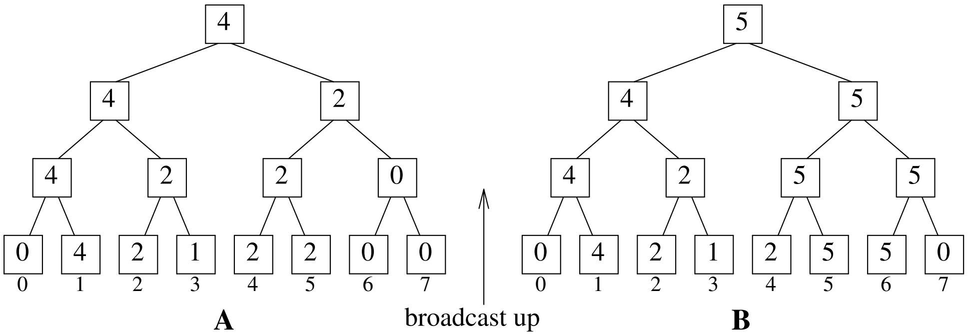


Figure 6: Broadcast Example.

In this example, **A** shows a tree after 4 broadcast time steps. We see that item 1 was broadcast most recently during time step 4, items 2, 4, and 5 where broadcast most recently during time step 2, and item 3 was broadcast most recently during time step 1. In **B**, we show the effect of concurrent broadcasts for items 5 and 6.

[59] The value of this optimization has not been determined. We note that the trees will often be quite shallow and the number of concurrent broadcast operations may not be large. The availability of hardware support (for Fetch&Store) and combining (for store or Fetch&Store) is a key factor.

No synchronization is necessary during step 3. Figure 6, on the previous page, gives an example.

Any number of poll operations can be performed together by doing depth-first search for all leaves marked with a time more recent than that recorded in the relevant `bct_memb` structure, and then updating the time in the `bct_memb` structure to the current root node time. This requires no synchronization other than exclusion of broadcasters. Figure 7, below, gives some examples.

The join, resign, and dup operations require no explicit synchronization, however the ability to read the root node time atomically is assumed for the join operation, which simply copies it to the `bct_memb` structure. The resign and dup operations are empty in the current implementation, except for diagnostic support code.

A simple overall implementation strategy (the one adopted) is to use a group lock with the first phase devoted to `bct_get`*n* and the first part of `bct_put`*n*, and the second phase devoted to the remainder of `bct_put`*n*. There is no need for `bct_join1`, `bct_resign1`, or `bct_dup1` to deal with the group lock at all, assuming `bct_join1` can read the root node time atomically. The trickiest part is handling time overflow within the tree nodes, which could cause loss of broadcast information. We see three attractive approaches:

(1) *Panic*. If the expected time to overflow is large compared to the expected time between system reboots, then we need not handle overflow at all, except to detect it as

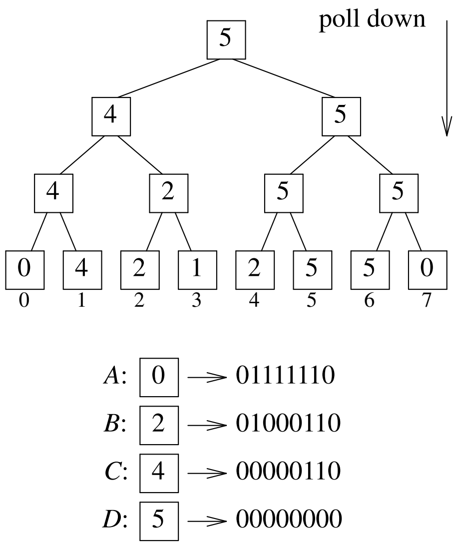


Figure 7: Poll Example.

Using the tree from Figure 6 **B**, we see the effect of 4 possible poll operations. Each member, *A*–*D*, is shown as a box with the time stamp from the previous poll operation. The result of the new poll performed at time 5 is given for each as a binary string, with the first bit representing item 0 and the last bit representing item 7.

an anomaly. In the unlikely event that overflow does occur, the `panic` function may be called to crash the system in a somewhat controlled manner. Of course, such a solution would be inappropriate for many operational environments.

As a concrete example, if time values are represented as 32-bit unsigned integers, and one broadcast occurs every millisecond, overflow won't occur for almost 50 days. With 64 bit unsigned integers, `panic` would probably be an acceptable way to handle overflow in all but the most unusual environments.

(2) *Postpone overflow* by increasing the range of time values beyond what is normally representable as an atomically-accessible integer by the machine. Each time value could be represented by a multi-word structure, for example. However, this solution might require additional synchronizing overhead, if the larger values can't be accessed or modified atomically. Of course, this only postpones disaster, but perhaps that is enough to make the *panic* approach acceptable (e.g., using 64 or 128 bit unsigned integers).

(3) *Allow wraparound*, using unsigned integers for time values. On a machine with two's complement arithmetic, comparison of two time values, whose "true difference" is representable as a signed integer, is performed by regarding the result of a subtraction as a signed number.[60] If members never lag behind broadcasts by more than $2^{wordsize-1}-1$, nothing will be lost. It is necessary, however, to "erase" the memory of any broadcasts that haven't been superseded in that time. This can be performed by inspecting the whole tree whenever the most significant bit (MSB) of the root node time value is about to change, and setting very old nodes, those whose MSB matches the new root value's MSB, to 0 (infinitely old). It is also important to "skip" the value 0 as wraparound occurs.

Leaving little to the imagination, we include the last (and most complex) of these alternatives in our more detailed presentation.

Implementation: Details

As already explained, we use group lock for synchronization:

```
typedef struct {
     bwglock gl;
} bct_sync;
```

Within a broadcast tree structure, the nodes are stored as an array, with the children of element i stored in positions $2i+1$ and $2i+2$. Here is the definition of `BCT_DECL`:

[60] See `udiff` in section 3.5.6 on page 72.

```
#define BCT_DECL(r)                                          \
     struct {                                                \
          int                    range;                      \
          volatile int           period_count[2];            \
          volatile unsigned int tree[2*(r)-1];               \
     }
```

A broadcast tree can accept items from 0 through `range-1`.[61] The two elements of `period_count` contain the number of members whose next unreceived broadcast times' MSB is 0 and 1, respectively. These counts are maintained by `bct_join1`, `bct_resign1`, `bct_dup1`, and `bct_get`*n*, and checked by `bct_put`*n* to verify that no periodically discarded old data remains unreceived (see `_GET_CLEANUP`, on page 117, and `_PUT_CAUTION`, on page 120); `period_count` and all code references to it can be eliminated if one is willing to sacrifice that check. The heart of the algorithm revolves around `tree`, which stores the time of last broadcast for each leaf or internal node of the binary tree. The nodes of the tree are initialized to zeros to indicate that the last broadcast was infinitely long ago.

A rather ordinary definition is all that is needed for members:

```
typedef struct {
     unsigned lastcheck;
} bct_memb;
```

The single field, `lastcheck`, is simply a copy of the root time value when the member joined the group or last executed `bct_get`*n*.

Here are definitions for the simplest functions, `bct_join1`, and `bct_resign1`, `bct_dup1`, and `bct_mcopy` (`BCT_MSB` is an unsigned integer constant with only the most significant bit set):

```
void
bct_join1 (bct_sync *ts, bct_generic *t, bct_memb *m)
{
     unsigned int tstamp;
     tstamp = t->tree[0];
     m->lastcheck = tstamp;
     vfai(&t->period_count[(tstamp&BCT_MSB)!=0]);
}

void
bct_resign1 (bct_sync *ts, bct_generic *t, bct_memb *m)
{
     vfad(&t->period_count[(m->lastcheck&BCT_MSB)!=0]);
}
```

[61] It would be slightly more efficient to pass `range` as a parameter to the various `bct_` functions, rather than keeping it in the structure itself and fetching it from memory for each operation. The combination of macros and a good measure of discipline can make such an approach reasonable.

```
void
bct_dup1 (bct_sync *ts, bct_generic *t, bct_memb *m, int mult)
{
      vfaa(&t->period_count[(m->lastcheck&BCT_MSB)!=0], mult);
}

void
bct_mcopy (bct_memb *from, bct_memb *to)
{
      *to = *from;
}
```

Because they must operate atomically on more than one broadcast tree, the implementations of the `bct_get`*n* and `bct_put`*n* functions are broken up into a series of macro calls; this allows the important code sections to be written only once. For performance reasons, macros are used instead of functions.

We begin a detailed presentation of the broadcast tree implementation with `bct_get1`:

```
int
bct_get1 (bct_sync *ts, bct_generic *t1, bct_memb *m1, int *bstring)
{
      int nleaves, // number of leaves
          r,       // return value: number of items found
          sp,      // stak index for fake recursion
          i;       // index of tree node to examine
      spl_t s;
      unsigned int rootval1;    // root time value for t1
      unsigned int mylastcheck1 = m1->lastcheck;
      unsigned int stak[MAXBCTLOG+1];

      if (!_GET_CHECK(t1,mylastcheck1))
            return 0;
```

An initial check, `_GET_CHECK(t1,mylastcheck1)`, may be performed without synchronization:

```
#define _GET_CHECK(t,mylast) (t->tree[0] != mylast)
```

This definition assumes that the root node, `tree[0]`, is read atomically, and compares the time stored in the root node with the time the member joined the group or last called `bct_get`*n*. If the root time has not changed, the caller can be assured that no broadcasts have since taken place. However, if the times no longer match, we must not further trust the exact root time examined; it must be re-read from shared memory after obtaining the group lock:

```
        r = 0;
        nleaves = t1->range;
        s = splsoft();
        (void)bwg_lock (&ts->gl, splcheckf(s));
        rootval1 = t1->tree[0];
        _GET_WORK(t1,mylastcheck1);
        bwg_unlock (&ts->gl);
        vsplx(s);
        _GET_CLEANUP(t1,m1,rootval1,mylastcheck1);
        return r;
}
```

The depth-first tree search will be carried out iteratively by the `_GET_WORK` macro, using an explicit stack, `stak`, which must have room for the largest broadcast tree to be encountered (`MAXBCTLOG` must be defined to log(*largest allowable range*)). The `stak` is indexed by `sp`, while `i` contains the `tree` index of a node to be examined. A push onto the stack can be performed by `stak[sp++] =` *val*, and a pop by *var* `= stak[--sp]`. Prior to obtaining the group lock, we initialize `stak`, `sp`, and `i`, so that `stak` contains the index of one of the root's children (1), and `i` contains the other (2). It will not be necessary to check the root node time again, since we already know it differs from `mylastcheck1`, and it cannot represent an older time than `mylastcheck1`.

The essence of the `bct_get`*n* algorithm is implemented within the first phase of the group lock:

```
#define _GET_WORK(t,mylast)                                    \
        i = 2;                                                 \
        sp = 1;                                                \
        stak[0] = 1;                                           \
        while(1) {                                             \
             while (BCT_GT (t->tree[i], mylast)) {             \
                  if (i >= nleaves-1) { /* leaf node */        \
                       if (!isset(bstring, i-nleaves+1)) {     \
                            setbit (bstring, i-nleaves+1);     \
                            r++;                               \
                       }                                       \
                       break;                                  \
                  }                                            \
                  else {                 /* interior node */   \
                       i = 2*i+1;                              \
                       stak[sp++] = i++;                       \
                  }                                            \
             }                                                 \
             if (sp > 0)                                       \
                  i = stak[--sp];                              \
             else                                              \
                  break;                                       \
        }
```

With current C compiler technology, the nested loops are more efficient than a simpler recursive algorithm. The macro `BCT_GT` evaluates to *TRUE* if the time value represented by the first argument is greater (later) than that of the second argument. The definition is

```
#define BCT_GT(stamp,thresh) ((stamp) && \
                               udiff(stamp,thresh)>0)
```

where `udiff` (§3.5.6/p72) produces the "true" difference between the time values represented by its unsigned integer parameters, even in the face of wraparound, provided that $-2^{wordsize-1} < stamp - thresh < 2^{wordsize-1}$, where *stamp* and *thresh* are the values of `stamp` and `thresh` that would result if the algorithm were run with arbitrary precision integers. For the broadcast tree algorithm, this means we must be sure that no member ever falls as much as $2^{wordsize-1}$ time stamp values behind the most recent broadcast.

To check whether we are meeting this constraint at run-time, we maintain counts, in the `period_count` array, of the members looking for broadcasts in each half of the time value space. This is done by the `_GET_CLEANUP` macro as the final step of `bct_get1` by looking at the most significant bit of `mylastcheck1` and `rootval1`, and adjusting the counts when each member moves from one half to the other:

```
#define _GET_CLEANUP(t,m,rootval,mylast)                        \
      if ((mylast&BCT_MSB) != ((rootval+1)&BCT_MSB)) {          \
              vfad(&t->period_count[(mylast&BCT_MSB)!=0]);       \
              vfai(&t->period_count[((rootval+1)&BCT_MSB)!=0]);  \
      }                                                         \
      m->lastcheck = rootval
```

where `BCT_MSB` is an unsigned integer constant with only the most significant bit set.

The other "get" functions, `bct_get2` and `bct_get3`, operate in a similar fashion. For the sake of brevity, we will present only `bct_get2` and leave `bct_get3` to the reader's imagination:

```
int
bct_get2 (bct_sync *ts, bct_generic *t1, bct_generic *t2,
          bct_memb *m1, bct_memb *m2, int *bstring)
{
      int nleaves,    // number of leaves
          r,          // return value: num items found
          sp,         // stak index for fake recursion
          i;          // index of tree node to examine
      spl_t s;
      unsigned int rootval1, rootval2; // root time for each tree
      unsigned int mylastcheck1 = m1->lastcheck,
                   mylastcheck2 = m2->lastcheck;
      unsigned int stak[MAXBCTLOG+1];
```

```
        if (!_GET_CHECK(t1,mylastcheck1) &&
            !_GET_CHECK(t2,mylastcheck2))
             return 0;
        r = 0;
        nleaves = t1->range;   // must == t2->range
        s = splsoft();
        (void)bwg_lock (&ts->gl, splcheckf(s));
        rootval1 = t1->tree[0];
        rootval2 = t2->tree[0];
        _GET_WORK(t1,mylastcheck1);
        _GET_WORK(t2,mylastcheck2);
        bwg_unlock (&ts->gl);
        vsplx(s);
        _GET_CLEANUP(t1,m1,rootval1,mylastcheck1);
        _GET_CLEANUP(t2,m2,rootval2,mylastcheck2);
        return r;
}
```

The `bct_putn` functions operate from leaf to root; we begin with `bct_put1`:

```
void
bct_put1 (bct_sync *ts, bct_generic *t1, int item)
{
        unsigned int mytime;
        int cookie;
        spl_t s;

        int nleaves = t1->range;
        item = 2*(nleaves+item-1)+1;
```

The `item` parameter is recomputed to index beyond the `tree` array, as a setup condition for the main loop, `_PUT_WORK`, on the next page.

```
        s = splsoft();
        (void)bwg_lock (&t1->gl, splcheckf(s));
        int seq = bwg_fsync1 (&ts->gl, &cookie);
        _PUT_TSTAMP (t1, mytime);
        bwg_fsync2 (&ts->gl, seq, cookie);
        _PUT_WORK(t1, mytime, item);
```

The first phase is used only to get a sequence number for later use and to compute the broadcast time value, based on the root node; this is compatible with the actions of `bct_getn` during the same phase. We use the “fuzzy” version of group lock barrier (§3.5.6/p69): statements after `bwg_fsync2` are not executed until all group participants have executed `bwg_fsync1`.

The definition of `_PUT_TSTAMP` is simple:

```
#define _PUT_TSTAMP(t,mytime)                         \
        if ((mytime = t->tree[0] + 1) != 0)           \
             ;                                        \
        else                                          \
             mytime = 1
```

The effect of this is to wrap around on overflow, skipping zero. The use of an `if` with empty then-part is a crude device to ensure generation of a compiler error message if the macro is invoked without a following semicolon.

The second phase uses the resulting time value to update the tree; this is embodied in the `_PUT_WORK` macro, for which we provide two versions, depending on the quality of hardware support for Fetch&Store:

```
    // if hardware supports ufas at least as well as stores
#if HARD_UFAS && (COMBINE_UFAS >= COMBINE_STORE)
#define _PUT_WORK(t,mytime,item)                              \
        do {                                                  \
             item = (item-1)/2;                               \
             if (ufas (&t->tree[item], mytime) == mytime)     \
                  break;  /* no need to go further */         \
        } while (item > 0)
#else   // not good hardware support for ufas
#define _PUT_WORK(t,mytime,item)                              \
        do {                                                  \
             item = (item-1)/2;                               \
             t->tree[item] = mytime;                          \
        } while (item > 0)
#endif
```

By initializing `item` to point beyond the array, and reducing it at the beginning of the loop, we remove any dependency on the behavior of the expression `(item-1)/2` for loop termination (when `item` is `0`, the result is "implementation defined" in ANSI C:[62] some machines and compilers will give the result `0`, and some will give `-1`). As indicated on page 111, this fragment shows how the algorithm can be adapted to the hardware characteristics of Fetch&Φ operations by means of conditional compilation (§3.1/p43).

The last part of `bct_put1` is concerned with insuring and verifying the safety of time value wraparound:

[62] According to section 3.3.5 of the ANSI C standard [5].

```
        if (_PUT_CHECK (mytime)) {
             seq = bwg_fsync1 (&ts->gl, &cookie);
             _PUT_CAUTION (t1, mytime);
             bwg_fsync2 (&ts->gl, seq, cookie);
             _PUT_ADJ (t1, mytime, item, nleaves);
        }
        bwg_unlock (&ts->gl);
        vsplx(s);
}
```

Wraparound support requires periodically erasing some values from the tree, as discussed on page 113. The _PUT_CHECK macro determines when this is necessary:

```
#define _PUT_CHECK(mytime)                                    \
        ((++mytime & (BCT_MSB-1)) == 0 && seq == 0)
```

It checks whether the *next* broadcast time value will cause a change in the most significant bit of the time value. It has the side effect of incrementing the local variable mytime, which is used again if _PUT_CHECK evaluates to *TRUE*. The additional check for seq==0 serves to skip the remaining wraparound support code in all but one participating processor.

The macro _PUT_CAUTION is merely a diagnostic test for failure:

```
#define _PUT_CAUTION(t,mytime)                                 \
        if (t->period_count[(mytime&BCT_MSB)==0])              \
             ; /* okay */                                      \
        else                                                   \
             panic("bctree member too far behind")
```

We verify that no group member is still waiting to poll for broadcast times that are about to be forgotten. This is considered unlikely, so a call to panic seems adequate.

The call to bwg_fsync2 after _PUT_CAUTION serves to gain exclusive access to the tree, so that any node bearing a time value that will be "reused" during wraparound support is changed to the infinitely old time, 0. In bct_put1, the bwg_fsync*x* calls are rendered unnecessary, though not harmful, by the seq==0 test in _PUT_CHECK, but they are important in bct_put2 and bct_put3.

Finally, the real work of erasing old time values from the tree is performed by the _PUT_ADJ macro:

```
#define _PUT_ADJ(t,mytime,item,nleaves)                           \
      for (item = 1;  item < 2*nleaves-1;  i++)                   \
           if ((t->tree[item]&BCT_MSB) == (mytime&BCT_MSB)) \
                t->tree[item] = 0
```

Similarly with bct_get*n*, the implementation of bct_put2 and bct_put3 is basically an expansion of bct_put1 with a few more variables and additional macro calls. We present bct_put2 and leave bct_put3 to the imagination:

```
void
bct_put2 (bct_sync *ts, bct_generic *t1, bct_generic *t2, int item)
{
      unsigned int mytime1, mytime2;
      int cookie;
      spl_t s;

      int nleaves = t1->range;
      item = 2*(nleaves+item-1)+1;
      s = splsoft();
      (void)bwg_lock (&ts->gl, splcheckf(s));
      int seq = bwg_fsync1 (&ts->gl, &cookie);
      _PUT_TSTAMP (t1, mytime1);
      _PUT_TSTAMP (t2, mytime2);
      bwg_fsync2 (&ts->gl, seq, cookie);
      _PUT_WORK(t1, mytime1, item);
      _PUT_WORK(t2, mytime2, item);
      int adj1 = _PUT_CHECK (mytime1);
      int adj2 = _PUT_CHECK (mytime2);
      if (adj1 || adj2) {
            seq = bwg_fsync1 (&ts->gl, &cookie);
            if (adj1)
                  _PUT_CAUTION (t1, mytime1);
            if (adj2)
                  _PUT_CAUTION (t2, mytime2);
            bwg_fsync2 (&ts->gl, seq, cookie);
            if (adj1)
                  _PUT_ADJ (t1, mytime1, item, nleaves);
            if (adj2)
                  _PUT_ADJ (t2, mytime2, item, nleaves);
      }
      bwg_unlock (&ts->gl);
      vsplx(s);
}
```

3.7.5. Sets

The need for unique small integers arises often enough in practice that it is useful to define functions to assist in their allocation and deallocation. For example, Dimitrovsky's hash table-based parallel queue algorithm [63] requires integer queue identifiers to be dynamically allocated from a restricted range.[63]

[63] One of the algorithms in section 3.7.1 on page 82, `afifo`, is based on this hash table structure. The hash table approach is also a good choice for other variations, such as a FIFO list without the requirement for interior removal.

These and other problems can be solved by using an abstract *set* type along with three operations:

- *Setadd*: add an integer to the set.
- *Setarb*: take an arbitrary integer from the set.
- *Setmemb*: return *TRUE* if a specified integer is currently a set member.

We assume a set initially contains all the integers in a restricted range of interest.

Because the required operations are quite limited (operations such as general union, intersection, etc. are not needed in many cases), and the universe of elements is a single range of integers, we expect to find a solution that is considerably more efficient than general sets.

We consider three approaches to providing at least *setadd* and *setarb*, and then consider some design alternatives and enhancements, such as support for *setmemb*.

Use a General-Purpose List

A good approach is to use one of the ordinary list algorithms presented in section 3.7.1 on page 82, such as `pool`, which is unordered. Initially, all the integers of interest are inserted into the list, so that *setarb* is achieved by a delete and *setadd* is an insert. The complexity of this solution depends on the underlying list algorithm, but presumably requires at least $O(n)$ words of storage when the universe consists of the integers from 1 to n. (See Table 14, on page 83.)

It should be noted that, as a set algorithm, an ordinary list is rather anemic: there is no built-in mechanism to prevent insertion of duplicates. However, if we add support for *setmemb*, which we will consider on page 126, it is easy to prevent duplicates.

Use a Special Algorithm

A general-purpose list is not naturally tailored for use in a set algorithm. Chiabaut [45] designed and implemented a package of special-purpose set routines for Symunix-1, based on a binary tree of counters [156]. The n leaves of the tree represent the range of integers upon which the set is defined. Deletions progress from the root to some leaf, while insertions progress from a leaf to the root, each adjusting the counter values along the way. Assuming n is a power of 2, inserts always require $\log_2 n$ Fetch&Add operations, and deletes require at least that many (up to $3 \log_2 n$ when only right branches are traversed).

The tree is stored as an array, where the children of the node at index i are located at indices $2i$ and $2i+1$, and each node is simply a count of the number of items below that node in the tree that are currently in the set.[64] This requires approximately $2n$ words of shared memory, but the actual code also provides for a more compact representation in which each word of the array contains two nodes, thus reducing the memory usage by a factor of 2 when n is small enough.

With this data structure, *setmemb* is easy to implement: it is merely necessary to consult the counter for the appropriate leaf. It is also easy to accomplish the removal of a specific item from the set, by proceeding from leaf to root, although it is necessary to prevent

[64] The actual code maintains the count as the number of items below the node in the tree that are *not* currently in the set.

setarb from executing at the same time.

Use a Restricted List

The ordinary list algorithms we have adopted for our target class of machines in section 3.7.1 all have an underlying structure based on linked lists; one reason for this is the variable capacity that results. In this section, on the other hand, we are considering a kind of list with a fixed capacity, so a list based on sequential allocation doesn't have a capacity disadvantage. Rather than a special-purpose set algorithm, perhaps we can do better by devising a list algorithm with certain restrictions that improve its suitability for sets. We adopt two restrictions:

- fixed capacity and
- restricted range: at least one integer value must be prohibited from insertion.

In all other respects, this list is as ordinary as those in section 3.7.1. Note, we are not ready to show *setmemb* support yet; that will come on page 126.

Implementation: Overview. The following C code implements such an algorithm; it was inspired by the queue algorithm of Gottlieb *et al.* [94], but resolves cell contention by use of Fetch&Store, requires at least one special data value, and is not FIFO.[65] For purposes of presentation, we assume the restricted value is 0, and the set operates on integers from 1 to n.

The basic idea is to use an array with *head* and *tail* pointers updated in a circular fashion. Each cell in the array contains a value in $\{1,...,n\}$ or 0, to indicate an "empty" location. The idea is to use Fetch&Add with *head* or *tail* to choose a cell in the array, then use Fetch&Store to perform the insert or delete, repeating the operation as necessary when collisions occur. An insert uses Fetch&Store(*cell*,*value*) and is complete when Fetch&Store returns 0; otherwise the returned value must be reinserted (at the same location). Likewise, a delete is complete when Fetch&Store(*cell*,0) returns a positive value; otherwise it must be repeated. In this way, cell contention is more tightly integrated with the insertion and deletion operations than with general-purpose list algorithms.

Herlihy and Wing [107] have a concurrent FIFO queue algorithm that is similar to this, in that it uses Fetch&Store for the delete operation and a special value for empty cells. They have no cell contention mechanism in the sense proposed by Gottlieb *et al.* [94], so their queue requires an array large enough for all inserts (i.e., the array never "wraps around"). Execution time differs as well; their insert requires $O(1)$ instructions, but their delete requires $O(p)$, where p is the number of previously completed deletes, even in the absence of contention (the number of instructions is unbounded in the presence of contention). In contrast, the set algorithm we present here has the same time requirement for both insert and deletion: $O(1)$ in the absence of contention, but potentially unbounded time otherwise. These differences reflect a different purpose: their algorithm was designed to be "non-blocking" and provably correct, while ours was designed to be compact and fast in the common case, even in the presence of much concurrency.

---

[65] This algorithm was originally described by this author in [72], and was subsequently studied by Wood [207].

Implementation: Details. For the reasons outlined in section 3.7.4 on page 109, we define a macro to describe the data structure for a set capable of handling integers from 1 to *n*:

```
// set type
#define SET_DECL(n) struct {                          \
      int max;      // largest array index (== n-1)   \
      int size;     // current set size               \
      unsigned head;                                  \
      unsigned tail;                                  \
      int q[n];                                       \
}
```

We define a specific generic type, to and from which pointers to other sets can be cast:

```
typedef SET_DECL(1) set_generic;
```

and then define functions in terms of that.

```
// initialize a new set to hold integers from 1 to n
int
setinit (set_generic *s, int n)
{
      if (n <= 0 || (n&(n-1)) != 0)
            return NULL;   // require n be a positive power of 2
      s->max = n-1;
      s->size = n;
      s->head = 0;
      s->tail = 0;
      while (n--)
            s->q[n] = n+1;
      return 1;            // success
}

// add an element to a set
// no test for prior membership is made; attempt to add 0 is ignored
void
setadd (set_generic *s, int val)
{
      int *ip;

      if (val == 0)
            return;
      ip = &s->q[ufai (&s->tail) & s->max];
      BUSY_WAIT_NO_INST ((val = fas (ip, val)) == 0, nullf);
      vfai (&s->size);
}
```

`BUSY_WAIT_NO_INST` is defined in section 3.5.1 on page 52.

```
// get an arbitrary element from a set, or 0 if set is empty
int
setarb (set_generic *s)
{
      int *ip;
      int val;

      if (!tdr1 (&s->size))
            return 0;
      ip = &s->q[ufai (&s->head) & s->max];
      BUSY_WAIT_NO_INST ((val = fas (ip, 0)) != 0, nullf);
      return val;
}
```

In this code, we have restricted the set capacity to a power of 2; some changes are required to handle overflow of `head` and `tail` if this restriction is to be removed.

This list structure is somewhat interesting in its own right, as it is can be extremely efficient: the cost of either `setadd` or `setarb` is only 4 shared memory accesses in the common special case of no cell contention. Furthermore, one of those memory accesses is just to fetch `s->max`, which is effectively a constant, and another is to maintain `s->size`, which higher-level logic might render unnecessary in some situations. So in some cases, the algorithm might be tailored to have a cost as low as 2 shared memory accesses per operation.

Although this list algorithm is not FIFO in general (e.g., Fetch&Store can reverse the order of inserts), it is FIFO in the absence of cell contention. This "mostly FIFO" property is similar to that described in section 3.7.1 on page 82, although this algorithm is starvation prone. The probability of an item being "starved", i.e., left to languish in the set, decreases as the number of other items deleted increases, so it is not as bad as a stack (LIFO) organization.

Another interesting property is that, although the list's capacity is explicitly fixed, it isn't possible for an insert to fail because of overflow. The storage capacity of the list is inherently extended to include the `val` variables of the inserting processes, possibly together with various hidden data paths of the machine. Indeed, the array size can be reduced to 1, and the algorithm still "works", regardless of the number of processors involved. Of course, performance will suffer, as all but one inserter will be busy-waiting. In such a degenerate case, any semblance of FIFO ordering disappears. Deadlock can occur if all processors get involved in such inserts.

This kind of structure can be trivially extended to support both insertion and deletion from either end, i.e., to be a deque (double-ended queue; see Knuth [130]). The value of such an extension may be low, however, considering the lack of true FIFO ordering in the basic algorithm.

A variant of this algorithm is to use Fetch&Store-if-0 (§3.1/p41) when constructing the busy-waiting condition for `setadd`, as follows:

original: `(val = fas (ip, val)) == 0`
alternate: `fase0 (ip, val) == 0`

This may have some advantages, but still does not guarantee FIFO behavior.

Incorporating a Bitmap

By using a bitmap together with a general or restricted list algorithm, we can add support for the *setmemb* operation and even save memory; the cost is some additional memory accesses for *setarb* and *setadd*.

Simple Approach. The simplest approach is to use a list algorithm (such as that given on page 124, but almost any will do) together with a bitmap of n bits representing the integers from 1 to n. A bit is 1 if the corresponding integer is in the set, and 0 if it isn't. Clearly, this approach doesn't save any memory (we will see how to do that below), but it is extremely simple. The set operations are implemented in the following manner:

- *Setadd*: use Fetch&Or to set the appropriate bit in the bitmap and (if Fetch&Or shows that the bit was previously 0) then insert into the list.
- *Setmemb*: check the appropriate bit in the bitmap.
- *Setarb*: perform an ordinary deletion from the list and then use Fetch&And to clear the appropriate bit in the bitmap.

This algorithm takes more memory and runs slower than the one on page 124, but it supports *setmemb* and avoids both duplicates and deadlock.

A Compact Approach. We can improve memory utilization by regarding the bitmap as an integer array and reducing the list size by a factor of w, the number of bits in an integer (word). The basic operations can be implemented as follows:

- *Setadd*: use Fetch&Or to set the appropriate bit in the bitmap and (if Fetch&Or shows that the whole bitmap word was previously 0) then insert the word's bitmap index into the list.
- *Setarb*: perform an ordinary deletion from the list to get the bitmap index of a word containing at least one set bit. Examine the word and use Fetch&And to clear a bit; this gives the result for *setarb*. If Fetch&And shows that the word is still nonzero, reinsert the bitmap index into the list.
- *Setmemb*: check the appropriate bit in the bitmap.

The parallelism of *setarb* is reduced by w with this approach.

3.7.6. Search Structures

This section differs from others of this chapter in that it presents not a packaged abstract data type, but only a general algorithmic technique. This is because there are so many significant variations to the basic algorithm, that attempts to abstract and package it tend to produce a different package for each use or a single package with intolerably many options.

We specifically excluded any kind of searching, traversal, or arbitrary insertion operations from the lists of sections 3.7.1 and 3.7.2. In section 3.7.3, we described visit lists, which support a traversal operation, but we have not yet dealt with the important problem of searching. True, one could perform a search by using visit lists, but the synchronization required would be awkward at best and the general approach of examining all items on the list requires too much brute force to be practical.

What we describe in this section is a general data structure and algorithm supporting two basic operations:

(1) *Search* for an item, given a *key*. If not found, allocate, initialize, and insert a new item with the key. Return a pointer to the item, after incrementing its reference count.

(2) *Release* an item previously returned by *search*. Decrement its reference count and destroy it when there are no more references.

This rough specification fits many situations in operating systems design quite well. Our design is based on a hash table, with the usual advantages and disadvantages thereof:

- The average number of items examined in a search is $O(1)$, but the worst case is $O(N)$ for a hash table containing N items.
- The hash table size must be determined in advance, unless we are willing to consider an extensible hashing scheme, such as in the hq algorithm described by Wood [207]. We don't consider such algorithms further here, because of their additional complexity and dynamic storage management needs.

We handle hash collisions by putting such items on linked lists associated with the hash buckets. The lists are protected by readers/writers locks (§3.5.4). The essential parallelism of the algorithm comes from three sources:

(1) The fundamental concurrency of hashing, which only requires synchronization on collision. An effective hash function will distribute most unrelated operations to distinct buckets, where they won't interfere with one another.

(2) When the parallelism of (1) can't help, optimistic use of reader locks avoids serialization whenever the *search* operation finds the sought-for key. This is true whether bucket list contention results from coincidental collisions or concurrent accesses of the same object.

(3) When the parallelism of (2) can't help because an item must be allocated, initialized, and inserted into the hash table, careful use of the upgrade operation, from reader to writer (§3.5.4/p63), allows processors to cooperate so the total serialization is independent of the number of contenders. This effect is similar to that achieved with the group lock (§3.5.6), but the synchronization is weaker and hence cheaper.

When many concurrent *search* operations conflict, no serialization is required unless the search fails. In that case, one contender performs the serial work of allocating, initializing, and inserting a new item while the others wait, then they all proceed without further serialization. There is no problem with many concurrent *release* operations conflicting, because serial work is only performed by the last one. A combination of concurrent conflicting *search* and *release* operations requires extensive serialization only in the case where the reference count oscillates rapidly between zero and positive numbers; this problem appears to be fundamental to the general specification given above.

This is essentially the algorithm used in Symunix-1 for managing file system i-node structures in memory. Although oscillation such as we have just described is rare in practice, it can be seen under some workloads (e.g., §8.3.2/p350). We outline a more elaborate search structure that avoids this problem on page 136.

Data Structures

There are three data structures needed for this algorithm:

- *The items.* For the sake of exposition, we will assume this definition:

```
typedef struct _searchitem {
     struct _searchitem *next; // linked list pointer
     int key;                  // search key
     int refcnt;               // reference count
} searchitem;
```

  Of course, the names `searchitem` and `_searchitem` may not good choices in practice. The `searchitem` structure should be expanded to include other data of greater interest. Certainly the `key` need not be restricted to an integer.
- *The hash table.* This is an array of buckets, each being the head of a linked list. We use the simplest such structure possible:

```
searchitem *hash[NSEARCHHASH];
```

- *The readers/writers locks.* The number of such locks can be chosen as a configuration parameter, similar to, but independent of, the hash table size. For the sake of generality, we assume there are `NRW_SEARCHHASH` such locks, and stipulate that $0 <$ `NRW_SEARCHHASH` $\leq$ `NSEARCHHASH`. If `NRW_SEARCHHASH` $=$ `NSEARCHHASH`, the two arrays can be merged into one (as in Symunix-1) but, for the sake of generality, we assume a separate array:

```
bwrwlock rwhash[NRW_SEARCHHASH];
```

  The advantage of having `NRW_SEARCHHASH` $<$ `NSEARCHHASH` is space savings. It seems likely that some such savings can be realized without significant sacrifice of parallelism.

The two hash tables are independent; it is even possible to share the `rwhash` table among multiple search hash tables used for entirely different purposes. However, we will assume the two arrays belong together:

```
typedef struct {
     searchitem *hash[NSEARCHHASH];
     bwrwlock rwhash[NRW_SEARCHHASH];
} searchtab;
```

These data structures should be initialized in the obvious manner. The hash table's pointers should be set to `NULL`, as in

```
for (i = 0;  i < NHASH;  i++)
      stab->hash[i] = NULL;
```

The items' `next` pointers should be initialized to point at themselves (not `NULL`; we use `NULL` to mean "end of list" and a pointer to self to mean "not on any list"; we will make use of this on page 140).

One detail we have omitted is management of unallocated `searchitem` structures; perhaps the `next` field within `searchitem` should be contained in a `union` with a `poolitem` (§3.7.1/p82); for brevity, we will gloss over such details here.

Algorithm

We need two hash functions; we will choose the most obvious ones:

```
#define SEARCHHASH(key)        ((key)%NSEARCHHASH)
#define RW_SEARCHHASH(hindex)  ((hindex)%NRW_SEARCHHASH)
```

where `hindex` is a hash index, 0 ≤ `hindex` < `NSEARCHHASH`.

The basic search function is:

```
searchitem *
search (searchtab *stab, int key)
{
     int hbuck = SEARCHHASH(key);
     int hrw = RW_SEARCHHASH(hbuck);
     searchitem *sitem;

again:
     spl_t s = vsplsoft();
     bwrw_rlock (&stab->rwhash[hrw], splcheckf(s));
     for (sitem = stab->hash[hbuck];
            sitem != NULL;
            sitem = sitem->next) {
          if (sitem->key == key) {
               vfai (&sitem->refcnt);
               Additional acquisition steps;
               bwrw_runlock (&stab->rwhash[hrw]);
               vsplx(s);
               Final acquisition steps;
               return sitem;
          }
     }

     // item not found
     if (!bwrw_rtow (&stab->rwhash[hrw])) {
          // upgrade failed; start over
          bwrw_runlock (&stab->rwhash[hrw]);
          vsplx(s);
          goto again;
     }
```

```
        // upgrade succeeded
        sitem = get new item;
        if (sitem != NULL) {
             sitem->key = key;
             sitem->refcnt = 1;
             Additional initialization steps;
             sitem->next = stab->hash[hbuck];
             stab->hash[hbuck] = sitem;
        }

        bwrw_wunlock (&stab->rwhash[hrw]);
        vsplx(s);
        if (sitem != NULL)
             Final initialization steps;
        return sitem;
   }
```

The reader lock prevents the set of items in the bucket list from changing. As soon as `sitem->refcnt` is incremented (or set to 1), the item will not be deallocated, even after the lock is released.

Before going on to present the release algorithm, we will discuss two variations on the search algorithm. These variations have to do with process blocking (context-switching) for long delays, a subject generally reserved for chapter 4.

Time Consuming Initialization. In some cases initializing an item can be time consuming. For example, when managing UNIX i-node structures in memory, initialization includes reading the i-node from disk. Holding the busy-waiting writer lock during that time is impractical, so we put a context-switching readers/writers lock in each item (§4.5.2). We use this lock in `search` at three points:

Additional initialization steps
: This is where we can't afford to perform expensive initialization steps, such as I/O. But since we have just allocated the item in question, we can be assured of obtaining an exclusive lock on it without blocking; this will prevent any other process from gaining significant access to the item until we complete initialization, after releasing the `rwhash` lock. All that is required before releasing the `rwhash` lock is setting up the key, reference count, and linked list pointers, so the item can be properly found.

Final initialization steps
: This is where the I/O or other expensive initialization goes. When complete, the exclusive item lock can be released.

Final acquisition steps
: At this point, we can obtain either a shared or an exclusive item lock, blocking as necessary. In Symunix-1, where the function corresponding to `search` is `iget`, the choice is based on an additional function parameter.

No action is required at the point indicated for *additional acquisition steps*.

Time Consuming Allocation. In many cases, including Symunix-1 i-node management, allocation of new items can take place from a pool (§3.7.1/p82), and an empty pool can be treated as an error condition returned to the caller (as we provide for in `search`, which will return `NULL`).[66] An alternative in some situations is to wait for an item to become available[67] or to take action to find one, such as looking for an unused but allocated item to reallocate (see page 136). In case item allocation is time-consuming, a *dummy* item can be used temporarily.

It is most efficient, if possible, to allocate an item without waiting and proceed as we have already outlined. If a delay is necessary, a dummy item structure can be allocated from a dedicated pool, or pre-allocated on a per-process basis, or preferably allocated as an automatic variable on the stack when `search` is called. A dummy item must be distinguishable from a regular item; a flag in another field of the item or an otherwise impossible value for some field are reasonable mechanisms for making the distinction. The dummy item need not be initialized, except for `key`, `refcnt`, `next`, and the aforementioned dummy-detection mechanism. Even if a context-switching synchronization structure is included in the item as described on the previous page, it need not be initialized within the dummy, because it will not be accessed.

We need a way to wait for a dummy to be replaced by a "real" item; this is the subject of section 4.5, but we will take a leap and assume the existence of a context-switching *event* mechanism, `csevent`, with operations `cse_wait`, `cse_signal`, and `cse_reset`. Again we will use hashing, and use an array of these events:

```
csevent dummy_hash[NDUMMY_HASH];
#define DUMMY_HASH(key)   ((key)%NDUMMY_HASH)
```

This array could be globally shared between all `searchtab` structures, but in this exposition we will assume it is added to each.

When a dummy is being used, we must augment `search` at the following four points:

Additional initialization steps

The dummy detection flag, or other suitable mechanism, must be set. The dummy event must be reset, with

```
cse_reset (&stab->dummy_hash[DUMMY_HASH(key)]);
```

(we assume this can be done in constant or near-constant time).

Final initialization steps

The expensive allocation must be performed and, when the real item is available, the dummy item replaced and the new item initialized, with `rwhash[hrw]` locked exclusively. (If allocation should fail at this point, the dummy item must be removed rather than replaced.) Finally, the dummy event must be signaled, with

---

[66] In this case, Symunix-1 also prints the traditional message, `i-node table overflow`, on the system console.

[67] But watch out for deadlock.

```
cse_signal (&stab->dummy_hash[DUMMY_HASH(key)]);
```

to wake up any other processes waiting for the same item.

Additional acquisition steps

Dummy items must be detected at this point and remembered until the *final acquisition steps*. It is crucial not to reference the dummy after releasing the `rwhash[hrw]` lock. It is not necessary to increment a dummy's reference count, but not harmful either, beyond the wasted memory access.

Final acquisition steps

When dealing with a dummy (detected before releasing the `rwhash[hrw]` lock), we must wait for the dummy to be replaced, by calling

```
cse_wait(&dummy_hash[DUMMY_HASH(key)]);
```

and then restart the search algorithm in case of `dummy_hash` collisions, by using `goto again`.

Because the events are hashed, it is possible for multiple dummy situations to interfere with each another. Indeed, a single event may be adequate if delayed allocation is rare enough. Such interference can cause some processes calling `cse_wait` to return prematurely, and some to wait longer than necessary, but none will wait indefinitely.

The techniques we have described for handling time consuming allocation and initialization may be combined and used together.

Release. We now present the release algorithm, which decrements an item's reference count and deallocates the item when it reaches zero:

```
void
release (searchtab *stab, searchitem *sitem)
{
     int key = sitem->key;  // may be needed after fad
     if (fad (&sitem->refcnt) > 1)
          return;  // done
     // sitem may be invalid pointer now
     int dealloc = 0;
     int hbuck = SEARCHHASH(key);
     int hrw = RW_SEARCHHASH(hbuck);
     searchitem *sp, **spp;
     spl_t s = splsoft();
     bwrw_wlock (&stab->rwhash[hrw], splcheckf(s));
     for (spp = &stab->hash[hbuck], sp = *spp;
             sp != NULL;
             spp = &sp->next, sp = *spp) {
          if (sp == sitem || sp->key == key) {
               if (sitem == sp &&
                   sp->key == key && sp->refcnt == 0) {
                    Un-initialization steps;
                    *spp = sp->next;// remove from list
                    sp->next = sp;  // indicate not on any list
                    dealloc = 1;
               }
               break;
          }
     }
     bwrw_wunlock (&stab->rwhash[hrw]);
     vsplx(s);
     if (dealloc)
          Deallocate *sitem;
}
```

We optimistically decrement the item's reference count without any locks. As soon as the count goes to zero, the item pointer becomes invalid; another processor could execute both `search` and `release`, possibly reassigning the old item structure. Although possible, such reassignment is unlikely, so we take an exclusive `rwhash` lock, assuming we will find the item undisturbed in the bucket, waiting to be deallocated. We complete the deallocation only if three conditions are met:

(1) We must find the very same item.

(2) It must have the same key.

(3) It must have `refcnt` equal to 0.

Along the way, if we find the same item or the same key without meeting the other two conditions, we can stop searching, safe in the knowledge that some one else found the item and began using it.

It is unfortunate that we must search the hash bucket in `release`; we can avoid the search at the cost of additional locking in the common case, and a retry loop:

```
void    // alternate inferior version
release (searchtab *stab, searchitem *sitem)
{
      int hrw = RW_SEARCHHASH(SEARCHHASH(sitem->key));
again:
      spl_t s = splsoft();
      bwrw_rlock (&stab->rwhash[hrw], splcheckf(s));
      if (fad (&sitem->refcnt) > 1) {
           bwrw_runlock (&stab->rwhash[hrw]);
           vsplx(s);
           return;      // done
      }
      if (!bwrw_rtow (&stab->rwhash[hrw])) {
           vfai (&sitem->refcnt);
           bwrw_runlock (&stab->rwhash[hrw]);
           vsplx(s);
           goto again;  // try again
      }
      // refcnt is 0 and we have exclusive bucket lock
      Un-initialization steps;
      Remove item from bucket linked list;
      bwrw_wunlock (&stab->rwhash[hrw]);
      vsplx(s);
      Deallocate *sitem;
}
```

There are several reasons for regarding this version of `release` as inferior:

- It can suffer from indefinite delay; there is no limit to the number of times we can `goto again` before completing `release`. On the other hand, our `search` algorithm can also suffer from this problem.
- If we assume deallocation is uncommon, this version costs more because it locks the hash bucket every time.
- Avoiding the hash bucket search requires use of a doubly-linked list, increasing the space requirements of both the hash table and items, as well as increasing the cost of list operations by requiring additional memory references.

Time Consuming Un-Initialization. Time consuming un-initialization is a problem for `release`, just as allocation and initialization were problems for `search`. Consider the problem of managing i-nodes in memory: when no longer needed, they must be written back to the disk. But such write-back can't be done at the point indicated for *Un-initialization steps* in the code for `release`, because context-switching might be required, and we cannot allow context-switching while we hold the busy-waiting bucket list lock. Write-back can't be done *before* obtaining the bucket list lock, because the item could become dirty again by the time we finally get the lock, and it can't be done *after* removing the i-node from the hash list, because a subsequent `search` might read stale data from the disk before the write-back is complete.

To solve this problem, we describe an extended version of `release`, slightly improved from that used in Symunix-1 and reported in Figures 35 and 36 in section 8.3.2, on pages

350–351.

As with time consuming initialization (p130), we assume each item is augmented with a context-switching readers/writers lock, which must be held exclusively for un-initialization. We also assume un-initialization can be viewed as dealing with "dirty" data: if the item is not marked dirty, no time-consuming un-initialization is required. While an exclusive item lock is held, a clean item will remain clean as long as no explicit action is taken to dirty it.

Given the existence of item locks, the first issue `release` must address is the item lock state: does the caller hold a shared lock, an exclusive lock, or no lock on the item? This is most easily answered by adding a three-valued parameter, `locktype`, to `release`. If `refcnt` is 1, it makes sense to go ahead and acquire an exclusive item lock right away, if not already held, because it will probably be needed for un-initialization.

```
          // enhanced for time consuming un-initialization
          // locktype=0: no item lock; 1: exclusive; 2: shared
void
release (searchtab *stab, searchitem *sitem, int locktype)
{
      if (locktype != 1 && sitem->refcnt == 1 &&
           Get exclusive or upgrade item lock successful)
            locktype = 1;
      int dealloc = 0, doagain = 0;
      int key = sitem->key;
```

The general strategy is to decrement the reference count. If it doesn't go to zero, we are done. Otherwise, if we don't have an exclusive lock, we must find the item in the bucket, increment the reference count, get an exclusive lock, and start over:

```
again:
      if (fad (&sitem->refcnt) == 1) {
            if (locktype == 1)
                  Time consuming un-initialization;
            Identify and lock bucket;
            if (locktype == 1) {
                  Remove item from bucket list;
                  dealloc = 1;
            }
            else {
                  Search for item in bucket;
                  if (sitem found in bucket &&
                       key == sitem->key && sitem->refcnt == 0) {
                        vfai (&sitem->refcnt);
                        doagain = 1;
                  }
            }
            Unlock bucket;
            if (doagain) {
                  Get exclusive or upgrade item lock;
                  locktype = 1;
                  goto again;
            }
      }

      if (locktype != 0)
            Unlock item;
      if (dealloc)
            Deallocate *sitem;
}
```

Expensive un-initialization steps are performed on an item with a zero reference count but with an exclusive item lock. This is safe, because our lock prevents the item from being deallocated (i.e., by another process calling `search` and `release`).

If the bucket list is doubly-linked, removal (at the point specified by *Remove item from bucket list*) can be done without searching, otherwise a search is needed, as in the version of `release` on page 133.

Searching and LRU Cacheing

As described on page 127 and exhibited in section 8.3.2 on page 350, excessive overhead can result when reference counted objects oscillate rapidly between counts of zero and positive numbers. While this is a fundamental problem with reference counted objects, it can be essentially eliminated by combining reference count management with an LRU replacement strategy so that objects are not deallocated immediately when their reference counts drop to zero.

The idea is to use search and release algorithms very similar to those presented on pages 129 and 133, but also to reference an LRU list (§3.7.2). When an object's reference count drops to zero, it remains on the search structure while being added to the LRU list. If such an object is subsequently searched for and found, it can be *reclaimed* by incrementing the reference count and removing it from the LRU list. Alternatively, another search request needing to allocate space may obtain the object from the LRU list and *reassign* it's data structure by removing it from the search structure and reinserting it under a different key. A race exists between searches needing to reclaim objects from the LRU list, and those needing to reassign the least recently used object. We resolve this race by deferring to the search structure: before an object can be reassigned, "approval" must be secured from the search mechanism, but a reclaim is regarded as merely advisory.

We will present three operations:

- *Cachesearch*: Like plain search, but failed searches are satisfied by reassigning objects not recently used. This eliminates the need for a separate pool for unallocated object structures. (As presented here, we assume a fixed total number of object structures, but the algorithm is readily extended to allow a changing number.)
- *Cacherelease*: Like plain release, but objects may be reclaimed by *cachesearch*.
- *Cachefree*: Like *cacherelease*, but the object cannot be subsequently reclaimed. This is useful when an object is to be destroyed (i.e., when the old data should not be found by a *cachesearch* with the old key).

Cache Data Structures. For purposes of exposition, we will call this combined mechanism a `cache`, and each object will be represented by a `cacheitem`:

```
typedef struct {
     searchtab stab;
     lrulist lru;
} cache;

typedef struct {
     searchitem si;
     lruitem li;
} cacheitem;
```

Initially, all items should be placed on the LRU list and none should be on the search list.

Cache Search. This is based closely on the `search` function presented on page 129:

```
cacheitem *
cachesearch (cache *c, int key)
{
     int hbuck = SEARCHHASH(key);
     int hrw = RW_SEARCHHASH(hbuck);
     cacheitem *citem;
     searchitem *sitem;
```

```
again:
     spl_t s = vsplsoft();
     bwrw_rlock (&c->stab.rwhash[hrw], splcheckf(s));
     for (sitem = c->stab.hash[hbuck];
            sitem != NULL;
            sitem = sitem->next) {
          citem = (cacheitem *)((char*)sitem -
                       offsetof(cacheitem, si));
          if (sitem->key == key) {
               int rc = fai (&sitem->refcnt);
               Additional acquisition steps;
               bwrw_runlock (&c->stab.rwhash[hrw]);
               vsplx(s);
               if (rc == 0) // advise lru list
                    lruremove (&c->lru, &citem->li);
               Final acquisition steps;
               return citem;
          }
     }

     // item not found
     if (!bwrw_rtow (&c->stab.rwhash[hrw])) {
          // upgrade failed; start over
          bwrw_runlock (&c->stab.rwhash[hrw]);
          vsplx(s);
          goto again;
     }
```

```
        // upgrade succeeded
        do {
             lruitem *litem = getlru (&c->lru);
             if (litem == NULL)
                  citem = NULL;  // failed
             else {
                  citem = (cacheitem*)((char*)litem -
                          offsetof (cacheitem, li));
                  if (_cache_reassign(c, citem, hrw)) {
                       litem = NULL; // success
                       citem->si.key = key;
                       citem->si.refcnt = 1;
                       Additional initialization steps;
                       citem->si.next = c->stab.hash[hbuck];
                       c->stab.hash[hbuck] = &citem->si;
                  }
             }
        } while (litem != NULL);
        bwrw_wunlock (&c->stab.rwhash[hrw]);
        vsplx(s);
        if (citem != NULL)
             Final initialization steps;
        return citem;
  }
```

The allocation portion of `cachesearch` is tricky, but most of the trickiness is buried inside the internal function `_cache_reassign`. The job of `_cache_reassign` is to remove an item from whatever search hash bucket list it is on, if possible. The candidate's reference count must be consulted while an exclusive lock is held on the hash bucket list. The need for a lock raises the possibility of deadlock, because at this point in `cachesearch`, we are still holding an exclusive lock. We will describe two deadlock avoidance solutions, and implement the first:

(1) For a conservative approach, we can implement `_cache_reassign` to use conditional locking, i.e., `bwrw_trywlock` instead of `bwrw_wlock` (§3.5.4/p64). But first, it must check to see if the reassignment candidate is on a bucket list protected by the readers/writers lock already held, the one with index `hrw`; if so, no additional lock is required:[68]

[68] Checking if the two buckets are the same could be omitted, but would result in some unnecessary failures, lowering the algorithm's efficiency in both space and time. Furthermore, in the degenerate case where `NRW_SEARCHHASH` is 1, no reassignments would ever occur.

```
int
_cache_reassign (cache *c, cacheitem *citem, int hrw_held)
{
     searchitem *sp, **spp;
     if (citem->si.next == &citem->si)
          return 1;  // item not in any bucket
     int hbuck = SEARCHHASH(citem->si.key);
     int hrw = RW_SEARCHHASH(hbuck);
     if (hrw != hrw_held &&
         !bwrw_trywlock (&c->stab.rwhash[hrw]))
          return 0;  // avoid deadlock; assume the worst

     for (spp = &c->stab.hash[hbuck], sp = *spp;
            sp != NULL;
            spp = &sp->next, sp = *spp) {
          if (sp == &citem->si) {
               if (citem->si.refcnt != 0)
                    sp = NULL;
               else {
                    *spp = sp->next;
                    sp->next = sp;  // not on any list
               }
               break;
          }
     }

     if (hrw != hrw_held)
          bwrw_wunlock (&c->stab.rwhash[hrw]);
     return sp != NULL;
}
```

(2) Another approach is to adopt the solution outlined on page 131 for time consuming allocation. If `_cache_reassign` can't obtain the needed `rwhash` lock without delay, a dummy item can be used temporarily. When we get to *Final allocation steps*, we can get a lock without concern for deadlock. After determining reassignment, we relock the original `rwhash` lock and replace or remove the dummy item.

This approach might have merit for another reason: it limits the amount of time the `rwhash` locks are held.

Cache Release and Free. The basic idea is to decrement the reference count, and call `putmru` when it drops to zero. We handle both `cacherelease` and `cachefree` with the same function:

```
#define cacherelease(c,ci)  _cacherel(c,ci,0)
#define cachefree(c,ci)     _cacherel(c,ci,1)
```

```
void
_cacherel (cache *c, cacheitem *citem, int makefree)
{
     int key = citem->key;  // may be needed after fad
     if (fad (&citem->si.refcnt) > 1)
          return;  // done
     // citem may be invalid pointer now
     int hbuck = SEARCHHASH(key);
     int hrw = RW_SEARCHHASH(hbuck);
     searchitem *sp, **spp;
     spl_t s = splsoft();
     bwrw_wlock (&c->stab.rwhash[hrw], splcheckf(s));

     for (spp = &c->stab.hash[hbuck], sp = *spp;
            sp != NULL;
            spp = &sp->next, sp = *spp) {
          if (sp == &citem->si || sp->key == key) {
               if (sp == &citem->si &&
                    sp->key == key && sp->refcnt == 0) {
                     if (!makefree)
                          putmru (&c->lru, &citem->li);
                     else {
                          *spp = sp->next;
                          sp->next = sp;  // not on any list
                          putlru (&c->lru, &citem->li);
                     }
               }
               break;
          }
     }
     bwrw_wunlock (&c->stab.rwhash[hrw]);
     vsplx(s);
}
```

3.8. Future Work

There is a need for more experience with these and other highly parallel algorithms. We need to understand their performance better under real workloads and learn how well they may be applied to different kinds of operating systems. The study by Wood [207] and the data in section 8 are only a start.

Busy-Waiting Synchronization

In section 3.5, we used an exponential backoff technique to reduce memory contention and interconnection network traffic while busy-waiting. Many busy-waiting algorithms are probabilistically fair with simple busy-waiting, but exponential backoff destroys this: processors poll at a lower frequency after waiting a long time. It might be useful to consider other forms of exponential backoff to regain fairness. One possibility is to use a ticket-based algorithm: this could be fair without regard to polling frequency, but might waste a lot of time since the senior waiter is always polling at the lowest frequency; somehow the opposite order should be used. Another approach might be to find a way to notify each new waiter of the "oldest" waiter's age; the new waiter can then use the same slow polling frequency, restoring probabilistic fairness to the non-ticket-based algorithm.

Broadcast Trees

It is possible to extend the algorithm to count broadcasts, so nothing is lost. Each leaf in the tree would be augmented with a count of the total number of broadcasts of the corresponding type. Each member would also remember the counts for all the leaves. This will consume a lot of memory (exponential in the number of processors) if the number of broadcast types and the number of members both grow with the size of the machine, but it seems more likely that the number of broadcast types will remain relatively fixed.

Since our initial application (delivery of traditional UNIX signals to process groups) had no need for counting, we have not fully investigated such an extension. There are some questions that arise immediately:

- How do we handle overflow?
- How well does the extended algorithm perform?
- How many broadcast types are needed for various other applications?

A good first step would be to identify specific worthy applications.

Hardware for Broadcast Interrupts

Two possible hardware enhancements are immediately apparent. The first was already mentioned in section 3.6 on page 77: the hardware could count RBC interrupts so that the soft RBC interrupt handler can know exactly how much work to do. Without this information, it is difficult to balance interrupt level RBC work fairly among the processors.

The second enhancement we propose is to support random broadcast directly in the hardware. Machines in our target class generally have some sort of interconnection network for processor–memory traffic. We want this network to provide a broadcast message capability. For concreteness, let us consider an Omega network composed of 2×2 switches. Normal messages go from one input port to one output port of each switch. In contrast, a full broadcast message would go from one input port to both output ports. The Ultra-3 prototype [29] supports such a full broadcast operation (as part of the "return path" of the network).

A partial broadcast message would not behave the same at every switch. Sometimes it would go to two output ports, like a full broadcast message, and sometimes it would go to only one, like a normal message. There are several possible ways of controlling this, and they differ in their suitability for different switch configurations. Some possibilities worthy of further study include:

- *Full bitmap.* If enough bits are available in the right place within a message, it is possible to specify exactly the set of processors to receive a broadcast interrupt message. The ability to do this is also limited by details of the routing mechanism used by the switches for normal messages. For example, the processor bitmap may need to be fully contained in the first packet of a message. In addition, each switch in the network needs to know exactly which bits to examine for routing.
- *Partial bitmap.* A partial bitmap could be provided in a broadcast message to control broadcast behavior at only certain stages of the network. For example, the first $\log n$

stages could be controlled by the n-bit bitmap, and the remaining stages could perform a full broadcast. This would allow a broadcast initiator to specify which *groups* of processors should be recipients.

- *Variable-Stage Splitting.* With only $\log N$ bits for an N processor machine, one could specify at which network stages broadcasting (or "splitting") should occur. The other stages could be controlled by the normal message routing mechanism (also using $\log N$ bits).
- *Balanced random.* A random broadcast message should contain a count of how many processors are to receive it. Each switch handling such a message should send it to both output ports. Along the way, the count is right shifted by one bit position, and the bit shifted out is added back into *only one* of the outgoing messages; the choice should be random. If one of the outgoing messages ends up with a count of zero, it may be discarded. It is easy to see that this scheme distributes the recipients in a balanced fashion across the machine.

There are many engineering issues to be considered in the possible implementation of these partial broadcast schemes. We need to know the cost and performance of each, the impact on normal (non-broadcast) traffic, the frequency of use, and the distribution of broadcast counts in practice.

So far we have discussed broadcast as an operation in the interconnection network only for purposes of interrupting processors. Another possible extension is to use it for accessing memory. In this case, the random aspect doesn't seem too useful, but both full and partial broadcast can have practical applications. We briefly discuss a few possibilities:

- *Filling memory.* A simple application is clearing blocks of memory, or, more generally, filling them with a fixed word value. This only has practical value when the memory system is finely interleaved, e.g., one or two words. For example, with memory interleaved across memory modules every 8 bytes, a machine with 1024 memory modules could set 8K bytes of memory in the same time required for a single ordinary memory access (but with increased use of memory and interconnection bandwidth). Smaller machines would have a correspondingly smaller benefit. Note that this requires the broadcast message to contain a partial memory address along with the data to be stored (and possibly partial broadcast information).
- *Broadcast memory modifiers.* By specifying an arithmetic or logical operation along with the broadcast, certain useful functions could be performed by the memory module, rather than simply storing data. For example, a constant may be added, ORed, or ANDed to each word of a suitably sized and aligned array.
- *''Reverse combining'': vector sum.* The normal role for combining in an Ultracomputer is to take place on the path from processor to memory, and de-combining to take place on the return path (the same path, in reverse). While it is common to depict an Ultracomputer in the manner of Figure 2 (**A**), in section 2.1, it is also simple (in concept) to fold or bend the network around so that the processor and memory modules are co-located. If this is done, one can imagine sending a message in the "wrong" direction. If, as in the Ultra-3 prototype, we have a network that performs full broadcast on the path normally taken for traveling from memory to processor, then we can consider traveling in reverse: do a full broadcast on the way from processor to memory modules, access memory as with a Fetch&Φ operation, and perform combining on the way back to the processor. For example, we could compute the sum of a suitably sized and aligned block of memory in the same time as a single normal memory access. At least two other enhancements are

needed to make it work:

Suppression of the wait-buffer: the operations being combined will never be decombined, so nothing should be held in the wait buffer for them.

Receiving buffer at processor: when the result arrives back at the originating processor, it may not yet be reduced to a single message, due to delays in the network interfering with combining at some point along the way.[69] The network interface, which receives the messages, should combine them. One additional problem is detection of completion: how can the processor know when all responses have arrived? This may not be possible without further switch enhancements (such as performing *two* ALU operations per message combine, so as to count the number of *virtual* messages combined into one), or using some additional network traversals.

Again, we need to know the cost, performance, and frequency of this kind of operation to evaluate it fully.

3.9. Chapter Summary

In the 7 major sections of this chapter we presented, sometimes in excruciating detail, the underpinnings of Symunix-2. This foundation is shaped primarily by our desire to maximize performance and scalability for our target class of machines.

One purpose of this chapter is simply to introduce some functions and notation to be used in code samples throughout the rest of the dissertation. This was our primary motivation in presenting material on Fetch&Φ, TDR, histograms, and interrupts, for instance. Another role played by this chapter is to review previously existing material on Fetch&Φ-based algorithms, in order to remind the reader how such algorithms are constructed. Our treatment of counting semaphores, readers/writers locks, and group locks, certainly falls into this category, but at the same time we tried to provide "added value" by relentlessly structuring the code to emphasize the distinction between interface and implementation and to maximize performance of the latter.

This attempt to define an interface has a greater goal than just to enlarge the presentation. In conjunction with the results to be presented in chapter 8, we hope to convince the reader that the modular interfaces we have chosen are general enough for a wider class of machines, while still unlocking the potential of our target class. We cannot overstress the need for careful design of this interface and the implementation(s) beneath it.

We presented several new data structures and algorithms in this chapter: APCs, RBCs, LRU lists, visit lists, broadcast trees, and search structures. In addition, our algorithms for readers/readers locks and sets will be new to almost all readers. We introduced several "minor" techniques that play a crucial role in minimizing overhead; chief among these are the two-stage implementation strategy for synchronization code, check functions, and the corresponding interrupt mask manipulating function, `splcheckf`. This level of detail is not often presented in the literature, but we presume we were not the first to invent such techniques.

---

[69] Also, with the Ultra-3 network, no combining could occur in the first stage (Dickey and Kenner [56]).

Chapter 4: Processes

This chapter is concerned with issues of process management as performed by the operating system kernel. We begin in section 4.1 by looking at the development history of the process abstraction in Symunix. Our process model is an extension of the traditional UNIX process, so the next two sections are devoted to describing the major attributes we added: section 4.2 is concerned with basic process management in the form of creation, destruction, and error handling, and section 4.3 presents the *activity*, a mechanism to harness and control asynchrony within a single process by providing asynchronous system calls and page faults. We then turn our attention to implementation issues. Sections 4.4–4.8 describe major data structures, context-switching synchronization, scheduling, process creation/destruction, and signals, respectively. After looking at a few ideas for future work in section 4.9, we summarize the chapter in section 4.10.

4.1. Process Model Evolution

Following the positive experience with Symunix-1, the general direction of design for Symunix-2 was clear, but one new goal influenced the Symunix-2 design in particularly far-reaching ways: flexible support for efficient user-mode thread systems.

Symunix-1 showed that the `spawn` and `mwait` system calls were an effective means of creating new processes in parallel (see section 8.3.1). As a result, early plans called for enhancement of `spawn` to support additional features such as *cactus stacks*, to provide a more direct match for our envisioned programming language constructs. In such a system, memory sharing and process creation would both closely follow the nesting structure of the programming language.

But, as time went on, it became clear that no matter how efficient we made `spawn` and `mwait`, there would still be many applications whose parallelism would be dominated by process management overhead. Attempting to address this problem, we proposed *pre-spawning*, a limited form of what has now generally become known as *user-mode thread management*. Under pre-spawning, *worker* processes would be spawned in advance of need, busy-waiting and performing work as necessary in response to execution of parallel statements in the program. In contrast to pre-spawning, the original model could be called *on-demand spawning*.

In their simplest forms, the run-time systems for on-demand spawning and pre-spawning would be quite similar: where the direct system would spawn, the pre-spawned system would simply engage the services of a worker. In practice, however, both kinds of systems are complicated by many issues, especially management of memory sharing and excess parallelism (beyond the obtainable number of processes or processors).

Pre-spawning was implemented as support for the ParFOR system (Berke [24, 25]), and was extremely successful, totally eclipsing all other parallel programming models on the Ultra-2 prototypes.

A more general approach than that taken in ParFOR would be more dynamic: instead of using a fixed-size pool of worker processes, they might be created on demand as an application's parallelism expands, and destroyed when the supply of idle workers exceeds a threshold; there are many possibilities. Semantics can be matched to language or application requirements by careful selection of shared vs. private memory and other operating system state. Each of these possible *user-mode thread management* systems supports a general model of execution in which threads of control, with certain well-defined properties, may be created, executed, and destroyed.

Supporting general thread models presents some difficulties for a user-mode implementation:

- *Input/Output.* When a thread waits for I/O, it must block. The simplest thread implementations built on a UNIX-like process model will allow the thread to execute most ordinary UNIX system calls directly. This will often block the process running the thread, leaving the processor available to run something else. For a single application to utilize a fixed number of processors continuously, it must have extra processes available to run in this situation. However, extra processes must be multiplexed onto the same physical processors, introducing load balancing problems and increasing overhead through context-switching and disruption of caches.
- *Page Faults.* Page fault handling is very similar to supporting synchronous system calls. The thread which encounters a page fault cannot continue executing until the page is brought into memory from secondary storage, but other threads could profitably use the same processor. The simplest potential solution, not having page faults at all, is often infeasible. Allowing page faults is somewhat akin to preemption.
- *Preemption.* When running a thread system under a general purpose timesharing OS, processes are vulnerable to preemption by the system scheduler at any time. The concept of preemption is reasonable; the problem is that the thread executing on a preempted process is also unable to run, even if other processes within the same application are idle. Worse yet, other threads running on other processes may be waiting for the preempted one, but there is no way to get the preempted thread running again quickly.
- *Thread management.* The user-mode implementation of thread management itself is subject to all the problems mentioned above, but in a much more severe form. For example, if a process executing a thread management operation (such as thread context-switch) makes a blocking system call, takes a page fault, or is preempted, there is no alternative: the process will be unavailable for the entire delay. The processor can switch to another process, but having extra cooperating processes, beyond the available number of processors, is counterproductive most of the time because of context-switch overhead. To make matters even worse, the process may hold critical locks that prevent *any* cooperating process from performing thread management operations.

As a direct result of these problems, a general trend developed in the research literature: *kernel implemented threads*, or, simply *kernel threads*. Kernel threads directly implement the thread abstraction: the flow of control itself is separated from other process attributes, most importantly from the address space. In such systems, the kernel directly supports and schedules threads, with the following advantages:

- *Popular semantics.* The common designs share the most important items of traditional process state, such as the address space and file descriptors. This means that, just as with a traditional UNIX process, memory allocation and *open* system calls have an application-wide effect.
- *Simplicity.* When implemented in the kernel, the thread implementation has full control over the system, and direct access to the algorithms and data structures used for scheduling, etc. This is in contrast to the "arm's length" relationship available to a user-mode thread implementation.
- *I/O, page faults, and preemption.* The problems faced by user-mode thread implementations are simply not relevant to kernel threads. When a kernel thread makes a synchronous system call or gets a page fault, it is suspended and another thread is scheduled onto the otherwise-idle processor. Since all threads are known to the scheduler, it can always find one to run, if there are any.
- *Ease of implementation.* Since the thread system is implemented within the kernel, the implementor has quite a bit of freedom. Algorithms and data structures hidden from the user process are available, and may be used directly or even redesigned to suit the thread system better. In general, kernel implementations have much greater control over the entire system, and enjoy privileges, such as control over preemption and interrupts, that are often unavailable to user-mode implementations.
- *Integration.* Because there is a single thread abstraction supplied by the kernel, every parallel programming environment can use many of the same tools and facilities for debugging, performance monitoring, etc. The similarity of threads from environment to environment also benefits programmers who work on multiple environments, or develop new ones, as there are no thread implementation decisions to make.

Unfortunately, kernel threads are also more expensive than user-mode threads. While creating, destroying, and context-switching between two kernel threads in the same address space is generally an order of magnitude cheaper than the corresponding operations upon ordinary UNIX processes, user-mode thread systems outperform them by about another order of magnitude. Of course, context-switching between different address spaces costs about the same, whether ordinary processes, kernel threads, or user threads are employed.

Of course, kernel threads can also be used as a basis for user-mode thread implementations, in almost exactly the same way as UNIX-style processes can, but the run-time cost is also essentially the same.

For *coarse grained* applications that don't frequently invoke thread operations, the advantages of kernel threads may still be attractive. The question may fairly be asked, why not have both kernel and user-mode thread implementations? This question is hard to answer, partially because there is still relatively little experience with sophisticated user-mode implementations, and partially because it is largely a question of design principles. To some extent, it's a matter of *where* to put the complexity: in the kernel, outside, or in both places. Complexity is concerned with implementation cost, run-time cost, and also the conceptual intricacies of the kernel/user and other interfaces. (See the discussion in sections 2.2.3, 2.3.11, and 2.3.12 for an overview of other approaches.)

We are hopeful future experience will support a more informed design decision, but the current approach taken for Symunix-2 is to improve the UNIX process model to support general user-mode thread models properly, rather than implementing threads as an alternative kernel provided service.

Symunix-2 provides support in the kernel to allow sophisticated applications to do performance critical operations directly. This has two advantages over kernel-based thread implementations: it avoids the direct overhead of crossing into and out of the kernel's protection domain for some very common operations, and it allows economy through specialization, since few applications need all the generality incorporated in a general purpose OS kernel.

This strategy has determined the shape of the most significant new features of Symunix-2, affecting both the kernel interface and the underlying implementation model.

- *Asynchronous system calls* (§4.3.1) and *asynchronous page faults* (§4.3.2) form the basis for thread implementations outside the kernel.
- The scheduler extends limited but direct control over preemption to user-mode programs via the *temporary non-preemption* facility (§4.6.4) and the `SIGPREEMPT` signal (§4.6.5).
- Since the kernel doesn't implement multiple threads per process, parallel programs are executed by collections of processes. As a result, scheduler policy extensions, needed to efficiently support some applications, apply to a new process aggregate, orthogonal to sharing of address spaces: the *scheduling group* (§4.6.3).
- Because flexibility is one of the advantages of user-mode thread implementations, the kernel doesn't directly support *shared address spaces* since they are rather special purpose. Instead, the memory management model was designed to provide efficient primitive support for a variety of sharing patterns, including shared address space emulation (§7.6).

4.2. Basic Process Control

This section discusses our changes to the basic UNIX process model other than those related to asynchrony (to be covered in section 4.3). These changes are designed to help an application with the most basic management of its processes—creation, destruction, and error handling.

We retain the `spawn` operation from Symunix-1, with which many processes may be efficiently created at once.[70]

```
int
spawn (int n, int flags, int *pids)
```

The parameter *n* tells how many processes to create, and *pids* points to a suitably large integer array, where the process ID of each child is stored. The return value is zero in the parent and a unique *spawn index*, *s*, chosen from the set {1,...,*n*} in each child, such that, for each child *i*,

$$\texttt{pids}[s_i - 1] = pid_i$$

The parameter *flags* provides several options shown in Table 18, on the next page.

As part of the `spawn` support for parallel process management, we introduce the notion of a *process family*: the collection of processes, related by `spawn` (or `fork`), comprising a parallel program. A new family is created when a process begins a new program via the `exec`

[70] Although the basic function of `spawn` remains the same, we have modified the details based on experience. The new `spawn` is both easier to use and more capable.

Flag	*Effect*
SPAWN_PROGENITOR	Children are adopted by family progenitor
SPAWN_N_EXITZ	Notify parent on zero exit status
SPAWN_N_EXITNZ	Notify parent on nonzero exit status
SPAWN_N_EXITSIG	Notify parent on death by signal
SPAWN_N_SUSPEND	Notify parent on stop signal
SPAWN_N_RESUME	Notify parent on continue signal

Table 18: Flags for spawn.

The flags are integer values which may be or'ed together. The SPAWN_N_*x* flags (N stands for "notify") control generation of the SIGCHLD signal and whether or not wait returns for child state transitions of type *x*. In this context, *parent* is either the natural parent or the progenitor in the case of SPAWN_PROGENITOR).

system call; such a process is called the *progenitor* of the new family. There are only two ways in which the family concept affects the kernel:

(1) The SPAWN_PROGENITOR option flag to spawn allows the role of parent to be concentrated in the progenitor, without requiring the progenitor to do every spawn itself. This feature is relevant to applications in which parallelism changes dynamically throughout the execution of the program. A spawn with this option proceeds in the usual way, but the children see the family progenitor as their parent (e.g., via getppid), rather than their true parent. Once the spawn succeeds, the true parent has no further role in managing the children. The progenitor will wait for them and collect error status as appropriate (p150). Because the true parent is free of such responsibilities, it may terminate before its children, with no risk of them becoming *orphans* and being inherited by the system's *init* process, which would discard their exit status.[71]

(2) The ability to send a signal to all family members is provided by an extended kill system call. As described in section 4.8.2, signals may be sent efficiently to several kinds of process aggregates; the family is one of them.

In Symunix-1, the mwait system call was introduced to fulfill the function of wait for spawned processes, without causing serialization in the common case. The most important behavioral difference between wait and mwait is that mwait does not actually return status information for a terminated child, rather it returns the number of spawned children terminated since the last call to mwait. In Symunix-1, one of the arguments to spawn is an array in which the exit status information for each child is stored when it eventually terminates. If mwait returns a positive number, it is necessary to search the array to determine if any child terminated abnormally.

A simpler and more flexible method is used in Symunix-2, eliminating the need for mwait and the status array argument to spawn. The SPAWN_N_*x* flags in Table 18 allow the

---

[71] In fact, in our implementation, *init* is never notified about orphans, completely avoiding any possibility of *init* becoming a bottleneck. Orphans' exit status values are discarded. See section 4.7.

parent to control which child state transitions will report status via `wait` and `SIGCHLD`. A call to `wait` will delay the parent (or progenitor, if `SPAWN_PROGENITOR` was used) until a reportable condition exists or no such conditions are possible. Typically, `spawn` will be specified with `SPAWN_N_EXITNZ|SPAWN_N_EXITSIG`, which will cause `wait` to delay until some child terminates abnormally or all children have terminated; the latter case is, we hope, more common.

Exit status information available to parents via `wait`, together with the `SIGCHLD` signal, allow a parent (progenitor) to report abnormal terminations of children reliably, but some mechanism is also needed to handle abnormal termination of a parent (progenitor). This is a potentially important error condition because, in some situations, inopportune termination of a parent (progenitor) is just as likely to result in deadlock among its children as the death of a child would. To solve this problem, we introduce the `SIGPARENT` signal, automatically delivered to all children of a terminating parent (including any "adopted" children, in the case of a progenitor's termination). This signal is ignored by default, but concerned applications may catch it or have it cause termination.

4.3. Asynchronous Activities

We introduce a major new kernel abstraction, the *activity*, and transform the process from an active to a passive entity. Activities are not explicitly visible at the user/kernel interface; their effect is apparent only in the form of two new features: asynchronous system calls and page faults. The situation is completely different internally, where activities are central. The unit of scheduling within the kernel is the activity. Each process has one or more activities, one of which is the *primary* activity. Only the primary activity ever executes in user-mode, and in fact we continue the tradition of using expressions like "user process" to mean the user-controlled execution path of the primary activity. Logically, activities are created and destroyed automatically as side effects of asynchronous system calls and asynchronous page faults, but the implementation can use cacheing techniques to avoid some of the work in practice. Whenever a new activity is created, it takes over the role of primary activity, meaning that it acquires all necessary user-mode state (such as register values) from the old primary. The old primary completes its work in the kernel and then vanishes.

Each activity consists of a structure containing various items of state necessary for scheduling and synchronization, together with a stack for use when executing in kernel-mode. Other items of process state not peculiar to a single activity are maintained in the proc structure, which is generally similar to that of traditional UNIX implementations. The u-block of traditional implementations has been eliminated in favor of the proc and activity structures; the major impact of this change will be discussed in section 4.4.5.

The main purpose of activities is to allow a user process better control over its own processor use. By using the new features of asynchronous system calls and page faults, it is possible for a user process to maintain control of its processor at times when it would block if only synchronous mechanisms were used. Of course, maintaining better control takes more work, and we can expect this to be reflected in larger and more complex applications. We expect the impact of this extra work to be ameliorated in two ways:

(1) For most applications, the extra work will be done within libraries and programming language run-time support systems, which only have to be written once. This will directly reduce the development cost of many applications.

(2) Tailoring such libraries and run-time systems to the needs of specific classes of applications will reduce their complexity and improve performance by eliminating unnecessary features. This will make the applications more cost-effective at run-time.

Nevertheless, performance gains come with a price: the old uniprocessor process models do not yield significant parallel speedups without extension, and extra work is needed to take advantage of those extensions.

4.3.1. Asynchronous System Calls

The basic UNIX I/O model of `open`, `close`, `read`, `write`, and `lseek` operations on byte streams is simple, powerful, and quite appropriate for the workload to which UNIX systems have traditionally been applied, but in high performance environments some extensions are often necessary. One such extension is asynchronous I/O, which has existed in non-UNIX systems for decades and even within the UNIX kernel since its inception, but has become directly available to the user only relatively recently, on some systems such as those from Cray [51] and Convex [49].

Of course it is possible to obtain a form of asynchronous I/O on any UNIX system that has (at least) shared memory, simply by using another process to issue the synchronous I/O call. Our objections to such an approach (and similar ones based on kernel threads) are that it is less efficient, since additional context-switches are needed and more complex user/library software is required. Asynchronous I/O and interrupts have been used in computer systems for over 30 years, and it is safe to say that they are better understood than general MIMD shared memory computing (perhaps because interrupts are more "structured"). The principle of using the least powerful mechanism available to solve a problem (§1.4) would seem to favor use of asynchronous I/O with interrupts, over separate processes or threads, in most cases.

In large-scale parallel systems, user access to asynchronous I/O is desirable not only for high bandwidth and increased overlapping of processing and I/O activity, but also to facilitate the construction of efficient user-mode thread management systems (§7.5).

Meta System Calls

Rather than propose individual extensions to allow asynchrony in certain designated I/O operations, we propose a general strategy that can be applied to any system call for which asynchronous operation makes sense, e.g., not only `read` and `write`, but also `open`, `close`, `ioctl`, `mkdir`, `rmdir`, `rename`, `link`, `unlink`, `stat`, and so forth. The general idea is to define a control block for asynchronous system calls, containing a *specific call* identifier, its arguments, return values, and status (e.g., executing asynchronously, successfully completed, or completed with an error). Both the user and the kernel have the option to suppress asynchrony, reverting to synchronous behavior.

Three new *meta system calls* are introduced, each taking a control block pointer as an argument:

`syscall`
: Issue a system call. The return value indicates whether the call has already completed; otherwise a signal, `SIGSCALL`, will be delivered when it is.

`syswait`
: Wait for completion of a system call. The return value distinguishes between control

blocks that aren't pending, those that have completed naturally, and those that have been canceled.

`syscancel`
Forcibly cancel a pending system call, if possible.

The control block for the meta system calls, called a `metasys` structure, is simple:

```
struct metasys {
     unsigned number;                // specific system call
     int  status;                    // done, async, aborted, error
     int  flags;                     // see below
     void (*done)(struct metasys *); // func for handler to call
     union {
          struct { int fd; }                             close_args;
          struct { char *path; int flags, mode, fd; }    open_args;
          struct { int fd; void *buf; long icount, ocount;
                   off_t pos; }                          read_args;
          struct { int fd; void *buf; long icount, ocount;
                   off_t pos; }                          write_args;
     ...additional substructures, one for each system call...
     } args;
};

// status and return values (also errno values, all positive)
#define SCALL__DONE     0    // completed or canceled
#define SCALL__ASYNC    (-1) // asynchronously executing
#define SCALL__ABORTED  (-2) // canceled
#define SCALL__NOTFOUND (-3) // bad arg for syswait or syscancel

// flag values
#define SCALL__NOSIG    01   // suppress signal on async completion
#define SCALL__NOCANCEL 02   // disallow syscancel
    // the following are mutually-exclusive
#define SCALL__SYNC     04   // synchronous; signal causes cancel
#define SCALL__AUTOWAIT 010  // implicit syswait
```

The usage is simple. The user allocates a `metasys` structure, fills in the `number`, `status`, `flags`, and `done` fields, as well as the input parameters of the appropriate `args` union member, and finally passes the structure's address to `syscall`. Each member of the `args` union is self-sufficient for a specific system call, and contains both input and output parameters. For example, a read operation uses the `read_args` structure, in which `fd`, `buf`, `icount`, and `pos` are input parameters for the file descriptor, buffer address, number of bytes requested, and file offset, respectively. If the operation is successful, the number of bytes actually read is returned in the `ocount` field. Correct use dictates that users not modify the `metasys` structure after calling `syscall`; the kernel is allowed to read the structure any time after the call begins, and to rewrite the structure any time prior to completion.

Once `syscall` is invoked, subsequent behavior depends on the specific call number, the flags, signal arrivals, and possibly the dynamic state of the kernel. There are four kinds of return values from `syscall`:

`SCALL__DONE`
: The call completed, successfully. There was no visible asynchronous behavior, for one of the following reasons:

  - The specific call was too simple to justify the extra overhead of an asynchronous activity. A short non-blocking system call should work this way. (To eliminate the meta system call overhead for such cheap calls executed by serial programs, they may also be supported by traditional trapping instruction sequences.)
  - The specific call was one of a few that always run synchronously, even if blocking is required, such as `exec`. An asynchronous `exec` would just not be useful.
  - The dynamic state of the operating system was such that the call was able to complete without actually needing to block. This condition, which may be non-deterministic, requires that no physical I/O be necessary and that all context-switching synchronization needed within the kernel be satisfied without delay. This optimistic strategy avoids activity overhead in common cases.
  - One of the flags `SCALL__SYNC` or `SCALL__AUTOWAIT` was specified by the user. Both cause synchronous behavior, but they differ in their response to signals. `SCALL__SYNC` allows a signal to interrupt "slow" system calls, such as reading from a terminal, as if `syscancel` were used; it closely models traditional UNIX system call semantics. `SCALL__AUTOWAIT` acts as if an implicit call to `syswait` was issued right after `syscall`.

  The return value, `SCALL__DONE`, is also stored in the `status` field of the `metasys` structure.

Positive (`errno`) values
: These values indicate failure rather than success, but are otherwise similar to `SCALL__DONE`. The return value (an `errno` value) is also stored in the `status` field of the `metasys` structure. None of the other syscall return and status values are positive.

  Note that none of the three meta system calls actually set the global variable `errno`, and that there are no errors attributable to the meta system calls themselves.[72] User-mode system call stub routines are responsible, as in traditional UNIX implementations, for setting `errno` appropriately and performing other similar actions to retain UNIX compatibility.

`SCALL__ASYNC`
: The call has not yet completed, and is progressing asynchronously. When complete, a `SIGSCALL` signal will be generated, unless suppressed with the `SCALL__NOSIG` flag. The signal handler will be invoked with the address of the `metasys` structure as an extra argument. In the meantime, the `status` field of the `metasys` structure may be examined at any time to poll for completion. Polling is best accomplished if the user stores `SCALL__ASYNC` into the `status` field prior to calling `syscall`; when

[72] The signals `SIGSEGV` and `SIGBUS` can be generated, however, if the argument to a meta system call is an invalid pointer.

the call is finally completed, the kernel will update `status` to a different value.[73]

If the `SCALL__AUTOWAIT` flag was specified, the `SCALL__ASYNC` return value indicates that a signal was caught, and the handler returned.

`SCALL__ABORTED`
: This value indicates the system call was aborted by `syscancel`. It can only be returned by `syscall` if `SCALL__SYNC` was specified, but can occur more generally as a final value for a `metasys status` field, and can also be returned by `syswait`.

The second meta system call, `syswait`, allows a process to await the completion of an asynchronous system call. It has the same set of return values as `syscall`, plus another possibility:

`SCALL__NOTFOUND`
: The argument to `syswait` is bad; this usually means the system call has already completed.

The third meta system call, `syscancel`, also operates on asynchronous system calls; it aborts them if possible. The high level view is that certain "slow" system calls, such as reading from a terminal, may be aborted. The low level view is that an activity within the kernel may be aborted when it performs an interruptible context-switching synchronization (§4.5/p167). The effect of `syscancel` upon `syscall` and `syswait` has already been described. A call to `syscancel` never blocks. The return value of `syscancel` itself indicates the possible success or failure of the cancelation:

`SCALL__ABORTED`
: Steps were taken to abort the system call but, because `syscancel` never blocks, it can't know if those steps were successful or not. To find out if the system call was actually aborted or not, it is necessary to wait for completion of the asynchronous system call itself (by busy-waiting on the `status` field, catching `SIGSCALL` if enabled, or calling `syswait` if necessary) and check the `status` field value; it will be `SCALL__DONE` or `SCALL__ABORTED` on completion. Unless suppressed, `SIGSCALL` is always delivered upon completion, whether a system call completes normally or is aborted.

`SCALL__ASYNC`
: The system call couldn't be aborted because it was marked with the `SCALL__NOCANCEL` flag, or because cancelation is always ignored for that specific kind of system call.

`SCALL__NOTFOUND`
: As with `syswait`, this indicates that the `metasys` pointer is bad; probably the system call has already completed.

The `SCALL__NOCANCEL` flag prevents a particular system call from being aborted, either by `syscancel`, or by a signal (if `SCALL__SYNC` is set).

---

[73] Since the `status` is set when `syscall` returns, there may be no need for the user to set the value initially. Doing so, however, eliminates any dependency on the atomicity of meta system calls with respect to caught signals. What if a signal is delivered just as `syscall` is returning? Does the user's handler see the `status` value set by `syscall`, or the previous value? We prefer to leave this undefined, as there are valid implementation strategies leading to either answer.

The `syscall`/`syswait`/`syscancel` framework relieves the programmer from having to know exactly which calls have asynchronous versions and which don't. By default, most calls can exhibit either kind of behavior, depending on whether or not blocking is actually necessary; the `syscall` return value lets the user know. By supporting all system calls within this framework, not just those that currently generate useful asynchronous work, the OS implementation is free to evolve; the set of useful asynchronous system calls may change as the system is modified in the future.

In some cases, new specific calls or modified semantics must be introduced to make the most of asynchrony. For example, versions of `read` and `write` are provided with explicit file position arguments, eliminating the need for separate `lseek` calls. This is important for asynchronous I/O since concurrent operations can be completed in any order.

Instead of requiring all uses of asynchronous system calls within a program to use the same signal handler, we provide the `done` field in the `metasys` structure, so that a single handler, provided by a standard library, can specify separate handlers for each call. Most programmers will not directly set the signal action for `SIGSCALL`.

Higher Level Interfaces

As a matter of convenience, we expect the development of a higher level interface to the asynchronous system calls along the lines of

```
size_t aread (int fd, void *buf, size_t count, size_t pos,
              void (*func)(struct metasys*));
```

where the function pointed to by `func` is called on completion with a pointer to the actual `metasys` structure, so that the various fields may be inspected as necessary. The implementation of this higher level interface is merely a matter of allocating the control block, issuing the `syscall`, and, upon receipt of `SIGSCALL`, calling `*func` and finally deallocating the control block.

As already mentioned, the traditional UNIX system call interface is synchronous; emulation of such is also straightforward. The only real source of complication is signal handling. In 7th Edition UNIX and compatible implementations, when a signal arrives during a "slow" system call (such as `wait`, `pause`, or some cases of `read` or `write` on "slow devices") and a handler has been set for the signal, the system call is interrupted. If the handler returns, things are arranged so that the interrupted system call appears to return a special error, `EINTR`. Because it is sometimes inconvenient to deal with `EINTR`, more recent UNIX systems will automatically restart the system call in some cases. The introduction of asynchronous versions of all interruptible system calls allows us to avoid the problem altogether (from the kernel's point of view), by simply not interrupting the system call. If the handler really wants to interrupt a system call, it can do so by invoking `syscancel`.

Implementation of the traditional synchronous system calls is clearly important, not only for compatibility with old programs, but also because the synchronous functions are genuinely useful. There are two reasonable implementation paths:

(1) Support the old calls directly in the kernel, alongside the new ones.

(2) Implement the old calls in terms of the newer ones. The main difficulty lies in properly handling interruption by signals.

We have chosen the second approach for Symunix-2, and the `SCALL__SYNC` flag was added to `struct metasys` to simplify the implementation.

4.3.2. Asynchronous Page Faults

Conceptually, asynchronous system calls and page faults are quite different: an asynchronous system call returns to the application while the operation is still in progress, but a page fault *cannot* return until the missing page is fetched from secondary storage. But the motivation for asynchronous page faults is nearly the same as for asynchronous system calls: to overlap computation and I/O, and to provide efficient support for user-mode thread management. Asynchronous page fault handling is less general than asynchronous system calls, since the latter are still useful in the absence of threads.

Of course, all page faults on normal machines are asynchronous at the hardware and operating system level; we are merely extending the technique to the user level. Since a process encountering a page fault can't "return" to user-mode until the page fetch is complete, we avoid blocking by delivering a new signal type, `SIGPAGE`. In a thread environment, the signal handler can perform a context-switch to another thread.

Clearly, for `SIGPAGE` to be useful, the handler must not encounter another page fault. The code of the handler itself, any routines it calls, its stack, and the data structures it may touch should all be "wired down" to be sure they remain memory resident. However, in some environments, it may be impractical or impossible to wire down all such pages, and some faults may occur while the `SIGPAGE` handler is running. For this reason, and to support non-thread applications, page faults are handled synchronously under any of the following conditions:

- If `SIGPAGE` is not to be caught (action `SIG_DFL` or `SIG_IGN`).
- If `SIGPAGE` is masked.
- If `SIGPAGE` can't be delivered due to a second page fault (e.g., accessing the user's stack).

It is expected that `SIGPAGE` will normally be handled on a dedicated stack. Issues of user-mode thread management with asynchronous page fault handling is discussed in more detail in section 7.5.

4.4. Implementation of Processes and Activities

Here are some key features of the implementation of processes and activities in Symunix-2:

- Activities are represented in the kernel by a `struct activity`, containing a stack and other data structures to allow independent operation of each activity. Activities are fully autonomous, much as processes are in regular UNIX. They can run, block, and be scheduled independently of other activities (but scheduling dependencies for efficient parallel execution are also accommodated (§4.6.3, §4.6.5)).
- Each process, represented by a `struct proc`, has a pointer to its *primary activity*. This pointer changes over the life of a process. Likewise, each activity has a pointer to the process of which it is a part. In addition, a doubly linked list is maintained for all activities of each process. Although protected by a lock, the simple structure of this list exposes a basic assumption of our design for activities: there are never very many of

them,[74] so the list will not become a bottleneck.

- The u-block, present in most other UNIX implementations, has been eliminated. Its nonredundant contents have generally been divided between the proc and activity structures. The peculiar one-to-many address mapping of u-blocks has not been retained, so each activity structure has an ordinary kernel address, and a context-switch does not need to modify the kernel address space. Since an activity structure contains the kernel's run-time stack, it would be difficult to move it around in memory or swap it out, as is commonly done with u-blocks. This is because stacks typically contain pointers to other locations within the stack (either explicitly programmed or part of the stack linkage structure itself). Whereas u-blocks are relocated with the MMU to appear always at the same address, even after moving, we have chosen to rely on a static memory mapping for the kernel.
- The way activities are created is different from the way processes are created in other UNIX systems. The traditional approach is to make a copy of the parent's entire u-block; this works because the u-block of the *current process* is always mapped to the same address in kernel space, so the validity of pointers within the stack is preserved. Because of the role activities play in the system, this fork-like method of creation isn't needed. The individual elements of a new activity are simply initialized as appropriate. A newly created activity automatically becomes the *primary activity* of its process, the only activity entitled to execute in user-mode. The stack is initialized so that when the new activity first runs it will execute a specified function within the kernel, and then "return" to user-mode.
- Similarly, the lowest level context-switching facilities have been designed to suit the nature of activities. In traditional UNIX implementations, kernel execution state is saved within the u-block in three different places: `u.u_rsav`, `u.u_ssav`, and `u.u_qsav` (see Bach [15] and Leffler, *et al.* [134]). With such an approach, a process may block while still maintaining separate resumption points for swapping (or forking) and error handling. Multiple resumption points aren't needed for each activity, because the structure is never swapped out, forking is done by giving the new activity a "fresh" stack, and error handling is done by carefully passing error status back from every appropriate routine in the kernel.[75]

The following sub-sections give additional details.

4.4.1. Process Structure

The proc structure contains all per-process state, except the minimum required to implement activities. (In this sense, activities are very similar to threads in systems such as Mach, Topaz, and V.) The contents of the proc structure can be summarized as follows:

Lock A busy-waiting binary semaphore used to serialize critical process manipulation (see §4.4.4/p165).

[74] Certainly the number of activities per process is expected to be far less than $O(P)$ for a machine with P processors.

[75] But, to make the initial implementation easier, the part of the kernel dealing with terminals still uses `longjmp` to handle errors.

Free list item

A list item for the free proc list. Proc structures are allocated from this structure during `spawn` and `fork`.

`vislist` structure for children

A visit list (§3.7.3) for all children (natural or inherited via the *progenitor* feature of `spawn` (§4.2/p149)). It is used for sending signals and for dealing with orphaned children when their parent pre-deceases them.

`visitem` structure

This item is used with the parent's (or progenitor's) visit list for children.

List structure for debuggees

If the process is a debugger, using the `ptrace` system call, this list contains all processes being debugged. The main purpose of the list is to find the debuggees and terminate them when pre-deceased by the debugger. Because of the fundamentally serial nature of a single debugger handling multiple debuggees, this list need not be highly parallel; a `dllist` (§3.7.1/p82) will suffice.

List item for debugger

This item is used when debugging, to place the process on the debugger's list.

List structure for children to await

A list of children to be waited on by using the `wait` system call. Such children have either terminated or stopped.

List item for parent's await list

This item is used to insert the process on the parent's (or progenitor's) await list.

Address space information

A representation of the virtual address space, including machine-dependent data structures, e.g., pointers to page tables (§5.3).

File descriptors

Data structures to support the UNIX file descriptor abstraction, mapping small integers to i-node references.

Process aggregate data

Every process is a member of several aggregates, as listed in Table 19, on the next page. These aggregates are used primarily for sending signals, but the user IDs are also an important form of access permission. In addition, each process has a set of access permission groups, but there is no precedent in UNIX to support sending signals to the processes of such a group. The mechanisms for sending signals to process aggregates are described further in section 4.8.2.

Activity information

A list of all activities belonging to the process is kept, ordered by age. This is a simple doubly-linked list, rather than one of the lists described in section 3.7.1, because it must support searching, and because its length isn't expected to grow proportionally to the machine size. This list is searched by the meta-system calls `syswait` and `syscancel` (§4.3.1/p152), and when catching a `SIGSCALL` (§4.3.1/p151) or `SIGPAGE` signal (§4.3.2). Besides the list, the proc structure also has a pointer to identify the primary activity and a pointer to a *reserved* activity, unused but ready to assume

Aggregate	*Purpose*
All	All non-system processes
Login	Processes in same login session
Job	Traditional UNIX process group, used for "job control"
Family	Processes related by `fork` and `spawn`, but not `exec`
Siblings	Children of same parent or progenitor
UID	User ID

Table 19: Process Aggregates.

Of these, only "all" and either "login" or "job" are supported by traditional UNIX systems. Each process has two user ID group memberships, one for the *real* user ID and one for the *effective* user ID. A signal sent by a non-root process to UID *u* will go to all processes with either kind of user ID = *u*.

primary responsibilities when needed to execute a signal handler while the old primary continues with an asynchronous system call or page fault.

User/kernel communication addresses
: Certain memory locations in user space are known to the kernel and used as a low overhead alternative to system calls. Currently this technique is used for temporary non-preemption (§4.6.4), signal masking (§4.8.1), and signal polling (§4.8.1/p200). These addresses are stored in the proc structure, and changed by the `setkcomm` system call.

Flags
: The flags indicate variations in process status, modify process behavior in some way, or maintain state for specific operations. For example, there are flags that correspond to the `spawn` options of Table 18, in section 4.2 on page 149.

Signal information
: Structures to keep track of signal status for each defined signal type. This includes which signals are pending, ignored, masked, or have handlers, some semantic option flags (e.g. should signals restart interrupted system calls or use an alternate user stack when caught?),[76] and an `apc_pair` structure used to get the primary activity's attention if it is running when a signal needs to be handled.

Process ID
: The unique identifier for a process, used by a number of system calls.

Resource limits and usage data
: Certain resources, such as memory and processor usage, can be limited. Resource usage is also recorded separately for each process and its descendents. These things are handled in the general manner of Berkeley UNIX.

Timers
: Timer support as in Berkeley UNIX; real, virtual, and profiling timers are provided, along with the time-out feature of the select system call.

---

[76] As described in section 4.3.1 on page 153, interrupting system calls is an optional feature in Symunix-2; asynchronous system calls are not interrupted by signals.

Profiling data
: The traditional UNIX mechanism for collecting a histogram of a user's program counter samples.

Parent pointer
: A pointer to the parent process.

Special directories
: Each process implicitly holds open its root directory, current working directory, and image directory (§5.2.5). A pointer to each is kept in the proc structure.

Controlling tty pointer
: UNIX semantics define the concept of controlling terminal; a pointer in the proc structure identifies it.

`umask` The traditional UNIX `umask` is used to force certain permission bits off when files are created.

Priority and scheduling data
: The traditional externally visible priority in UNIX systems is the *nice* value. Actual scheduling relies on an internally generated priority based on recent processor and memory usage; these are all stored in the proc structure. Scheduling issues are discussed in section 4.6.

Name The last component of the path name of the running program. It is mostly for debugging, but can also be examined by system utilities such as *ps*.

Machine-dependent data
: Anything special needed by a particular machine.

4.4.2. Activity Structure

The contents of the activity structure can be summarized as follows:

Lock (`a_lock`)
: A busy-waiting binary semaphore, used to serialize critical manipulation of each activity (see §4.4.4/p165).

List item (`a_li`)
: A union of item structures described in section 3.7.1. This allows the activity to move among the free activity list, the ready list, and various waiting lists for the synchronization mechanisms described in section 4.5 (but not to be on more than one such list at a time).

Stack A run-time stack to support most operation within the kernel. The organization and size of the stack are completely machine-dependent, for example, the Ultra-3 activity stack is actually composed of two separate stacks to support the AMD 29050 microprocessor [2]. Whatever the organization, the stack area is fixed in size (a trait in common with most other UNIX-like kernels). Determining the correct size for the stack area is problematic, usually accomplished by making a conservative guess.

Context-switching state
: To perform a normal context-switch, in such a way that the previous activity may eventually run again, some state must be saved. In most machines, this includes things like register values, program counter, and stack pointer. In Symunix-2 most of this information can simply be saved on top of the stack, but the top of stack itself

must still be locatable. For most machines, this simply means the activity structure must contain a special field to store the stack pointer when the activity isn't running.

Process pointer (`a_proc`)
: A pointer to the process to which the activity belongs. The process structure in turn contains a pointer to the primary activity.

Activity pointers
: Each activity contains forward and backward pointers for the list of all activities belonging to a process. This list is implemented directly rather than as an abstract list type (§3.7.1) because it functions primarily as a search structure, and because highly parallel properties are not needed.

`metasys` or page fault address
: The user space address of the `metasys` structure (§4.3.1/p152) for the system call being executed or the virtual address of the page fault being handled. This field allows the primary activity to identify the target activity for a `syswait` or `syscancel` meta system call, and to generate the argument for a `SIGSCALL` or `SIGPAGE` handler.

Status (`a_status`)
: The general state of an activity is reflected in the `a_status` field; values are given in Table 20, below. There is no state for "executing"; in Symunix-2, that is reflected in a flag value.[77]

Flags (`a_flags` and `a_pflags`)
: The *zombie* state exists for much the same reason as the traditional UNIX zombie state for processes: as a holder for information to be passed back to the parent. In the case of activities, the notion of "parent" is replaced by the primary activity, but the role remains essentially the same. The information to be passed back is a `SIGSCALL` or `SIGPAGE` signal, and the accompanying address parameter (`metasys` structure or fault address, respectively). Several flags indicate special conditions for the activity; they are listed in Table 21, on the next page. The difference between ordinary flags and private flags is that the private ones may be manipulated without any locking. Typically, the private flags are only manipulated by the activity itself. A good example is the `ACT_RUNNING` flag, which is set by an activity when it begins

State	*Meaning*
`ACT_DEAD`	Terminated–unused structure
`ACT_READY`	Ready to run
`ACT_BLOCKED`	Blocked
`ACT_ZOMBIE`	Terminated–waiting to deliver signal

Table 20: Activity Status.

[77] `ACT_RUNNING`; see Table 21.

Ordinary Flags	*Meaning*
ACT_CANCEL	syscancel pending
ACT_NOCANCEL	Disallow syscancel
ACT_SYNC	Signal causes syscancel
ACT_SIGPAGE	SIGPAGE has been sent for fault
Private Flags	*Meaning*
ACT_RUNNING	Activity is executing
ACT_SPAWNING	Activity is spawning
ACT_DISTURBED	Activity prematurely unblocked
ACT_OPTIMASYNC	Primary activity doing async work
ACT_AUTOWAIT	Implicit syswait
ACT_UPFAULT	Activity handling user page fault

Table 21: Activity Flags.

Ordinary flags may only be examined or manipulated under protection of the general activity lock. Private flags need no locking because logic guarantees that no concurrent updates will occur.

to run and cleared as the final step in context-switching away from an activity (§4.4.5).

Processor number
: The number of the processor running the activity or last to run it. This is maintained for general information, but may also influence scheduling or context-switching.

Scheduling information
: Because each activity is individually schedulable, the priority is maintained on a per-activity basis. This priority is strictly internal to the kernel, and reflects the organization of the scheduler (§4.6.6). Additional data is kept as required by the scheduler, e.g, recent processor usage. The user-visible priority factor, called the *nice* value in UNIX systems, is maintained on a per-process basis.[78]

Blocking information (a_waittype and a_waitfor)
: When an activity is blocked, the nature of the blockage is recorded. The primary purpose is to support syscancel, but additional information is also maintained for the sake of general status reporting and to aid in kernel debugging. Two fields are required to record this information fully: a_waitfor, a union of possible blockage information, and a_waittype, which provides the interpretation for a_waitfor. The possible values for these fields are given in Table 22, on the next page.

[78] See section 4.6.3 for extensions to the traditional user-visible scheduling model.

a_waittype value	Meaning	a_waitfor union use	See
ACT_CSSEM	Counting semaphore	cssem *	§4.5/p166
ACT_ICSSEM †	Interruptible cssem	icssem *	
ACT_CSLOCK	Binary semaphore	cslock *	
ACT_ICSLOCK †	Interruptible cslock	icslock *	
ACT_CSRWR	Readers/Writers reader	csrwlock *	
ACT_ICSRWR †	Interruptible r/w reader	icsrwlock *	
ACT_CSRWW	Readers/Writers writer	csrwlock *	
ACT_ICSRWW †	Interruptible r/w writer	icsrwlock *	
ACT_CSEVENT	Event	csevent *	
ACT_ICSEVENT †	Interruptible event	icsevent *	
ACT_SYSWAIT †	Syswait	activity *	§4.3.1/p151
ACT_KSEM †	Ksem	ksem_wait *	§7.3.4
ACT_PSUSPEND †	sigsuspend system call	NULL	§4.5.1
ACT_JSUSPEND †	Job control suspension	NULL	§4.5.1
ACT_TSUSPEND	Tracing suspension	NULL	§4.5.1
ACT_WAIT †	wait system call	NULL	§4.5.1

Table 22: Types of Activity Blocking.

Appropriate values for a_waittype and a_waitfor are indicated. Values marked with † may be *or*ed with ACT_MAYCANCEL (§4.5/p167) if the activity doesn't have the ACT_NOCANCEL flag set (Table 21).

4.4.3. Activity State Transitions

The state transition diagram for activities is given in Figure 8, on the next page. The states and flags are the same as given in Figures 20 and 21, in section 4.4.2 on pages 161 and 162. Functions for block and unblock transitions n the next page will be given in Figure 23, in section 4.5 on page 166. The scheduler transitions will be discussed in section 4.6. The activity states in Symunix-2 are simpler than those of other UNIX systems for two reasons:

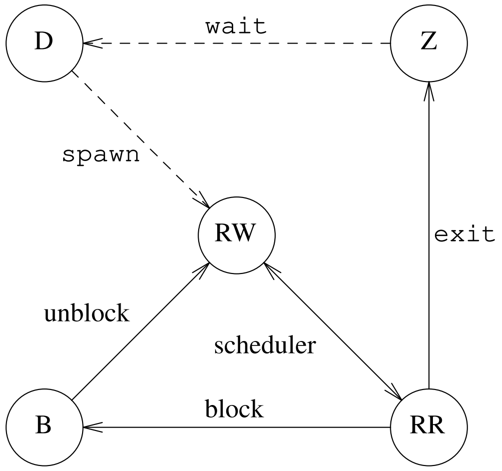
(1) We use the blocked state for many similar purposes, distinguishing them with the activity's a_waittype field (Figure 22, in section 4.4.2 on this page).

(2) We handle suspension for job control and debugging mostly at the process level, rather than the activity level. When the primary activity is ready to go from kernel-mode to user-mode and the process suspension flag is set, the activity blocks (with a_waittype flag set to ACT_TSUSPEND or ACT_JSUSPEND). With this approach, it is not necessary to deal with transitions between blocked and suspended states and any corresponding interference with normal control flow.

4.4.4. Process and Activity Locking

As already indicated, the proc and activity structures each contain a busy-waiting lock. Neither of these locks represents a serial bottleneck, because there are no natural operations that generate contention for them in proportion to the machine size.

The proc structure lock provides serialization in several situations:



D = Dead
Z = Zombie
RW = Ready (waiting to run)
RR = Ready (running)
B = Blocked

Figure 8: Activity State Transition Diagram.

> In the current implementation, all activity structures are created at system boot time and never destroyed. This is not fundamental, however, and even if the implementation is ever changed to do true dynamic allocation and deallocation of activity structures, it will still be worthwhile to keep a certain number of pre-allocated activity structures around, thus justifying the state "D" and the pseudo-transitions shown to and from it in the diagram.

- *Changing or finding the primary activity*. The only way to change the primary is to create a replacement (`csynth`, see §4.4.5). If the primary is running, it alone may do this, otherwise, it is sometimes possible for another process or even an interrupt handler to do it.
- *Manipulating or searching the list of activities*. Additions to the list are done only when the primary activity changes. Deletions are done by the primary as part of `syswait` or delivery of `SIGSCALL` or `SIGPAGE` signals.
- *Allocating, deallocating, or examining the file descriptors*.
- *Changing or examining the list of user/kernel communication addresses*, such as for temporary non-preemption (§4.6.4) and signal polling (§4.8.1/p200).
- *Changing or examining the flags* of the process (§4.4.1/p159).
- *Changing or examining signal handling actions or the signal mask*. Changing them is only done by the primary activity, but examining them is part of sending a signal, which is asynchronous.

The activity structure lock provides serialization in the following situations:

- *Changing or examining the status of the activity* (§4.4.3).
- *Changing or examining the ''ordinary'' flags* of the activity (§4.4.2/p162).

A simple ordered locking protocol is used to prevent deadlocks related to process and activity locks. In general, only a single lock is held at one time. There are three rules that allow multiple locks to be held in certain circumstances.

(1) A process may lock its parent while holding a lock on itself.

(2) A process that must be waited-for by its parent may be locked by the parent while the parent holds a lock on itself. This is permissible, since the child can't run anyway, and therefore isn't itself entitled to use the previous rule.

(3) If a process lock is already held, any number of activities belonging to the process may also be locked.

Note that if it is necessary to find and lock the primary activity of a process, the process must be locked first to prevent the primary from changing or terminating before the lock can be obtained. This is not a problem if the primary activity wants to lock itself: masking soft APC interrupts will prevent any asynchronous change of primary.

4.4.5. Primitive Context-Switching

In traditional UNIX kernels, and also in Symunix-1, the most basic context-switch to another process is performed by the function `resume` (see, for example, Bach [15] and Leffler *et al.* [134]). The corresponding function in Symunix-2 has been simplified, renamed, and is logically part of a three-function suite:

`void cswitch (struct activity *newact)`

> Save the context of the currently executing activity and switch to the activity indicated by `newact`. The context saved and restored includes appropriate machine registers and the current hard and soft interrupt masks, but doesn't necessarily include the user's memory mapping: that is set by the `asset` function (§5.3.1/p227).

`void cjump (struct activity *newact, int freeit)`

> Similar to `cswitch`, but the old context isn't saved. `cjump` is used when an activity is terminating. The `freeit` argument specifies that the function `freeact` should be called to dispose of the old activity structure as soon as possible. This option is needed because an activity cannot dispose of its own structure before termination. Freeing the activity structure is appropriate at `cjump` time unless the structure is needed for `syswait`, `SIGSCALL`, or `SIGPAGE`.

`void csynth (struct activity *oldact, struct activity *newact, struct proc *p, int f(genarg_t), genarg_t x)`

> This is the trickiest of the three context manipulation functions. It initializes (*synthesizes*) a new activity, based on an old one. The old one must be primary, and must either be the calling activity or be blocked. Both old and new activities are specified, as the old one need not be the currently executing one. The `proc` structure for the new activity is also specified, as it need not be the same as the old one's (e.g., for `spawn`). The new activity's `proc` structure must already be locked, since the primary activity will be changing. The old activity must also already be locked, if it is not the calling activity, to prevent it from changing significantly while the synthesis is taking place. The new activity becomes the primary, and is initialized to begin execution by calling `f(x)` in such a way that control will "go back" to user-mode

upon return, in the same way as the old activity would have (i.e., as a system call or exception would return). Note that `csynth` only performs initialization; no actual context-switch is performed. This allows flexibility in the order of execution of the two activities, and also allows `csynth` to be called from interrupt handlers (unlike `cswitch` and `cjump`).

Part of the price paid for the simplicity of `cswitch` is the delicate nature of the `csynth` implementation, which must construct a viable stack for the new activity and arrange for a fake "return" to user-mode that preserves all user-visible processor state (e.g. registers) in the general case. Although `csynth` can be pretty cheap, especially compared to the cost of copying a whole u-block, it must be intimately related to the first level trap and system call handlers. Because such code deals with processor concepts beneath the level of abstraction appropriate to C programming, much of it must be written in assembly language.

4.5. Context-Switching Synchronization

The fundamental mechanisms for context-switching were described in section 4.4.5: the functions `cswitch`, `cjump`, and `csynth`. This section addresses several higher level facilities, which may be divided in two alternative ways: whether or not waiting lists are used, and whether or not a blocked activity can be forcibly disturbed (this is called *premature*

	Mechanism	*Non-Interruptible*		*Interruptible*	
		Structure	*Key Functions*	*Structure*	*Key Functions*
Non-List based	ad-hoc	–	`dopause` `unpause`	–	`idopause` `iunpause`

List-based	binary semaphores	`cslock`	`csl_wait`
			`csl_signal`
				`icslock`	`icsl_wait`
					`icsl_signal`
					`icsl_trywait`

	counting semaphores	`cssem`	`css_wait`
			`css_signal`
				`icssem`	`icss_wait`
					`icss_signal`
					`icss_trywait`

	readers/ writers	`csrwlock`	`csrw_rlock`
			`csrw_runlock`
			`csrw_wlock`
			`csrw_wunlock`
				`icsrwlock`	`icsrw_rlock`
					`icsrw_runlock`
					`icsrw_wlock`
					`icsrw_wunlock`
					`icsrw_tryrlock`
					`icsrw_trywlock`

	events	`csevent`	`cse_wait`
			`cse_signal`
			`cse_reset`
				`icsevent`	`icse_wait`
					`icse_signal`
					`icse_reset`

Table 23: Context-Switching Mechanisms.

For list-based synchronization, the relevant structure name is given. For the non-list-based case, there is no structure.

unblocking or *cancelation*, and is generally brought about by `syscancel`; see section 4.3.1 on page 154). Table 23, on the previous page, lists the most important functions for all the context-switching mechanisms used in the Symunix-2 kernel.

The simplest mechanisms are the non-list-based ones, implemented by the `dopause` and `idopause` functions. In either case, the calling activity enters the `ACT_BLOCKED` state,[79] and is moved back to the `ACT_READY` state when some processor calls the corresponding `unpause` or `iunpause`.

There are many situations where `dopause` and `idopause` would be inadequate because the code that would have to call `unpause` or `iunpause` couldn't figure out what activity to awaken. This is the reason for the list-based synchronization mechanisms. Each type of list-based synchronization is made available through a structure containing the necessary synchronization variables and waiting list(s). Different functions are generally provided to wait for a resource and to release it. For example, a simple context-switching lock (e.g., `cslock`) provides a function to obtain the lock (`csl_wait`), blocking if necessary, and another function to release the lock (`csl_signal`), unblocking the next waiting activity if appropriate. The same blocked and runnable states are used as with non-list-based synchronization (`ACT_BLOCKED` and `ACT_READY`), but blocked activities are put on a waiting list.

The difference between non-interruptible and interruptible synchronization is that the latter supports *premature unblocking*: the function `unblock` can be called to force the activity back to the `ACT_READY` state. In such a case, the blocking function (e.g., `idopause` or `icsl_wait`) returns zero (*FALSE*). Generally the *FALSE* return causes the activity to abandon its current objective and clean up as appropriate. We say that the activity has been *disturbed* or *prematurely unblocked*, that its wait has been *interrupted*, and that the system call it was working on is *canceled*.

The choice of interruptible or non-interruptible synchronization is made statically, by the kernel programmer. Interruptible synchronization is used only when waiting time might be unbounded; all other context-switching synchronization is non-interruptible. (In determining the potential for unbounded waiting time, we assume no hardware failures.)[80] Only true system calls should be cancelable; interruptible synchronization must never be used when handling exceptions, such as page faults, synchronous or asynchronous. Aside from such behavior being nonsensical, this restriction is important because such exceptions have no way to return failure to the user, and the user has no reasonable way to respond.

To reduce overhead in common cases, we use a two-stage implementation for the key list-based synchronization functions, as we did for some of the busy-waiting synchronization functions (§3.5/p50). The first stage function can be *inlined*, or integrated into the calling function, eliminating function call overhead. The second stage is a conventional function, but is only called in the relatively rare case where blocking (or unblocking) is necessary.

---

[79] The activity state transition diagram is given in Figure 8, in section 4.4.3.

[80] We are only assuming reliability of the processors, memory, and interconnect. I/O peripherals can be expected to have higher failure rates, a fact which device drivers should account for, e.g. by setting timers to handle the possibility that an expected interrupt does not occur.

Some synchronization methods require awakening many activities at once. Indeed, these methods are preferred because they allow greater concurrency. The mechanisms in Table 23 with this property are readers/writers locks and events. In both these cases, a single processor must be able to awaken many blocked activities. In Symunix-1, the following approach was used: as each blocked process was awakened, it “helped out” by awakening other blocked processes, until there were no more. This approach offers, at best, a speedup of $N/\log N$, for N processors, but the scheduling latency of each wakeup limits the effectiveness of this technique. In practice, very little helping can go on because it takes too long for each awakened process to get actually running. Still, the scheme is simple and doesn’t require a special relationship with the scheduler itself.

In Symunix-2, we use Wilson’s approach [205] of developing algorithms that, working in conjunction with the scheduler, are able to wakeup many activities in a single step: the entire waiting list of blocked activities is inserted on the multiqueue portion of the ready list. The remainder of this section provides additional implementation details.

4.5.1. Non-List-Based Synchronization

An activity blocked by `dopause` is normally unblocked by `unpause` (a pointer to the activity is a parameter to `unpause`). The only difference between `dopause` and `idopause` is that the latter also allows the `unblock` function to disturb the activity. Before calling any of these functions (`dopause`, `unpause`, `idopause`, `iunpause`, or `unblock`), the activity must already be locked and soft interrupts must be masked. Those functions that cause blocking (`dopause` and `idopause`) are unusual in that they unlock the activity before returning. This behavior (releasing a lock passed into a function) is asymmetric, but is deemed natural for a blocking function, since releasing the lock before a context-switch is crucial. We do not apply this logic to the handling of the interrupt mask, however: interrupts must be unmasked by the caller upon return. Aside from being symmetric, the normal strategy for interrupt masking makes sense because the proper mask to restore is not known inside the blocking function.

The basic steps in blocking and context-switching are illustrated in `dopause`:

```
        // act is current activity
void
dopause (struct activity *act, int waittype)
{
      act->a_waittype = waittype;
      act->a_waittime = curticktime();
      act->a_status = ACT_BLOCKED;
      bwl_signal(&act->a_lock);
      cswitch(resched(0));
}
```

The identity of the current activity could be learned independently (e.g., the `curact()` function returns a pointer to the current activity), but the caller of `dopause` already knows the activity (recall that the caller must lock the activity before calling `dopause`) so may as well pass it to `dopause`. Recording `waittype` and `waittime` for the activity is mostly informational (e.g., for the UNIX *ps* utility), although `waittype` may be examined to decide whether or not to call `unpause` for this activity, and the scheduler may make good use of `waittime`. Once the activity’s state is changed, the lock may be released; an `unpause` will be able to succeed at this point. The next activity to run is chosen by `resched`, and the context-switch is accomplished by `cswitch`, which doesn't return until after the activity is awakened by `unpause`.

The complementary actions of `unpause` are even simpler:

```
void
unpause (struct activity *act)
{
      assert (act->a_status == ACT_BLOCKED);
      assert ((act->a_waittype & ACT_MAYCANCEL) == 0);
      assert (DOPAUSE_TYPES_OK(act->a_waittype));
      lmkready (act);
}
```

We have omitted assertions from most other code sections for brevity, but we include these here because `unpause` has no other real reason for being: `lmkready` (§4.6.6), which sets a locked activity's `status` to `ACT_READY` and puts it on the ready list, could just as easily be called directly. Furthermore, `unpause` and `iunpause` differ from one another only in the second assertion: the former wants the `ACT_MAYCANCEL` flag clear, and the latter wants it set. The third assertion is programmed with a macro defined at the point where all the `waittype` values are defined; it checks all the acceptable values in a single boolean expression:

```
#define DOPAUSE_TYPES_OK(waittype) \
         (((waittype) & ~ACT_MAYCANCEL) == ok_value1 || \
          ((waittype) & ~ACT_MAYCANCEL) == ok_value2 || \
     similar tests for other values)
```

This simple device preserves the independence of `unpause` from its uses.

The `idopause` function is basically the same as `dopause`, but with a return value and two new tests, for cancelation:

```
int
idopause (struct activity *act, int waittype)
{
      if (act->a_flags & ACT_CANCEL) {
            act->a_flags &= ~ACT_CANCEL;
            bwl_signal(&act->a_lock);
            return 0;
      }
      act->a_waittype = waittype | ACT_MAYCANCEL;
      act->a_waittime = curticktime();
      act->a_status = ACT_BLOCKED;
      bwl_signal(&act->a_lock);
      cswitch(resched(0));

      if (act->a_pflags & ACT_DISTURBED) {
            act->a_pflags &= ~ACT_DISTURBED;
            return 0;
      }
      return 1;
}
```

The `ACT_CANCEL` flag is set by `syscancel` (§4.3.1/p154) and the `ACT_DISTURBED` flag is set by `unblock` (§4.5.3/p185). Unlike `dopause`, the setting of `waittype` is more than informational here: `unblock` depends on it.

In both `dopause` and `idopause`, we have refrained from checking the `ACT_OPTIMASYNC` flag, as we will do for the list-based synchronization mechanisms (§4.5.2/p178). This flag improves the efficiency of asynchronous system calls in common cases, and there is no reason why it could not apply to `dopause` or `idopause`. We ignore it only because the purposes to which we have put non-list-based context-switching synchronization, such as `syswait` (§4.3.1/p151), tend to preclude asynchronous execution (i.e., `dopause` and `idopause` would never see `ACT_OPTIMASYNC` in practice).

4.5.2. List-Based Synchronization−Readers/Writers Locks

For the sake of brevity, we refrain from a full presentation of all the list-based context-switching synchronization methods of Table 23; one suffices, as they all share a common approach. We choose to present readers/writers locks, because it is the most complex and gives the fullest flavor of the design issues for this class of synchronization mechanisms. Furthermore, we concentrate on the interruptible version, pointing out only the major differences in the form of the non-interruptible version.

The basic algorithm, which comes directly from Wilson [205], maintains a counter equal to $MAXREADERS \times w + r$, where w and r are the numbers of writers and readers, respectively, at any particular time. *MAXREADERS* is a constant greater than r can ever be, including would-be readers that would still be waiting if they hadn't been disturbed. The essence of the lock protocol, which gives priority to write lock requests, is to increment the counter by 1 to get a read lock, or by *MAXREADERS* to get a write lock. By using Fetch&Add, the success of this attempt is indicated if the returned value is less than *MAXREADERS* (for a read lock) or equal to zero (for a write lock). Likewise, Fetch&Add is used to subtract the same value when releasing a lock; the presence of waiting readers and writers is indicated by the return value. This value alone doesn't reveal when the *last* reader releases its lock, this is determined with the help of a "running readers count", so that the first waiting writer can be awakened at the right time.

We use the following values for *MAXREADERS*:

`CS_MAXR` For `csrwlock`; the maximum number of activities allowable in the whole system. This number must be $\leq \lfloor\sqrt{\mathtt{INT\_MAX}}\rfloor$, i.e. `CS_MAXR` × `CS_MAXR` must be representable as a positive integer in standard C.

`ICS_MAXR` For `icsrwlock`; the maximum number of activities allowable in the whole system plus the number of premature unblocks allowed during one "write phase", i.e., while writers hold the lock and readers are waiting. We use the largest acceptable value: $\lfloor\sqrt{\mathtt{INT\_MAX}}\rfloor$. The consequences of making this number too small are discussed on page 176.

Here is the `icsrwlock` structure:

```
     // interruptible context-switching readers/writers lock
typedef struct _icsrwlock {
     unsigned char rpri; // readers priority
     unsigned char wpri; // writers priority
     int     acount;     // counter (MAXREADERS ×w + r)
     int     rrcount;    // running reader count
     mqitem  mqi;        // for ready list insertion
     dafifo  rwlist;     // readers' wait list
     dllist  wwlist;     // writers' wait list
} icsrwlock;
```

Besides `acount` and `rrcount`, the most important fields are the wait lists. The different kinds of wait lists used are shown in Table 24, below. The primary factor in making these choices was efficiency: only interruptible locks require "deluxe" wait lists supporting interior removal, and only the readers' wait lists require high concurrency. The `rpri` and `wpri` fields specify the scheduling priority to be assigned to readers and writers, respectively. The priority doesn't affect the service order for the lock, but does affect the service order in the system ready queue, to which awakened activities are eventually moved.

The complete list of operations supported is given in Table 25, on the next page.

	Non-Interruptible	*Interruptible*
readers	`afifo` *(almost FIFO)*	`dafifo` *(deluxe almost FIFO)*
writers	`llist` *(linked list)*	`dllist` *(deluxe linked list)*

Table 24: Wait Lists for Readers/Writers Synchronization.

Specific list types are described in section 3.7.1 on page 82.

Operation	*Non-interruptible*	*Interruptible*	*See Page*
Initialize	`csrw_init`	`icsrw_init`	173
Destroy	`csrw_destroy`	`icsrw_destroy`	173
Obtain read lock	`csrw_rlock` †	`icsrw_rlock` †	174,177
Assured success	`csrw_qrlock` •	`icsrw_qrlock` •	183
Nonblocking attempt		`icsrw_tryrlock` •	183
Get lock status	`csrw_isrlocked` •	`icsrw_isrlocked` •	184
Release read lock	`csrw_runlock` †	`icsrw_runlock` †	175,179
Obtain write lock	`csrw_wlock` †	`icsrw_wlock` †	175,180
Assured success	`csrw_qwlock` •	`icsrw_qwlock` •	183
Nonblocking attempt		`icsrw_trywlock` •	183
Get lock status	`csrw_iswlocked` •	`icsrw_iswlocked` •	184
Release write lock	`csrw_wunlock` †	`icsrw_wunlock` †	176,181
Special purpose functions			
Interrupt waiting reader		`_icsrw_runblock` •	186
Interrupt waiting writer		`_icsrw_wunblock` •	185
Wakeup next reader	`_csrw_rwakeup`	`_icsrw_rwakeup`	182

† First stage may be inlined.
• Function may be fully inlined.

Table 25: Context-Switching Readers/Writers Functions.

The “get lock status” functions have no effect other than returning a boolean indicator of whether a normal lock attempt of the corresponding type would block. The nonblocking attempt is semantically equivalent to a normal attempt that is immediately disturbed if it blocks. The last category of functions given are not generally called directly: `_icsrw_runblock` and `_icsrw_wunblock` are called only by the `unblock` function (§4.5.3/p185), while `_csrw_rwakeup` and `_icsrw_rwakeup` are called by the scheduler (§4.6.9).

Initialization and Destruction

Initialization and destruction are simple; the former is mostly a matter of assigning values to fields.

```
int
icsrw_init (icsrwlock *rw)
{
      if (!mqiinit(&rw->mqi,_icsrw_rwakeup))
            return 0;
      rw->acount = 0;
      rw->rrcount = 0;
      if (!dafinit(&rw->rwlist)) {
            mqidestroy(&rw->mqi);
            return 0;
      }
      if (!dllinit(&rw->wwlist)) {
            dafdestroy(&rw->rwlist);
            mqidestroy(&rw->mqi);
            return 0;
      }
      return 1;
}
```

Each lock is set up in the unlocked state; this is reflected in the initial values for `acount` and `rrcount`. The major effort of `icsrw_init` is to initialize the multiqueue list item (`mqiinit`) and the wait lists (`dafinit` and `dllinit`). The multiqueue list item, `mqi`, allows an entire group of blocked readers to be inserted onto the ready list in a single operation (§3.7.1/p81). Each time the item is "deleted" from the ready list, the function `_icsrw_rwakeup` (p182) is called to get the next blocked reader from the readers waiting list, `rwlist`.

While the waiting list implementations described in section 3.7.1 on page 82 and their underlying busy-waiting synchronization mechanisms (§3.5) don't actually admit the possibility of failure during initialization, they allow for it in their interface, and we allow for it here as well, for the sake of independence.

Lock destruction is very simple, requiring only the destruction of the more complex subcomponents:

```
void
icsrw_destroy (icsrwlock *rw)
{
      mqidestroy(&rw->mqi);
      dafdestroy(&rw->rwlist);
      dlldestroy(&rw->wwlist);
}
```

First Stage Reader Lock and Unlock

The most important operations are the acquisition and release of read locks, because they are the only ones that have a chance to be completely serialization free. Also, recall that readers/writers locks are most appropriate for situations in which one expects the large majority of accesses to be reads. Fortunately, the overhead of these common operations can be kept quite low.

Here is the first stage routine for `icsrw_rlock`:

```
int
icsrw_rlock (icsrwlock *rw)
{
      struct activity *a = curact();

      a->a_pri = rw->rpri;
      if (a->a_flags & ACT_CANCEL)
            return _act_clearcancel(a);
      if (fai(&rw->acount) >= ICS_MAXR)
            return _icsrw_rlock(rw,a);      // call second stage
      return 1;
}
```

The function `curact()` is really a macro that returns a pointer to the *current activity*, the one requesting the lock. The activity's scheduling priority is set to `rw->rpri` regardless of whether or not the activity must block. This can be important in Symunix-2 since preemption can be permitted at almost any point in the kernel, even when holding a context-switching lock.

The basic operation required to obtain a read lock is to increment `acount` with Fetch&Increment; the result indicates whether the lock is obtained or blocking is necessary. Only when blocking is necessary is the second stage, `_icsrw_rlock`, called.

The `ACT_CANCEL` flag in the activity structure is set by `syscancel` (§4.3.1/p154) or by receipt of a signal during certain synchronous system calls (§4.3.1/p153). The `_act_clearcancel` function clears the `ACT_CANCEL` flag and returns 0; it is a separate routine simply to minimize the memory expansion cost of inlining the first stage routine:

```
int
_act_clearcancel (struct activity *act)
{
      spl_t s = splsoft();
      bwl_wait (&act->a_lock, splcheckf(s));
      act->a_flags &= ~ACT_CANCEL;
      bwl_signal (&act->a_lock);
      return 0;
}
```

Clearing of the `ACT_CANCEL` flag must be protected by locks (§3.5), since it is accessed asynchronously by other activities and interrupt handlers (as are the other bits in `act->a_flags`). The use of explicit locking is necessary even if an operation such as Fetch&And is available (and combinable—as it is on the Ultra-3 prototype), because the operation must be atomic with respect to activity state changes.

By examining `icsrw_rlock`, it is clear that, in the absence of write lock requests or cancelation, only four data memory accesses are required: two loads, one store, and one Fetch&Increment. (We are assuming `curact()` is implemented with a single cacheable load as on the Ultra-3 prototype; other reasonable implementations range from 0 to 2 loads, all of which could easily be directed to cache or local memory.) For the non-interruptible version, `csrw_rlock`, one of the loads (`a->a_flags`) is not required, representing a further savings of 25% or more.

Here is the first stage function to release a reader lock:

```
void
icsrw_runlock (icsrwlock *rw)
{
      if (fad(&rw->acount) > ICS_MAXR &&
            fai(&rw->rrcount) == ICS_MAXR-1)
            _icsrw_runlock(rw);              // call second stage
}
```

The Fetch&Decrement of `acount` simultaneously releases the reader lock and indicates the presence or absence of waiting writers. The Fetch&Increment of `rrcount` identifies the last unlocking reader, with the help of `_icsrw_wlock`, on page 180. The last unlocking reader calls the second stage function, on page 179.

First Stage Writer Lock and Unlock

The first stage writer lock functions are complementary to the reader lock functions:

```
int
icsrw_wlock (icsrwlock *rw)
{
      struct activity *a = curact();
      int v;

      a->a_pri = rw->wpri;
      if (a->a_flags & ACT_CANCEL)
            return _act_clearcancel(a);
      v = faa(&rw->acount,ICS_MAXR);
      if (v != 0)
            return _icsrw_wlock(rw,v,a);  // call second stage
      return 1;
}
```

Comparing `icsrw_rlock` and `icsrw_wlock`, we see that there are only three significant structural differences between the first stage reader and writer lock functions:

(1) The value added to `acount` is 1 for a reader lock and `ICS_MAXR`, a large constant, for a writer lock.

(2) For a writer lock, the old `acount` value must be passed to the second stage function, `_icsrw_wlock`, so that it may distinguish the first waiting writer from all others.

(3) The comparison made to determine initial success or failure is more restrictive for a writer, allowing only exclusive access.

We can see that, in the absence of any other writers or readers, or cancelation, the cost (measured in shared memory references) of obtaining a writer lock is the same as a reader lock (four data memory accesses, p174). In fact, the uncontended cost of our simple binary semaphores (`cslock` and `icslock`) is also the same.

The first stage function to release a writer lock is very simple:

```
void
icsrw_wunlock (icsrwlock *rw)
{
      int v = faa (&rw->acount, -ICS_MAXR);
      if (v > ICS_MAXR)
           _icsrw_wunlock(rw,v);     // call second stage
}
```

Because writers have exclusive access, there is no need to determine the last unlocking writer, hence there is no equivalent of `rrcount` for writers.

Second Stage Reader Lock and Unlock

We now turn our attention to the second stage functions to complete the basic operations of locking and unlocking. Again, the reader functions are the most crucial, since they have the chance to be completely serialization-free.

Whereas the first stage functions are closely related to the busy-waiting algorithm (§3.5.4) in that a single Fetch&Add operation is generally the center of attention, the second stage routines are focused on scheduling and context-switching.

Recall that by the time the second stage reader lock routine is called, the decision to block has already been made (p174). In general terms, all that remains is to get the activity on the waiting list and reschedule, but certain details complicate things:

- *The implementation of asynchronous system calls.* Recall from section 4.3.1 on page 152 that a logically asynchronous system call normally executes in a synchronous fashion so long as blocking is unnecessary. The user may also request that such synchronous behavior continue so long as the process has no signal to handle. In either case, it is possible that a new primary activity must be set up before blocking.
- *The possibility of cancelation.* We already saw that the first level locking function checks the `ACT_CANCEL` flag; this is adequate if the flag is clear or set long before the lock request. But cancelation must generally be synchronized with activity state changes, so we have to check again, *after* locking the activity.[81] Likewise, code that sets `ACT_CANCEL` does so only with the activity locked.
- *Keeping the counter from overflowing*, a possibility brought about by premature unblocking. When a blocked activity is disturbed, the initial adjustment made to the counter in the first stage lock function is not altered, so the state of the lock appears as if the activity were still blocked. The fact of the interruption is recorded as a "hole" in the waiting list, and the appropriate compensation is made when the hole is "deleted" from the front of the list. It is possible for a long sequence of lock attempts to be interrupted while a single activity holds the lock. It is even possible, although unlikely, for this sequence to be long enough that the increments in `icsrw_rlock` would cause `acount` to overflow. Our strategy is to detect imminent overflow, and revert to non-interruptible behavior in such unlikely situations.

[81] The activity lock is a busy-waiting lock (§4.4.2).

Here is the code for the second stage reader lock function:

```
int
_icsrw_rlock (icsrwlock *rw, int old_acount, struct activity *a)
{
     struct activity *n;
     spl_t s;
     int f, pf;

     dafireinit(&a->a_li.daf);
     a->a_waitfor.irw = rw;
     if (old_acount < (ICS_MAXR-ACT_NUMACT)*ICS_MAXR)
          a->a_waittype = ACT_ICSRWR | ACT_MAYCANCEL;
     else
          a->a_waittype = ACT_ICSRWR;       // avoid overflow of acount
     pf = a->a_pflags;   // private flags; no locking required
     s = splsoft();
     bwl_wait (&a->a_lock, splcheckf(s));

     f = a->a_flags;
     if (f&ACT_CANCEL && old_acount < (ICS_MAXR-ACT_NUMACT)*ICS_MAXR) {
          a->a_flags = f & ~ACT_CANCEL;
          bwl_signal(&a->a_lock);
          dafputhole(&rw->rwlist);
          vsplx(s);
          return 0;
     }
     n = _cs_common (a, pf);
     dafput(&rw->rwlist, &a->a_li.daf);
     bwl_signal(&a->a_lock);
     vsplx(s);
     cswitch(n ? n : resched(0));

     if (a->a_pflags & ACT_DISTURBED) {
          a->a_pflags &= ~ACT_DISTURBED;
          return 0;
     }
     return 1;
}
```

The function `dafireinit` reinitializes the list item structure for activity so that it can be used for insertion onto the `dafifo` waiting list. The activity fields `a_waitfor` and `a_waittype` are set for general information purposes, but mostly to support cancelation. The value `(ICS_MAXR-ACT_NUMACT)*ICS_MAXR` is the threshold for counter overflow detection; if the value before incrementing in the first stage was above this threshold, we resort to non-interruptible behavior. Above this level, we can still handle every activity trying to get a writer lock. The function `dafputhole` inserts a “hole” onto the waiting list, just as if the activity were inserted and then removed. In the absence of cancelation, `dafput` is called for an ordinary insertion.

The function `_cs_common`, which is also used for the other list-based context-switching mechanisms, sets up a new primary activity, if necessary, allowing the current one to continue asynchronously handling a system call or page fault. If a new activity is to be started, it is simply "handed off" to the primitive context-switch function `cswitch`; otherwise, `resched` is called to choose the next activity to run. Upon being awakened, the `ACT_DISTURBED` flag is checked to see if cancelation occurred after blocking. The code for `_cs_common` follows:

```
struct activity *
_cs_common (struct activity *a, int pflags)
{
      struct activity *n;
      struct proc *p;
      genarg_t x;

      if ((pflags & ACT_OPTIMASYNC) == 0 ||
          ((pflags & ACT_AUTOWAIT) != 0 && !issig(a->a_proc)))
           n = NULL;
      else if ((n = new_act(p = a->a_proc)) != NULL) {
           if ((pflags & ACT_UPFAULT) != 0) {
                x.vd = a;
                csynth (a, n, p, retapfault, x);
           }
           else {
                x.i = 0;
                csynth (a, n, p, retasyscall, x);
           }
      }
      a->a_status = ACT_BLOCKED;
      return n;
}
```

The activity is marked with the `ACT_OPTIMASYNC` flag whenever a system call or page fault is begun that is logically considered asynchronous. This is essentially a lazy strategy, done in the hope that the activity's work can be done without blocking. If blocking eventually proves to be necessary (i.e., we get to `_cs_common`), we finally incur the overhead of creating a new primary activity (§4.3.1/p153). A variant of this behavior is available by setting the `ACT_AUTOWAIT` flag (corresponding to the meta system call `SCALL__AUTOWAIT` flag (§4.3.1/p152)), causing the system call to operate synchronously unless a signal is to be handled, indicated here by the `issig` function.

There is the possibility of failure when allocating a new activity structure, e.g., the memory available for allocating such structures may be exhausted. In case of such failure, we revert to synchronous behavior, but this is not acceptable for system calls, where correct operation may depend on user-level signal handling. We prevent `new_act` from failing by pre-allocating an activity structure when the system call is begun, to ensure its availability when needed. Pre-allocation, e.g., by keeping a few structures associated with each process, is cheap; most of the expense of setting up new activities is in the `csynth` function, which initializes the activity to begin execution by calling the indicated startup function. Different `csynth` startup functions are used for page faults and system calls, which can be distinguished by checking the ACT_UPFAULT activity flag. The fifth argument to cssynth is a union of basic data types; the value passed in each case is expected by the indicated function.

The final small action of _cs_common is one of the most crucial for the second stage: setting the activity's status to ACT_BLOCKED.

The second stage function for releasing a reader lock is only called when the first stage determines that it may be necessary to wakeup a waiting writer. The function is simple:

```
void
_icsrw_runlock (icsrwlock *rw)
{
      dllitem *di;
      rw->rrcount = 0;
      FBUSY_WAIT ((di = dllget(&rw->wwlist)) != NULL, nullf, &iwwakehg);
      if (di == &dllhole)
            icsrw_wunlock(rw);
      else
            mkready ((struct activity *)((char *)di -
                       offsetof(struct activity, a_li)));
}
```

Because the first stage function has incremented rrcount to identify the last running reader, we must reset it to zero, for the next cycle of readers. A simple assignment suffices, since there can be no further modifications by other activities. There are basically two steps required to complete the lock release:

(1) Obtain a writer to unblock. The most critical work, resolving the race between a blocking writer and the last running reader, is handled by the FBUSY_WAIT macro (§3.5.1/p52). This macro expands to a loop that executes until the first argument evaluates to *TRUE*, in this case until the call to the list deletion function dllget returns a non-NULL pointer to a blocked writer. The second argument is a pointer to a *check function*, to be executed periodically while busy-waiting, but in this case no function is needed, so we simply pass a special empty function, nullf.

(2) Unblock the writer obtained in step 1, or release the write lock just granted to the interrupted writer, if a hole was found.[82] For a real writer, this is just a call to mkready (§4.6.9). The argument to mkready is a portable way to derive an activity pointer for the writer from a pointer to the list item structure obtained from dllget.

Second Stage Writer Lock and Unlock

Much of the same structure as we have seen in _icsrw_rlock is also applicable for writer locks. The main difference is the need to deal with the rrcount field, which determines

[82] Being a first stage function, icsrw_wunlock may be inlined within the second stage function _icsrw_runlock. Inlining is unlikely to be very helpful in this case, assuming that premature unblocking is fairly rare, but the expansion cost is small enough that it isn't worth much trouble to avoid.

when the last reader has relinquished its lock, so the first writer may proceed:

```
int
_icsrw_wlock (icsrwlock *rw, int old_acount, struct activity *a)
{
     struct activity *n;
     spl_t s;
     int f, pf;

     if (old_acount < ICS_MAXR &&
         faa(&rw->rrcount,ICS_MAXR-old_acount) == old_acount)
          rw->rrcount = 0;
     else {
          dllireinit(&a->a_li.dll);
          a->a_waitfor.irw = rw;
          if (old_acount < (ICS_MAXR-ACT_NUMACT)*ICS_MAXR)
               a->a_waittype = ACT_ICSRWW | ACT_MAYCANCEL;
          else
               a->a_waittype = ACT_ICSRWW;
          pf = a->a_pflags;
          s = splsoft();
          bwl_wait (&a->a_lock, splcheckf(s));

          f = a->a_flags;
          if (f&ACT_CANCEL &&
              old_acount < (ICS_MAXR-ACT_NUMACT)*ICS_MAXR) {
               a->a_flags = f & ~ACT_CANCEL;
               bwl_signal(&a->a_lock);
               dllputhole(&rw->wwlist);
               vsplx(s);
               return 0;
          }
          n = _cs_common (a, pf);
          dllput(&rw->wwlist, &a->a_li.dll);
          bwl_signal(&a->a_lock);
          vsplx(s);
          cswitch(n ? n : resched(0));

          if (a->a_flags & ACT_DISTURBED) {
               a->a_flags &= ~ACT_DISTURBED;
               return 0;
          }
     }
     return 1;
}
```

The basic idea is that when the first writer makes its presence known by adding `ICS_MAXR` to `acount`, the number of currently running readers is also determined (in this function, it is `old_acount`, the prior value of `acount`, returned by Fetch&Add in `icsrw_wlock` (p175)).

All of these running readers must release their locks before the first writer may proceed. As each does so, in `icsrw_runlock` (p175), it increments `rrcount` with Fetch&Increment. The first writer adds `ICS_MAXR-old_acount`, so the value of `rrcount` will be `ICS_MAXR` when it is safe for the writer to proceed. This value may be reached by the last reader or by the first writer, the race is easily resolved by having both check for the final value. Whoever reaches the final value resets `rrcount` to 0.

The second stage writer unlock function is a bit more complicated than the corresponding reader unlock function, because of the need to unblock either one writer or all readers:

```
void
_icsrw_wunlock (icsrwlock *rw, int old_acount)
{
again:
     if (old_acount >= 2*ICS_MAXR) {   // wake up another writer
          dllitem *di;
          FBUSY_WAIT ((di = dllget(&rw->wwlist)) != NULL,
                      nullf, &iwwakehg);
          if (di != &dllhole)
               mkready ((struct activity *)di);
          else {
               old_acount = faa (&rw->acount, -ICS_MAXR);
               if (old_acount > ICS_MAXR)
                    goto again;
          }
     }
     else if (old_acount-ICS_MAXR >= MINMKQREADY)
          mkqready(&rw->mqi, rw->rpri, old_acount-ICS_MAXR);
     else {
          do {    // wake up readers one at a time
               dafitem *di;
               FBUSY_WAIT ((di = dafget(&rw->rwlist)) != NULL,
                           nullf, &irwakehg);
               if (di != &dafhole)
                    mkready ((struct activity *)di);
               else
                    icsrw_runlock(rw);
          } while (--old_acount > CS_MAXR);
     }
}
```

Because `acount` is incremented by `ICS_MAXR` for each writer, and by 1 for each reader, it is easy to determine the number of waiting readers and writers by examining `old_acount`. There are three cases:

(1) If `old_acount` is at least `2*ICS_MAXR`, at least one other writer is waiting. Since the algorithm always gives priority to writers over readers, this condition is checked first. The work to do in this case is essentially the same as in the second stage reader unlock function.

(2) If `old_acount` is less than `2*ICS_MAXR`, any excess over `ICS_MAXR` is due to waiting readers (note that `old_acount` must be at least `ICS_MAXR` due to the unlocking writer itself). For efficiency, we treat two separate cases, depending on the number of waiting readers. (2A) If the number is large enough (`MINMKQREADY` or more), the quickest and most efficient wakeup will be achieved by inserting the entire `icsrwlock` structure on a multiqueue component of the global ready list (§4.6.7). The work is done by a variant of `mkready`, called `mkqready`; we have already seen the initialization of the `mqi` field (p173), including a pointer to `_icsrw_rwakeup`, a function to be described below. The number of readers awakened is, as already explained, `old_acount-ICS_MAXR`.

(2B) When the number of waiting readers is small, it may be more efficient to awaken them one by one. This is because the highly parallel multiqueue algorithms may have higher overhead than singleton queues (§3.7.1/p82). Serial awakening of readers is accomplished by enclosing within a `do` loop code similar to that already seen for unblocking writers.

The multiqueue item structure, `mqitem`, contains a pointer to a function to be called by `mqget` whenever the item is encountered. The function must return a pointer to another `mqitem` structure, and is intended to allow abstract structures to be placed on the system ready list along with ordinary activities. We have already seen that the readers/writers lock structure contains an `mqitem`, and that the function pointer is set to `_icsrw_rwakeup` during lock initialization (p173), and we have seen how the item is inserted onto the global ready list by calling `mkqready`. Now we shall examine `icsrw_rwakeup` itself:

```
mqitem *
_icsrw_rwakeup (mqitem *mqi)
{
      spl_t s;
      icsrwlock *rw;
      dafitem *di;
      struct activity *a;

      rw = (icsrwlock *)((char *)mqi - offsetof(icsrwlock,mqi));
      FBUSY_WAIT ((di = dafget(&rw->rwlist)) != NULL, nullf, &irwakehg);
      if (di == &dafhole) {
            icsrw_runlock(rw);
            return NULL;
      }
      a = (struct activity *)((char *)di -
                  offsetof(struct activity, a_li.mq));
      s = splsoft();
      bwl_wait (&a->a_lock, splcheckf(s));
      a->a_status = ACT_READY;
      bwl_signal (&a->a_lock);
      vsplx(s);
      return &a->a_li.mq;
}
```

The expression including `offsetof` is the portable way of obtaining a pointer to a structure given a pointer to one of its members. The rest of this function resembles the code in `_icsrw_wunlock` to awaken a single reader, except that `mkready` isn't called (because the activity *is* ready: it just came off the ready list).

Variant Lock Functions

There are two variants of each basic locking routine, *guaranteed* and *conditional*. Guaranteed versions are for situations where higher level logic dictates that the lock can always be granted without delay. The major difference between a normal lock request and a guaranteed one is that the latter is unprepared for blocking. The main reason for using the guaranteed version instead of the normal one is clarity of purpose: it acts as an assertion of what the programmer expects. For example, it is safe to hold busy-waiting locks when calling a guaranteed locking function (this situation will arise in section 3.7.6 on page 130, where `csrw_qwlock` would be appropriate).

```
void
icsrw_qrlock (icsrwlock *rw)
{
      if (fai(&rw->acount) >= ICS_MAXR)
            panic("icsrw_qrlock");  // impossible
}
```

Because it is guaranteed not to block, we add the letter `q`, for “quick”, to the function name. To further exploit the low cost nature of the function, we also omit the check for cancelation, i.e., we are not viewing the guaranteed lock function as a significant synchronization point, worthy of also being a failure point. The corresponding write lock function is almost identical; the major differences are that the increment is `ICS_MAXR` instead of 1, and the old `acount` value must be 0 instead of merely < `ICS_MAXR`. The guaranteed locking functions for non-interruptible mechanisms are essentially the same.

A conditional synchronization function is one that operates successfully if and only if it can be done without blocking, otherwise it fails. In general, the requirement to support conditional locking functions constrains the algorithm design space, so we don't have conditional locking functions for our non-interruptible mechanisms. Our implementation relies on the waiting list's support for explicit *hole* insertion:

```
int
icsrw_tryrlock (icsrwlock *rw)
{
      if (rw->acount >= ICS_MAXR - ACT_NUMACT)
            return 0;      // avoid overflow
      if (fai(&rw->acount) < ICS_MAXR)
            return 1;
      dafputhole (&rw->rwlist);
      return 0;
}
```

The conditional writer lock is essentially the same. We saw more about how holes are handled in the second stage functions on page 179. Most of the peculiarity of `icsrw_tryrlock` is dedicated to avoiding overflow of acount. Imagine, for example, that a writer holds the lock for a very long time, and at least one potential reader calls icsrw_tryrlock repeatedly, in a loop (i.e., uses the context-switching conditional lock as a busy-waiting lock). If icsrw_tryrlock didn't have the anti-overflow protection, acount could easily accumulate enough would-be reader increments to look as though another activity was trying to obtain a write lock. ACT_NUMACT is the maximum number of activities. The anti-overflow check ensures that each activity in the system can have at least one additional pending read lock request, although interruptibility may be sacrificed. Because the check is conservative, it is possible for a conditional lock request to fail unnecessarily, but this is unlikely if ICS_MAXR is much greater than ACT_NUMACT. This overflow avoidance matches that in the second stage functions in section 4.5.3 on page 186.

Occasionally it is useful to access the lock value directly; we provide boolean functions for this purpose, e.g.:

```
int
icsrw_isrlocked (icsrwlock *rw)
{
      return rw->acount >= ICS_MAXR;
}

int
icsrw_iswlocked (icsrwlock *rw)
{
      return rw->acount != 0;
}
```

4.5.3. Premature Unblocking

We have already seen how the interruptible first stage locking functions check for the ACT_CANCEL flag, and how the second stage functions check again after locking the activity. But what happens if the activity is already blocked when the cancelation is needed? This is handled by the generic unblock function. This function can prematurely unblock any locked activity with a_status == ACT_BLOCKED and (a_waittype & ACT_MAYCANCEL) != 0; this includes not only the list-based synchronization forms such as readers/writers locks, but also uses of idopause:

```
int
unblock (struct activity *act)
{
      int r;
      int wtype = act->a_waittype;
      if ((wtype & ACT_MAYCANCEL) == 0)
            return 0;                   // not cancelable
      switch (wtype) {
      default:
            panic ("unblock");          // can't happen
      case ACT_MAYCANCEL|ACT_ICSSEM:
            r = _icss_unblock(act);     break;
      case ACT_MAYCANCEL|ACT_ICSLOCK:
            r = _icsl_unblock(act);     break;
      case ACT_MAYCANCEL|ACT_ICSRWR:
            r = _icsrw_runblock(act);   break;
      case ACT_MAYCANCEL|ACT_ICSRWW:
            r = _icsrw_wunblock(act);   break;
      case ACT_MAYCANCEL|ACT_ICSEVENT:
            r = _icse_unblock(act);     break;
      case ACT_MAYCANCEL|ACT_KSEM:
            r = ksem_unblock(act);      break;
      case ACT_MAYCANCEL|ACT_SYSWAIT:
      case ACT_MAYCANCEL|ACT_PSUSPEND:
      case ACT_MAYCANCEL|ACT_JSUSPEND:
      case ACT_MAYCANCEL|ACT_TSUSPEND:
      case ACT_MAYCANCEL|ACT_WAIT:
            r = 1;                      break;
      }
      if (r)
            act->a_pflags |= ACT_DISTURBED;
      return r;
}
```

For activities blocked in `idopause` (i.e., `act->a_waittype` is `ACT_MAYCANCEL|ACT_SYSWAIT`), the only thing to do is set the `ACT_DISTURBED` private flag in the activity structure, since there is no waiting list or other state to adjust. For list-based synchronization mechanisms, `unblock` calls a synchronization-specific function to do the real work of removing the activity from the waiting list and, if that is successful, sets the `ACT_DISTURBED` flag. The activity's `a_waittype` field selects the action to be performed; every possibility must be represented in the switch statement. We saw how the second stage lock functions (§4.5.2/p177) and `idopause` (§4.5.1/p170) deal with the `ACT_DISTURBED` flag.

The synchronization specific unblock functions are normally inlined for efficiency, since `unblock` is the only place they're called. What do these functions look like? Here is the one for readers; most of the others have a similar structure:

```
int
_icsrw_runblock (struct activity *a)
{
      icsrwlock *rw = a->a_waitfor.irw;
      return dafremove (&rw->rwlist, &a->a_li.daf);
}
```

A more complex example is `ksem_unblock`, given in section 7.3.4 on page 292.

4.6. Scheduling

In a multiprogrammed operating system, processors are *scheduled* to run processes (or tasks, threads, activities, etc., depending on local terminology). The problem can be viewed in different ways but, for most general-purpose operating systems, scheduling can be divided into at least two phases: *short-term* and *long-term* scheduling. Short-term scheduling is concerned with selecting which processes (tasks, threads, activities, ...) are selected to run from among those that are ready. Long-term scheduling is concerned with selecting which processes (tasks, threads, activities, ...) are given a chance to be ready, e.g., admitted to the system, swapped into main memory, or simply considered eligible to run. In some cases, one scheduling phase is degenerate, such as an interactive system that accepts all job submissions, or a first-come first-served batch system.[83]

Most of our attention to scheduling is directed at short-term scheduling, but the importance of long-term scheduling should not be dismissed. There are several aspects to short-term activity/processor scheduling in Symunix-2:

- Selecting another activity to run when an activity blocks or terminates.
- Maintaining the *ready list*, a data structure to keep track of all activities that are ready to run, and arranging for the execution of an activity when it is created or unblocked.
- Preempting a running activity in favor of another with better or similar priority.
- Consideration for the interdependent scheduling needs of the processes comprising a parallel application.

Before we can discuss scheduler design and policy alternatives, we must present the basic mechanisms for preemption and interdependent scheduling.

4.6.1. Preemption

Traditional UNIX kernels, including Symunix-1, use a *polling* approach for process preemption: Each processor periodically executes code to check for a possible preemption and choose a new process to run if appropriate. The check can be performed explicitly at any point, but usually only after executing a system call or when returning to user-mode after handling certain device interrupts, such as a clock interrupt. *True preemption* (preemption at almost any point) is prohibited within the kernel itself.

[83] Some authors reserve *long-term* scheduling to refer only to admission of new jobs into the system, and use *medium-term* for mechanisms that reduce the effective multiprogramming load of those already admitted, e.g., swapping. Our notion of long-term scheduling includes both of these categories.

Although the preemption mechanism itself is somewhat complicated by the prohibition against true kernel preemption, the rest of the kernel in a uniprocessor or master/slave system can enjoy the benefit of freely ignoring preemption issues. However, some of this advantage disappears from symmetric multiprocessor kernels where most of the kernel must be aware of the concurrent effects of other processors anyway. Thus, allowing preemption generally throughout such kernels is not a serious additional complication. Of course preemption cannot be allowed at *any* point within the kernel, e.g. when holding busy-waiting locks, but suppressing it is easy since, by definition, the kernel has full control over the machine.

Following this reasoning, Symunix-2 allows true kernel preemption. The specific approach used is to implement preemption as the lowest priority soft interrupt.[84] In this way, kernel preemption may be deferred in exactly the same way as other interrupts (§3.4/p47), and because preemption has the lowest priority, it is normally masked and unmasked "for free" along with other interrupts. Another benefit of making preemption the *lowest* priority interrupt is that a single dedicated interrupt stack can still be maintained for each processor: when preemption does occur, the interrupt stack is empty except for the preemption interrupt itself. In addition, by eliminating one function out of two (preemption and soft interrupts), machine-dependent interrupt and trap handling code is streamlined.

The function to handle soft preemption interrupts is called `softresched`; the interrupt may be masked by `splsoftresched`, and initiated by `setsoftresched`. Recall that pending interrupts and masks in Symunix-2 are considered part of each processor's state (§3.4); they do not form a central pool accessed by all processors,[85] and they don't migrate from processor to processor with activities or processes.

Conceptually, `softresched` is quite simple; it calls `resched` (§4.6.8) and, if `resched` returns non-`NULL`, it calls `cswitch` (§4.4.5).

The implementation of `softresched` is machine-dependent, to allow reasonable implementation options such as use of a dedicated interrupt stack for each processor. Reasons for using an interrupt stack include:

- Minimizing the (fixed) size of the activity stack. Since the activity stack cannot be swapped out of memory and each activity has one, this can be important.
- Hardware that explicitly maintains a separate interrupt stack, e.g., the DEC VAX.
- The presence of local memory associated with each processor. Such memory may be faster than global memory, and accesses to it may be contention free, or may impose less contention on other processors. The memory need not be accessible to other processors.

Whatever the reason, the use of a dedicated interrupt stack requires that `softresched` be implemented very carefully. By the time `cswitch` is called, the portion of the interrupt stack

---

[84] Recall from section 3.4 on page 46 that *soft interrupts* are directly caused by software, rather than a hardware device such as a disk drive or clock.

[85] This primarily reflects the way interrupt handling is designed into most, but not all, commercial microprocessors. Some machines, like the Sequent Balance [19, 80] and Symmetry machines [139], and some Intel Pentium-based machines [200], have special hardware to distribute device interrupts automatically to the processor running at the lowest priority, but masked and pending interrupt status are still "per processor" entities.

containing the state of the interrupted processor execution of `softresched` must be moved to the regular activity stack. (Recall that `cswitch` saves and restores the processor state to and from the activity stack (§4.4.5).) Fortunately `softresched` is the lowest priority interrupt and is thus guaranteed to be the sole contents of the interrupt stack. We assume the organization of saved state on the interrupt stack is well enough defined to make the writing of `softresched` a tractable problem, albeit a delicate machine-dependent one.

4.6.2. Interdependent Scheduling

Scheduling has traditionally been transparent in UNIX, that is, with the exception of the `nice` system call (`setpriority` in 4.2BSD and successors), there have been essentially no scheduling controls available to the programmer or system administrator. Scheduling has been performed on the basis of priorities that are adjusted dynamically according to resource usage and system load. This is a generally good design considering the traditional UNIX time-sharing workload; unnecessary controls on a system should be avoided. However, with the introduction of parallel applications, new factors emerge that make the traditional approach less desirable:

- *Frequent synchronization between processes.* From the point of view of a single parallel application running on a multiprocessor, the most efficient way to perform synchronization is to run all synchronizing processes simultaneously, and allow them to *busy-wait* as necessary during the synchronization. Most parallel architectures directly support an efficient hardware mechanism to do this, avoiding the overhead associated with blocking a process and context-switching to another one. Unfortunately, even multiprocessors don't always have enough processors, so some form of process preemption is still needed. The problem is that busy-waiting in the presence of preemption can be very bad for performance. For example, a process might busy-wait a substantial amount of time for another process that has been preempted, creating a feedback effect by performing useless work that effectively raises the system load, in turn causing more preemption. This problem arises because the scheduler isn't aware of user synchronization (Zahorjan, *et al.* [215]).
- *Uneven rates of execution.* Some parallel algorithms perform poorly when the instruction streams don't progress at approximately the same rate. While "even" progress is a property of any "fair" scheduler over a sufficiently long period of time, some algorithms have less tolerance and require fairness over shorter time periods. An example is an algorithm that uses *barrier synchronization* (Jordan [122]) frequently. In such a situation, the progress of the overall algorithm is limited by the progress of the last process to reach each barrier.
- *Complete program control over resource scheduling.* Beyond the problems we have just described, where lack of control over preemption can cause serious performance problems, there are applications that ask almost nothing of the operating system at all, except that it not interfere. What is often wanted by the users for such applications is a long-term commitment of resources (e.g. processors, memory), and, given that they are willing to pay the price for such commitment, they expect to get performance near the absolute maximum deliverable by the hardware, with no operating system overhead. Such control may not be appropriate on small machines or uniprocessors, but becomes increasingly important on large high-performance multiprocessors.
- *user-mode thread systems.* Some applications and parallel programming environments benefit from being structured as a set of independent *threads of control*, operating

essentially within a single address space and sharing access to all resources. For fundamental reasons, the most efficient implementations of creation, destruction, synchronization, and context-switching for threads can only be done primarily in user-mode (§4.1/p147). Such a system can be efficiently implemented in the manner of the previous bullet, but we will show less drastic alternatives.

The common theme running through all these problems is preemptive scheduling of processes, and we now propose three new mechanisms to eliminate or control it: *scheduling groups*, *temporary non-preemption*, and the `SIGPREEMPT` signal.

4.6.3. Scheduling Groups

We introduce a new process aggregate, the *scheduling group*, which is identified by a *scheduling group ID*. Every process is a member of a scheduling group, although that group may consist of only the single process itself. In many cases, we expect scheduling groups to correspond to families (§4.2/p148). A scheduling group is subject to one of the following scheduling policies:

- *Fully Preemptable*. This is the traditional scheduling policy of UNIX, and is the default. This policy is appropriate for timesharing and for parallel applications with little or no synchronization.
- *Group Preemptable*. The scheduling group members may only be preempted or rescheduled at (approximately) the same time. This is similar to Ousterhout's coscheduling of task forces [160], but is concerned with preemption rather than blocking. This policy is appropriate for parallel applications with moderately frequent synchronization, and a fairly stable amount of parallelism.
- *Non-Preemptable*. The scheduling group members are immune from normal preemption. This policy is appropriate for parallel applications with the primary requirement for "raw" processors, without interference from the kernel. It is the most "expensive" policy, because of its unwillingness to share processors among multiple users, but can offer unequaled performance when combined with long-term memory allocation.

Since it is necessary for all members of a group- or non-preemptable group to execute simultaneously, it must be possible to meet their combined resource requirements. It's obvious that uniprocessors can't provide these new scheduling policies for groups with more than one process, but multiprocessors also have only a finite supply of processors and memory and, since spawning a new process into such a group or requesting more memory are operations that increase the resource requirements of the group, it is possible for them to fail. To avoid this problem, we must provide for long-term allocation of processors and memory. Because of their long-term resource allocations, group- and non-preemptable scheduling policies must be restricted by system administrative policy and implemented by a long-term component of the scheduler. In addition to new system calls for manipulation of scheduling group IDs, new calls are needed to select the scheduling policy of a group and to request long-term allocation of processors and memory.

These new mechanisms directly support multiprocessor users whose primary objective is to "get rid of" operating system overhead, and solve the other problems outlined on pages 188−189, provided that the users are willing to pay the cost−which may include restrictions on resource utilization, increased average turnaround time and, in some cases, a negative impact on interactive or real-time responsiveness.

The group- and non-preemptable scheduling policies have no effect on voluntary blockages (e.g. when a process issues a system call for which it is necessary to block). When a process is blocked, other members of the group will continue to be run, preempted, and rescheduled according to the current policy. Note also that the scheduling policy does not necessarily affect interrupt processing, so a process may still suffer the overhead of interrupts caused by some other process doing I/O; this is an unavoidable overhead of multiuser activity on some architectures.

4.6.4. Temporary Non-Preemption

Group- and non-preemptable scheduling policies are simple mechanisms for users who need long-term control over preemption; as such they are expensive because of the long-term commitments required. There are other situations, in which synchronization occurs for short periods of time, that can be satisfactorily handled by a cheaper mechanism; we introduce the notion of *temporary non-preemption* for this purpose. This is a "hook" into the scheduler to allow processes to accommodate very short-term conditions. An example of typical usage is to prevent preemption while holding a short-duration busy-waiting lock. The process essentially says, "I don't mind being preempted, but please wait a bit until I say it's ok". The mechanism also provides protection against malicious processes that *never* say "ok". Unlike the group- and non-preemptable scheduling policies, it is also useful on uniprocessors as well as multiprocessors, and doesn't require interaction with the long-term scheduler. Two calls are provided, `tempnopreempt` and `tempokpreempt`. The first notifies the scheduler that temporary abeyance of preemption is desired, and the second signals a return to normal scheduling.

Much of the usefulness of this feature derives from the fact that the implementation is extremely efficient. The scheduler maintains for each process the address of a two word vector in the user address space; this address is established via a new system call normally issued by the language start-up code immediately after `exec`. The first word is for communication from the user to the scheduler, the second is for communication from the scheduler to the user. Initially, both words are set to zero. What `tempnopreempt` does is essentially

```
word1++;
```

while `tempokpreempt` is approximately

```
// ordinary version
if (--word1 == 0 && (temp=word2) != 0) {
        word2 = 0;
        yield(0);
        return temp;
}
return 0;
```

The `yield` system call does nothing if there is no preemption pending, otherwise it reschedules the processor, temporarily adjusting the external priority[86] of the calling process by addition of the supplied argument. When the process next runs, the external priority is restored.

[86] In traditional UNIX terminology, the external priority is the *nice* value.

In this case, an argument of 0 causes no external priority change, just as with a normal preemption. By specifying `yield` to do nothing when no preemption is pending, we close the window of vulnerability between decrementing *word1* and calling `yield`: if the process is preempted during this interval, the `yield` is unnecessary and will have no effect.

Because *word1* is incremented and decremented, rather than simply set, `tempnopreempt` and `tempokpreempt` can be safely nested. (An alternate version of `tempokpreempt` will be presented in section 4.6.5 on the next page.)

What the scheduler does is complementary. When it wants to preempt a process, code such as the following is executed on the processor running the process (i.e., the scheduler runs this as part of the `softresched` interrupt handler):

```
if (word1 == 0)
        ok to preempt;
else if (preemption not already pending for this process) {
        word2 = 1;
        note preemption pending;
}
else if (preemption pending for at least tlim time) {
        word2 = 2;
        ok to force preemption;
}
```

Where *tlim* is perhaps a few milliseconds, and the *preemption pending* state is maintained per-process and cleared on actual preemptions and on `yield`. The purpose of the `tempokpreempt` return value is to notify the user (after the fact) if preemption was requested or forced.[87]

Overhead for this implementation is only a few instructions in the common case where no preemption would have occurred anyway, and only the overhead of the `yield` otherwise. The most abusive a user can get with this scheme is to lengthen a time-slice by *tlim*; it is easy to prevent any net gain from such a strategy by shortening the next slice by *tlim* to compensate.

4.6.5. The `SIGPREEMPT` Signal

We introduce a new signal type, `SIGPREEMPT`, specifically for user-mode thread systems, but any preemptable application can use it to keep track of scheduling assignments over time. The basic idea is to notify the user process before preempting it, and allow a short grace period for it to put its affairs in order. On receipt of `SIGPREEMPT`, a user-mode thread system might save the state of the currently executing thread before giving up its processor by calling `yield` (§4.6.4); this makes it possible for another cooperating process to run the thread. Section 7.5 will describe more details of how a user-mode thread system can use `SIGPREEMPT` together with asynchronous system calls, asynchronous page faults, temporary non-preemption, and carefully chosen counting semaphores (§7.3.3), in order to retain control of its threads and processors, as much as possible, even in the face of kernel-initiated

[87] Presumably this may be helpful in performance evaluation and tuning.

preemption.

The grace period for SIGPREEMPT need not be the same as the time limit for temporary non-preemption; the choice is up to the system implementor (they could conceivably even be non-constant). As with temporary non-preemption, the grace period must be enforced in such a way that no long-term gain can be obtained by abusing it. SIGPREEMPT is easier than temporary non-preemption in this respect, since the kernel knows explicitly when the grace period begins. If the process fails to block or call yield during the grace period, its scheduling priority can be temporarily reduced or its next scheduling quantum can be reduced to compensate for the full grace period consumed.

There are several cases of preemption to consider:

- *Preemption of primary or other activities.* Only preemption of a primary activity can generate SIGPREEMPT. Other activities are not under direct user control, since they are executing "in the background"; there is nothing the SIGPREEMPT handler can do about them. Although the activity is the unit of scheduling (and hence preemption) within the kernel, activities are largely invisible to users. At the user level, processes are the relevant unit, and a process is represented by its primary activity.
- *Masking of SIGPREEMPT.* It is tempting to use the signal masking facility with SIGPREEMPT instead of the temporary non-preemption mechanism presented in section 4.6.4. Such an approach would require special case handling compared to other signals and would lose the functionality of the tempnopreempt return value. Furthermore, temporary non-preemption can be useful even without SIGPREEMPT and, for purposes of presentation, it is helpful to separate the two mechanisms. Using SIGPREEMPT masking instead of temporary non-preemption is likely to add complexity and overhead in the common case of no preemption. In terms of basic technology, temporary non-preemption is applicable to any multiprogrammed operating system, while SIGPREEMPT has UNIX signal semantics built-in (although presumably something similar could be done for systems with some other form of user interrupt mechanism).

  Our choice is to preempt silently (without a signal) if SIGPREEMPT is masked when preemption is required (i.e., the same as SIG_DFL and SIG_IGN). If temporary non-preemption is in effect, the signal is discarded and preemption is deferred; this reflects the assumption that the user will call yield soon; if not, silent preemption is appropriate.
- *Using SIGPREEMPT with tempnopreempt and tempokpreempt.* The version of tempokpreempt presented in section 4.6.4 calls yield when the kernel desires preemption; this is inadequate if the user is trying to intervene in all preemptions by catching SIGPREEMPT. The following version is more appropriate in that case:

```
// version for users with SIGPREEMPT handlers
temp = 0;
while (--word1 == 0 && (temp=word2) != 0) {
        word2 = 0;
        word1++;
        user_sigpreempt_handler(SIGPREEMPT);
}
return temp;
```

  This version re-enables temporary non-preemption and calls the SIGPREEMPT handler to perform the desired actions and yield. If a natural preemption occurs between decrementing *word1* to zero and re-incrementing it, the call to `yield` from within the user's `SIGPREEMPT` handler will do nothing (because no preemption would be pending by then). Of course the temporary non-preemption time limit must be long enough to allow the execution of the handler in addition to whatever code originally required temporary non-preemption.

- *Preemption from user-mode or kernel-mode.* If the `softresched` interrupt comes from user-mode, the activity is already in a state ready to handle a signal. kernel-mode preemption must be further divided into two separate cases: is the activity potentially asynchronous (§4.5.2/p178)? If so, receipt of a signal creates a new primary activity to return to user-mode and handle it, leaving the old activity to finish its system call or page fault in truly asynchronous fashion. The old activity, no longer primary, is preempted silently.

  Otherwise, the activity must be executing a synchronous system call or page fault, or must be an internal kernel support activity; all such synchronous activities are preempted silently. This represents a loss to a user-mode thread system only when executing a long but fundamentally synchronous system call such as `exec`, or incurring a synchronous page fault; the former is unusual for a thread system, and the latter only happens in rare but unavoidable situations, as described in section 7.5.2 on page 320. The kernel masks preemption during short synchronous system calls that never block, such as `getpid` or `gettimeofday`.

The use of preemption (in any form) adds overhead to any system and to the preempted applications, compared to a fully non-preemptive system. On the other hand, the use of preemption can improve average response time, increase the overlap of I/O with computation, and make the system more responsive to external process priorities; these are reasons why the overhead of preemption is considered acceptable. The use of `SIGPREEMPT` increases the overhead somewhat, compared to traditional (silent) preemption, because of signal delivery and user-level actions taken by the handler. We claim these additional overheads are justified by the benefits of `SIGPREEMPT`, i.e., the efficiency gains of implementing threads in user-mode. In either case (with or without `SIGPREEMPT`), the "ratio" of overhead to benefits is greatest when a preemption is begun, or `SIGPREEMPT` is sent, and another processor becomes idle immediately. This is a fundamental problem of on-line scheduling: future knowledge is not available. Use of `SIGPREEMPT` makes this somewhat more likely, because a full preemption (including the user-level actions of the `SIGPREEMPT` handler) takes longer than a silent preemption. Keeping the grace period short tends to lower the vulnerability.

When should `SIGPREEMPT` be delivered? If a primary activity is preempted in favor of an activity with equal priority, as in round-robin scheduling, it seems fair to delay the beneficiary, while the victim handles `SIGPREEMPT`. This is the most efficient way to deliver `SIGPREEMPT`.

When the priorities are different, we must be concerned about two problems: *priority inversion* and *interrupt-to-wakeup latency*. Their severity depends on two factors: our level of commitment to real-time requirements and the actual costs involved (e.g., the maximum grace period length).

If we delay the higher priority activity while the lower handles `SIGPREEMPT`, we have created a priority inversion. Any real system has priority inversions, such as when interrupts are blocked (for however short a time), when a low priority process saves state before allowing a higher priority process to run, or when a high priority process needs a resource for which a low priority process holds the lock. The important thing is to keep inversions short, so their impact is minimal.

Interrupt-to-wakeup latency is the time from when an external interrupt is raised to when a blocked activity is unblocked by the interrupt handler and actually begins running on a processor. This latency is often considered an important component of overall system responsiveness and a key requirement for real-time systems.

The expected time to execute a `SIGPREEMPT` handler should not be very large (all it is really expected to do is save thread state), but the full grace period may be uncomfortably long, especially for real-time systems. (If the possibility of buggy or malicious programs is ignored, the expected time may be all that matters.)

There is an alternative to waiting while a victim handles `SIGPREEMPT` in user-mode: the victim can be preempted silently, and `SIGPREEMPT` can be delivered when it runs again, on another processor. To make this an effective solution, it must run again very soon, otherwise the purpose of `SIGPREEMPT` would be defeated. Instead of just waiting for it to run according to the normal scheduler mechanisms (§4.6.6), we can take special action to preempt another activity silently, just long enough to run the `SIGPREEMPT` handler to the point of `yield`. This *secondary preemption* victim, which must be running on another processor and should be of equal priority, is an "innocent bystander" and suffers, without notification, a delay of up to one grace period (plus possible cache effects). The secondary victim's delay is comparable to the delay of arbitrary activities by unrelated interrupts on their processors and to memory contention between different processors (although the time scale is much different, the cumulative affect over time may be similar).

Assuming the APC mechanism (§3.6) uses a hardware interrupt, the secondary preemption itself will not take long, but choosing the secondary victim quickly and without serial bottlenecks poses an algorithmic and data structure challenge. The best way to perform secondary preemption depends on the general organization of the short-term scheduler, discussed in section 4.6.6.

4.6.6. Scheduler Design and Policy Alternatives

For flexibility, and especially because this is an experimental system, the scheduler is isolated enough so that it may be readily replaced. The concept of separating policy decisions from mechanisms is well known, but for practical purposes we find it advantageous to broaden our notion of "policy" in this case. By declaring the design of some major structures, such as the ready list, to be policy issues, we increase flexibility while maintaining the potential for high performance.

The scheduler consists of six visible functions:

`void mkready(struct activity *)`
: Change activity state to `ACT_READY`, if it wasn't already, and arrange for it to run. In the most common case, `mkready` is called when a blocked activity is unblocked. There are other cases where the calling activity already has state `ACT_READY`, such as `yield` (§4.6.4).

`void lmkready(struct activity *)`
: Like `mkready`, but the activity must already be locked.

`void mkqready(mqitem *mqi, int pri, int n)`
: Make `n` activities ready with priority `pri`. The activities are obtained by calling the function specified when `mqi` was initialized (§4.5.2/p173). There is a status anomaly

associated with `mkqready`: the awakened activities are not handled individually until they are selected for running; although they are on the ready list, their `a_status` fields still show `ACT_BLOCKED`.

`struct activity *resched(int isready)`

Pick another activity to run instead of the current one. The parameter is a boolean value that indicates if the current activity can continue running or not; this would be equivalent to checking the `ACT_RUNNING` flag of the current activity, but such a check would sometimes fail, due to a race with `mkready`. If `isready` is true, `resched` may return `NULL` if there is no other activity that should run instead of the current one, otherwise the caller is expected to pass the return value to `cswitch` or `cjump` (§4.4.5). If `isready` is false, `resched` will never return `NULL`, since each processor has a dedicated *idle activity* that is always ready to run.

The call `resched(0)` is used when an activity is blocking; `resched(1)` is used to attempt a preemption. All preemption attempts are performed by the `softresched` interrupt handler, and may be requested at any time by calling `setsoftresched` (§4.6.1, §3.4/p45).

`void setpri(struct activity *)`

There are only two ways to set the priority of an activity: by calling one of the context-switching synchronization functions that can block an activity (such as `icsrw_rlock` (§4.5.2/p174))[88] or by calling `setpri`. The conditions for calling `setpri` are:

- After changing the external priority (traditionally called the *nice* value), as part of an explicit priority changing system call, i.e., `nice` or `setpriority`.
- Before returning to user-mode from kernel-mode. This ensures that temporary priority changes made by context-switching inside the kernel are not propagated to user-mode.
- When an activity has been running for longer than its *run quantum*, `setpri` is called to recalculate its priority, presumably downwards.
- The scheduler implementation may assign a *wait quantum* to each activity that is ready but not running, to place a limit on the amount of time an activity can languish before having its priority reconsidered. In this case, `setpri` is called to recalculate its priority, presumably upwards.

`void newrquantum(struct activity *)`

When an activity is run (by `cswitch` or `cjump` (§4.4.5)), `newrquantum` is called to compute and establish a *run quantum*, the maximum time before preemption will be attempted. When the run quantum expires, `setpri` and `resched(1)` are automatically called, and `cswitch` is called if `resched` returned a non-`NULL` value.

By replacing these six functions and changing their data structures we can provide very different schedulers.

---

[88] Recall that priorities are not preserved when blocking (§4.5.2/p171).

4.6.7. The Ready List

The key data structure of the scheduler is the *ready list*. There are many possible approaches to its design, depending on the overall system requirements (e.g., strength of real-time goals), anticipated workload, and architectural details (e.g., machine size and hardware support for Fetch&Φ and combining).

One common approach is to implement N distinct priority levels by N ordinary lists (a highly parallel variety (§3.7.1) should be used for our target class of machines (§1.3)). A ready activity goes on the list associated with its priority. A processor looking for work simply attempts to delete from each list, from the highest to the lowest priority to be considered, until an activity is found. An idle processor may check all priorities, but a non-idle processor is checking to see if preemption is appropriate, so it doesn't consider priorities below that of the currently running activity and only checks the equal priority list when the running activity's time quantum has expired. There are many possible enhancements to this basic scheme, including:

- *Fast scanning*. Schedulers for uniprocessors and small multiprocessors often speed up the search for the highest priority runnable process by maintaining a bitmap to show which lists are non-empty at any given moment. By consulting the bitmap, a processor looking for work can immediately determine the highest priority non-empty list. Unfortunately, manipulation of such a bitmap would cause a serial bottleneck on machines outside of our target class. Bottleneck-free parallel algorithms exist to find the highest priority non-empty list of N lists in $O(\log N)$ time, but they are rather complicated (Gottlieb, *et al.* [93]).
- *Local ready lists.* Additional lists can be added on a per-processor basis so that, for example, an activity needing access to hardware associated with a single processor can be limited to run only there. This feature was used in some versions of Mach to limit execution of most system calls to a single "master" processor (the local scheduling lists are general enough to use for other purposes as well). Of course the changes associated with adopting local ready lists can't be confined to the scheduler itself, since other parts of the kernel must also be changed to take advantage of it.
- *Use of multiqueues.* Symunix-2 includes some operations, notably the `spawn` system call (§4.2/p148) and certain context-switching wakeups (§4.5.2/p179), that involve putting many activities on the ready list at the same time. An appropriate implementation of `mkqready` (§4.6.9) can take advantage of multiqueues in the ready list to execute these operations very quickly. Unfortunately, multiqueue algorithms tend to be more expensive than their singleton cousins (see Table 13, in section 3.7.1 on page 82). One solution is to represent each priority with *two* lists, one multiqueue and one singleton queue. The order in which they are checked when looking for work determines a sub-priority relationship between them.

Regardless of any such variations, these kinds of ready lists are rather weak in their ability to match activities and processors. For example, consider:

- *Cache Affinity.* The idea that execution threads exhibit *cache affinity* suggests that extra effort should be made to schedule an activity on the last processor to run it whenever possible. The list-per-priority approach to ready list design is not well suited to supporting cache affinity without adding per-processor ready lists, and that comes at the cost of load balancing.

- *Cascade Scheduling.* In a "pure" list-per-priority scheduler, a newly ready activity goes on the shared ready list, to be picked up by the first processor to find it. But the processor to find it may not be the "best" choice. Ignoring for a moment the issue of cache affinity, the best choice is the processor running the lowest priority activity (the per-processor idle activity being the lowest of the low), but there is no mechanism to ensure this choice. As a result, we observe a *cascade scheduling* effect, where the first victim preempts a lower priority activity, which preempts an even lower one, etc. It is easy to fix this problem when some processors are idle: keep a list of idle processors, and schedule a ready activity directly to one if available, without going through the ready list itself; the Mach system uses this approach. Avoiding cascade scheduling in the absence of idle processors is harder (but see section 4.9 on page 207).

4.6.8. The `resched` Function

The purpose of the `resched` function is to select another activity to run. In addition to manipulating the ready list, it considers the priority of the currently running activity and participates in preemption.

As indicated by the usage in `_icsrw_rlock` (§4.5.2/p177) and `_icsrw_wlock` (§4.5.2/p180), the `resched` function returns a pointer to the activity chosen to run, assuming the caller will call `cswitch` if the chosen activity is non-`NULL`. This not only allows for *hand-off* scheduling by simply avoiding the call to `resched` altogether, but also solves a machine-dependent problem of relocating the interrupt stack when performing preemption, as discussed in section 4.6.1 on page 187.

The `resched` function takes a parameter, `isready`, to indicate whether or not the calling activity is logically able to continue execution. This parameter is true only when `resched` is called from preemption (§4.6.1) and the `yield` system call (§4.6.4). In these cases, another activity is selected to run only if it has "high enough" priority.

```
struct activity *
resched (int isready)
{
      int pri;
      struct activity *oact, // old activity
                      *nact; // new activity

      oact = curact();
      if (!isready)
            pri = PRI_WORST;
      else {
            pri = oact->pri;
            if (oact->quantum > 0 && ++pri > PRI_BEST)
                  pri = PRI_BEST;
      }
```

The variable `pri` is the "good enough" priority value we seek. `PRI_WORST` is the worst priority in the system, and is reserved especially for the idle activities.

```
        nact = rqget (pri);
        if (nact == NULL) {
             if (isready) {
                  if (oact->quantum <= 0)
                       newrquantum (oact);
                  return NULL;
             }
             panic ("resched: no idle act");  // can't happen
        }

        if (isready) {
             spl_t s = splsoft();
             bwl_wait (&oact->lock, splcheckf(s));
             rqput (oact);
             bwl_signal (&oact->lock);
             vsplx(s);
        }

        md_resched(oact,nact);           // machine-dependent portion
        if (nact->quantum <= 0)
             newrquantum (nact);
        return nact;
   }
```

4.6.9. The **mkready** Functions

As shown by the usage in context-switching readers/writers routines (§4.5.2), `mkready` and the related functions `lmkready` and `mkqready` are responsible for moving an activity into the `ACT_READY` state. The three functions are designed to fit different situations efficiently:

`mkready`	*Make one unlocked activity ready*
`lmkready`	*Make one locked activity ready*
`mkqready`	*Make a whole list of activities ready*

The basic `mkready` function is quite simple:

- Mask interrupts and lock the activity.
- Adjust the system's *instantaneous load average*, the number of currently runnable activities, used for informational purposes.
- Change the activity's state to `ACT_READY`.
- Call `rqput` to insert the activity onto the ready list.
- Unlock the activity and unmask interrupts.

In the case of `lmkready`, the locking and interrupt masking are unnecessary, because they are assumed to have been done by the caller.

The `mkqready` variant is even simpler: it simply adjusts the instantaneous load average and calls `rqqput` to insert the whole list of activities onto the ready list at once. The remainder of the work is performed by synchronization-specific routines such as _icsrw_rwakeup, (§4.5.2/p182), called as each item is deleted from the ready list. Activity lock and interrupt mask manipulations are required only in the synchronization-specific routines, as only they deal with individual activities.

4.7. Process Creation and Destruction

A reference count, p_rcnt, is maintained for each process, primarily to control deallocation of the most fundamental resources of a terminated process: the proc structure, the process ID, and the process group ID. In traditional UNIX, final deallocation of a process is done when a parent makes a wait system call to collect the exit status of a terminated child. There are several reasons why that strategy is inadequate for a highly parallel operating system kernel:

(1) The traditional wait system call is a direct source of serialization, since a parent must eventually make a separate call to wait for each child. If many processes are to cooperate in solving large problems, this potential bottleneck must be avoided. We have introduced the spawn system call to allow creation of many processes at once without serial bottlenecks (§4.2/p148). One of the arguments to spawn is a set of flags, several of which affect what happens when the spawned children eventually terminate (§4.2/p149). In particular, it can be arranged for the children to take care of their own deallocation in the absence of errors; this allows the parent's wait to complete without doing per-child work in the most common case. In case of errors, it is usually desirable for the parent to investigate them serially.

(2) In UNIX terminology, when a parent process pre-deceases a child, the child is said to be an *orphan*. Traditionally, orphans are inherited by the *init* process, with PID 1. In particular, a wait system call executed by init will return an orphan's exit status, and a getppid (get parent PID) system call returns 1 to an orphan. To avoid the serial overhead of having init handle orphans, we observed that, in practice, no real use was made of the exit status returned to init for orphans. In Symunix, orphans only *appear* to be inherited by init: getppid returns 1 and orphans are counted to determine when wait should report "no more children" to init, but an orphan's resources are returned to the system when it terminates, without involving init. Consequently, process deallocation for orphans is not performed serially, and init does not receive their exit status.

(3) To support interactive debugging of parallel programs, the ptrace mechanism is extended to allow tracing of processes that aren't immediate children. As a result of this extension, debuggers are treated like a second parent, as far as reference counting and resource deallocation on termination are concerned.

The basic idea behind reference counting for a process is to count the number of other processes that have pointers to it: its parent, children, non-parent debugger, and, if it is a debugger, each non-child it traces. A zombie process is destroyed when the reference count drops to zero.

When a process with children terminates, each child is individually orphaned. This is not considered a serious serial bottleneck, because a parent terminating before its children is not considered normal behavior for large parallel programs.[89] The key steps in orphaning a child are to destroy its reference to the parent, and decrement both reference counts. Termination of a debugger with debuggees is handled similarly.

In addition to determining when the final resources of a process are deallocated, the reference count is also used in the implementation of the `wait` system call, to decide if there are live children to wait for.

Since init never terminates, a special case is applied to handling its reference count: it includes all processes in the system except init itself, and never-terminating internal system daemons, such as the idle process (which has one activity per processor). When any process terminates, init's reference count is decremented. Init is awakened when the count reaches zero if it's blocked in the `wait` system call. This allows init to wait for termination of all processes during a system shutdown, even though orphans aren't really adopted by init. In addition, init's reference count is a convenient place for instrumentation programs to look for the current number of processes.

4.8. Signals

We have already seen that signals play a central role in management of asynchronous system calls (`SIGSCALL` in section 4.3.1 on page 151), page faults (`SIGPAGE` in section 4.3.2), and scheduling control (`SIGPREEMPT` in section 4.6.5). In addition, standard UNIX semantics require signal support for various (mostly exceptional) purposes. While some of these traditional uses are not frequent they still present an implementation challenge for highly parallel systems.

4.8.1. Signal Masking

The performance impact of signal masking is considerably greater in a highly parallel environment than otherwise because of the need to mask signals in many locking situations. We can implement the standard Berkeley UNIX signal masking functions without system calls, in a manner analogous to the implementation of temporary non-preemption described in section 4.6.4. In this case, the address of a single word (*sigword*) and two bitstrings in the user address space are maintained by the kernel (*bitstring1* and *bitstring2*). The user can set *sigword* to a non-zero value to mask all maskable signals, and *bitstring1* to mask particular signals with individual bits. (We could eliminate *sigword* entirely if the bitstrings could be manipulated atomically, or if we chose not to require atomic signal mask changes.) The kernel examines *sigword* and *bitstring1*, and modifies *bitstring2* to notify the user of pending masked signals. The implementation of `sigsetmask` is essentially

---

[89] Perhaps we have simply defined the problem away. The *family* and *progenitor* concepts help to eliminate the need for orphaning children in parallel programs (§4.2).

There is no conflict between our claim here that creating orphans is abnormal and our claim in this section that handling of orphans might cause a serial bottleneck for init, since they apply to different contexts (a single parallel program and the whole multi-user system).

```
sigword++;
oldmask = bitstring1;
bitstring1 = newmask;
sigword--;
if (bitstring2 & ~bitstring1)
        sigcheck();
return oldmask;
```

where *newmask* is the argument and `sigcheck` is a new system call that causes all non-masked pending signals to be handled. The implementation of `sigblock` is analogous. In this pseudo-code, we are treating the bitstrings as if they were simple scalars, although this is not possible in standard C if they are array or structure types.

Whenever a signal is sent to a process, the kernel examines *sigword* and *bitstring1* to see if it is masked or not, and it adjusts *bitstring2* if necessary whenever a signal is sent or handled:

```
if (sigword != 0 || (bitstring1 & sigbit) != 0) {
        // signal is masked
        bitstring2 |= sigbit;
}
else { // signal is not masked
        if (signal is to be caught) {
                bitstring2 &= ~sigbit;
                arrange to deliver signal
                  to user's handler on return;
        }
        else
                take appropriate default action;
}
```

Here *sigbit* is a bitstring with only a single bit set, corresponding to the signal being sent. Note that the pseudo-code is written as if the user variables (*sigword* and the two bitstrings) were known and directly accessible; this isn't true. The kernel knows these locations only by addresses supplied through the `setkcomm` system call (§4.4.1/p159), usually called by the user-mode start-up code immediately after `exec`. If the user-supplied addresses are invalid, the real code behaves as if *sigword* and *bitstring1* are both zero.

To understand the kernel part of this signal masking and delivering code, we need to know the context in which it executes. Because signals are essentially interrupts in user-mode and our process model is the traditional UNIX single-threaded one, the primary activity handles signals in the kernel immediately before "returning" to user-mode. Thus, it is correct to think of the kernel's code accessing *sigword* and *bitstring1* as a subroutine or an interrupt handler, not as a separate process or thread executing concurrently with the user.

We wrote the pseudo-code as if a single known signal is being sent, but in reality the kernel keeps a bitstring of pending signals and the real version of this code looks for an unmasked one from among those pending. Some mechanism must also be provided for the kernel to gain control to deliver a signal sent after this code has finished and possibly returned to user-mode. Traditional UNIX kernels sometimes rely on periodic clock interrupts to cause rechecking for newly arrived signals; Symunix-2 minimizes signal latency by using interprocessor interrupts (APCs; §3.6) together with careful interrupt masking in the code before returning to user-mode.

4.8.2. Sending Signals to Process Groups

Traditional UNIX systems support the concept of *process group*, which represents either all processes of a login session or all processes of a “job”, depending on whether “job control” is in effect. The traditional implementation for sending a signal to a group relies on a serial scan of all processes, during which time process creation, termination, and other changes are prohibited.

Sending a signal to a group is generally an indication of something abnormal, and therefore might be considered unimportant as a candidate for parallelization. But consider:

- Signals may be sent to process groups by the kernel, such as when certain special characters are typed on a terminal (e.g., `SIGINT` is often sent by typing control-C). In traditional implementations for directly connected terminals, this signal processing is done by an interrupt handler within the kernel, with some class of interrupts masked, thus jeopardizing interrupt latency on machines with many processes.
- Signals may be sent to an entire process group by explicit system calls, such as `kill` and `killpg`.
- In Symunix-2, parent and child process relationships must be maintained such that when a process with children terminates, the children are sent the `SIGPARENT` signal.
- The maximum number of processes in a group is likely to grow linearly with the maximum number of processes a system supports. (Linear growth is probably valid for scientific workloads, but possibly not for others.)
- Process group membership changes dynamically, e.g., through process creation and termination and the `setpgrp` system call. Sending a signal to a group should be atomic with respect to group membership changes, otherwise it might be impossible to kill a group (e.g., with an infinite `fork` loop). (Some traditional UNIX implementations fail in this case, as does Symunix-1: it doesn’t use the mechanisms described in this section.)
- Even if signals are seldom sent to groups, the group relationships must still be maintained in a bottleneck-free fashion, because they change much more often and even with great synchrony. (Consider the case of many sibling processes being created or terminating at about the same time.)

Section 3.7.3 and section 3.7.4 showed how *visit lists* and *broadcast trees* can be built to help solve these problems. Broadcast trees help provide the correct (atomic) semantics by defining which processes should receive a group signal, and visit lists are used to implement signal polling by group members. We now discuss how these data structures are used to implement signals efficiently for machines in our target class.

Besides the traditional process group, Symunix-2 supports several additional process aggregates, as listed in Table 26, on the next page. Each aggregate has broadcast tree,[90] `bct_sync`, and `vislist` structures, and each process has `bct_memb` and `visitem` structures. In some cases, this requires the introduction of a structure, not present in traditional UNIX implementations, to represent the aggregate itself. For example, a new structure type,

---

[90] Recall from section 3.7.4 that broadcast trees are declared with a macro, `BCT_DECL(`*n*`)`, where *n* is the range of items to be broadcast, in this case `NSIG`, the number of signal types.

Aggregate	*Purpose*
All	All non-system processes
Login	Processes in same login session
Job	Traditional UNIX process group, used for "job control"
Family	Processes related by `fork` and `spawn`, but not `exec`
Siblings	Children of same parent or progenitor
UID	User ID

Table 26: Process Aggregates.

Of these, only "all" and either "login" or "job" are supported by traditional UNIX systems. Each process has two user ID group memberships, one for the *real* user ID and one for the *effective* user ID. A signal sent by a non-root process to UID u will go to all processes with either kind of user ID = u. (This is the same as Table 19, in section 4.4.1 on page 159.)

`active_uid`, contains the broadcast tree, `bct_sync`, and `vislist` structures for each user ID active in the system; `active_uid` structures are allocated and deallocated dynamically according to reference counts, and a searching mechanism is provided along the lines of section 3.7.6. There are two additional factors to complicate this basic scheme, however: user ID checking and changing group membership, which we discuss in turn.

User ID Checking

When the kernel sends a signal to a traditional aggregate, e.g., because the user typed control-C, the signal goes to all processes in the group. But when a process uses the `kill` or `killpg` system call,[91] only those group members with a matching user ID[92] receive the signal, unless the sender's effective user ID is 0, i.e., the super-user, in which case all group members receive the signal.

The user ID matching requirement only affects the traditional process group ("job" in Table 26), but significantly complicates the implementation. The other traditional aggregate, "all" in Table 26, would be affected too, but we have introduced the "UID" aggregate to use instead of sending unprivileged signals to "all". A process with different effective and real user IDs is associated with two different `active_uid` structures.

We augment the traditional process group's data structures with an additional broadcast tree for each user ID represented among its members,[93] arranging them in a simple

[91] The `kill` system call sends a signal to a single process when a positive ID is provided, and to a process group otherwise; `killpg` only sends signals to process groups.

[92] The notion of user ID matching varies between different versions of UNIX. The POSIX standard accepts any of the four possible matches [113].

[93] Multiple user IDs can be represented in a process group by execution of programs with the set-user-id bit asserted, by use of the `setpgrp` system call, and by various flavors of the `setuid` system call.

linked list protected by a readers/writers lock (§3.5.4). Note this is not the kind of list described in section 3.7.1; it is a simple ad-hoc list used for searching (as in section 3.7.6, but without the hash tables). The first broadcast tree on the list is special: it references all members, regardless of user ID, and is used for signals sent by the kernel itself or by processes with an effective user ID of 0. The other broadcast trees are used by non-privileged processes.

We give each process three `bct_memb` structures for its process group:

- One for the broadcast tree assigned to the member's real user ID,
- One for the broadcast tree assigned to the member's effective user ID, and
- One for the all-members broadcast tree.

To preserve UNIX signal semantics, we need to ensure that a single signal sent to a process group isn't received more than once on account of the recipient having two distinct user IDs. We do this by coordinating actions on 1, 2, or 3 broadcast trees, using `bct_put1`, `bct_put2`, `bct_get2`, and `bct_get3` (§3.7.4/p110):

- *Sending a signal to a process group.* If the sender's effective user ID is 0, the sender's real user ID is 0, or the sender is the kernel itself, the signal is posted to the all-members broadcast tree. Otherwise, it is posted to the one or two broadcast trees for the sender's user IDs. Posting is performed by calling `bct_put1` or `bct_put2`. It isn't necessary to use `bct_put3` because a process can have at most two user IDs.
- *Receiving a signal as a process group member.* One, two, or three broadcast trees must be checked: the one for all-members and the one or two for the recipient's user IDs, if they are non-zero. Checking is performed by calling `bct_get1`, `bct_get2`, or `bct_get3`.

By using the broadcast tree operations that coordinate actions on more than one tree, we guarantee that if a single signal is both sent and received on two separate trees, it will only be counted once. (Recall that traditional UNIX signal semantics do not involve counting or queuing signals.[94])

Although we must implement separate broadcast trees for each user ID/process group combination actually in use, the visit lists need not be so duplicated; one per process group is sufficient. The function executed through the visit list for each process group member checks the correct broadcast trees, and no significant action is performed unless signals are received. The potential inefficiency of always visiting all group members doesn't seem serious because most process groups are expected to have only one user ID represented, and because we assume signal sending is much less common than group membership changes.

Changing Group Membership

Most process aggregates in Table 26 are only subject to changes by process creation and termination; the exceptions are the "job" and "UID" aggregates. A process can change from one process group to another via the `setpgrp` system call, and can change user IDs with system calls such as `setreuid` or by executing a file with the *set user ID* file attribute bit set. Traditionally, there were no actual kernel resources dedicated to these aggregates, so the

---

[94] Some signals, e.g., `SIGCHLD` and `SIGSCALL`, may appear to be queued although the implementation doesn't usually work by queuing the signal. But note that these signals aren't sent to groups.

implementation simply recorded the appropriate IDs with each process. With the introduction of explicit data structures for process aggregates in Symunix, it is possible for these system calls to fail due to resource depletion. The addition of new (albeit unlikely) failures is unfortunate, but these system calls already have other failure modes so robust programs should be prepared for a failure indication. Furthermore, these system calls are used by relatively few programs other than shells and login programs.

Not only is it possible for a change in aggregate membership to fail, but a naive implementation could cause the process to be left with membership in neither the old nor the new aggregate: consider the consequences of resigning from the old one before joining the new one. We adopt a simple and safe method, which is to join the new aggregate before resigning from the old one. This is accomplished by using a spare aggregate membership structure (reserved at system boot time by each processor for this purpose); the spare and the old one are swapped after a successful join and resign.

4.9. Future Work

Considerable work remains to evaluate the merits of the design we have presented in this chapter. In this section, we briefly describe some additional ideas worthy of further exploration.

Temporary Interrupt Masking

The temporary non-preemption mechanism described in section 4.6.4 is effective at reducing the large and unpredictable delays incurred when a user process is preempted at the wrong time. Unfortunately, most architectures impose other unpredictable delays as well, including memory contention and hardware interrupt handling. The only real solution to either of these problems is architecture-dependent, but generally based on optimal assignment of data to memory modules and instruction streams to processors. Such solutions are well beyond the scope of this dissertation.

The idea of allowing an unprivileged user process to disable preemption for a short period of time suggests an analogous approach to dealing with interrupts. A new hardware facility could be designed to allow unprivileged code to mask hardware interrupts temporarily. The simplest measure of time is instruction execution cycles. There are many possible designs along these lines, including some that have already been implemented.

A very general design along these lines (but not necessarily the best to implement) would include two new special processor registers:

- A down counter, giving the number of instruction execution cycles remaining before interrupts are unmasked.[95] A value of 0 would disable the counter and unmask interrupts. This register would be user readable (and writable when zero).
- A maximum count register, to limit the maximum value of the down counter. An attempt to set the down counter to a value greater than the maximum count register would fail in some appropriate way. The maximum count register would be user readable but only

[95] By specifying "instruction execution cycles" instead of "instructions" allows for handling long-executing instructions, such as vector operations or the classic CISC instructions, such as string operations, CRC computation, polynomial evaluation, etc.

kernel writable.

This approach would let the operating system decide how much the user may add to interrupt latency.

The biggest problem with any temporary interrupt masking facility is the handling of exceptions, especially TLB misses and address translation failures (which are the same on machines with software TLB reload). Not only do these exceptions usually take more cycles than should probably be allowed for temporary interrupt masking, but they may even allow interrupts (e.g., a context-switch to a new thread while handling a page fault should unmask interrupts). Furthermore, it isn't possible to specify a portable method of preventing such exceptions, except to avoid referencing memory, because of some limiting TLB organizations. For example, some machines have only 2-way set associative TLBs; some sequences of more than 2 memory references are impossible on such machines without TLB misses.

Despite these problems, the basic concept of temporary interrupt masking is attractive, for example, to limit more strictly the execution time of small critical sections. Given a machine-dependent definition of what code sequences can be executed without TLB misses, how valuable would such a facility be? How strongly does the usability of such a facility depend on the TLB organization (e.g., size and associativity)?

The processors in the Ultra-3 prototype include a Field Programmable Gate Array (FPGA) which can be used to implement a facility along the lines we have just described (see Bianchini, *et al.* [29]). Promising results in such an evaluation might help justify inclusion of similar facilities in more commercial microprocessors.

Limiting Execution Time in User-Mode

As described in section 4.6.4, the temporary non-preemption mechanism depends on an implementation-defined time limit for delaying preemption. Similarly, the `SIGPREEMPT` signal promises a grace period before a user process is preempted (§4.6.5). These time limits must usually be implemented in terms of programmable or periodic clock interrupts. Memory contention, exceptions, TLB miss handling, and interrupts all contribute to the expected execution time of a given code sequence, motivating the choice of relatively long time limits.

An alternative approach is to specify the time limits in terms of user-mode instruction execution cycles. Instead of setting a programmable clock interrupt, or waiting for another regular clock "tick", the kernel could set a special register that counts down by 1 for every user-mode instruction execution cycle completed and generates an interrupt when it reaches 0. This would enable the time limits to be defined in terms the user-level programmer can easily understand and measure, because it is independent of memory contention, exceptions, TLB misses, and interrupts.

This kind of hardware facility could be incorporated into most commercial microprocessors. The Ultra-3 processor's FPGA can also be used to implement a facility of this type for evaluation purposes.

Retaining Priority When Blocking

In the tradition of most previous UNIX systems, we use fixed priorities when requesting or holding context-switching locks. A real-time flavor could be given to the system by replacing the wait lists with versions that maintain priority order, and implementing a method for handling priority inversions. When an activity blocks with priority better than the lock holder, the lock holder's priority should be temporarily raised to match, until the lock is released. For binary semaphores, this is not particularly difficult, but those mechanisms that provide greater parallelism, such as readers/writers locks or even counting semaphores, must avoid solutions that impose serialization or cost proportional to the number of lock holders.

Temporary Preemption

Sometimes an activity needs to run for only a very short time before blocking again. This typically happens when a system call performs many small I/O operations with a small amount of computing between each, e.g., when searching directories on a file's path. Such an activity may need to preempt another when it wakes up after each I/O is complete. Preemption may involve delivering the `SIGPREEMPT` signal, with attendant overhead that could exceed the running time of the I/O-bound activity. Cascade scheduling, described in section 4.6.7, is another problem caused by I/O-bound activities in systems with certain kinds of scheduling.

In these kinds of situations, better performance would result from taking the unusually short running time into account. If activities anticipating short processor requirements could indicate this prediction when they block, we could use an alternate preemption strategy. For example, we could perform preemption silently and temporarily, restoring the preempted activity to the same processor as soon as possible, without using the ready list to reschedule the preemption victim. Some method of dealing with inaccurate predictions must be devised as well.

Priority Scheduling

Cascade scheduling does not always involve short running times. One possible solution to the more general problem of cascade scheduling is based on using a ready list that always maintains a totally ordered ranking, by priority, of all ready or running activities in the system. Within each priority, activities may be ranked by how long they have been ready or running. Such a ready list can be maintained by using a group lock (§3.5.6) to synchronize all processors examining or manipulating the ready list. When an activity becomes ready, it is then immediately determined whether it should run and, if so, which activity to preempt; an asynchronous procedure call (§3.6) can force the preemption.

An algorithm and data structure along these lines has been devised, but not tested. The group lock uses 6 phases in the worst case. The algorithm makes some effort to respect cache affinity, but only within the context of a single group lock execution. It remains to be seen if the algorithm works, or can be "fixed", and how the increased cost of synchronization within the scheduler compares with the benefits of precise priority ranking and cache affinity support thus provided.

Induced System Calls

The meta system call mechanism described in section 4.3.1 on page 151 is well suited for extension in a direction that is both powerful and dangerous. Whereas `syscall` can be used to issue an asynchronous system call for the current process, a new meta system call function, say `induce_syscall`, could be introduced to issue an asynchronous system call to run in the context of another process. Subject to necessary security restraints, such as a requirement for matching real and effective user IDs, this would allow things like:

- After-the-fact I/O redirection.
- Improved control for interactive debuggers.
- A simple method of implementing binary compatibility for another operating system by using a supervisor process that intercepts user traps and exceptions, interprets them, and issues induced system calls to carry out required native system call actions on behalf of a compatible application.

There are a number interface and implementation details to resolve in this area.

4.10. Chapter Summary

There are several key elements presented in this chapter: activities (and their embodiment, asynchronous system calls and page faults), context-switching synchronization routines, scheduling issues, and signals. Each of these came about because of our commitment to low-overhead thread management in user-mode (to be elaborated on in chapter 7) and to a UNIX-like process model. While some of the implementation details are driven by the needs of our target class of machines, each of these key elements has significant value for other shared memory systems, and uniprocessors, as well.

Activities bear a superficial resemblance to kernel threads, but our goals for them are very different and they are visible at the user level only in their effect: they are key to the implementation of asynchronous system calls and page faults. Introduction of asynchronous system calls allows us to provide a clean, uniform framework for solving a number of long-standing problems with UNIX system calls, e.g., interruption by signals. The simple structure and semantics of activities also provides opportunities to streamline the context-switching synchronization code; in turn, the context-switching code provides fundamental support for the implementation of activities, especially automatic creation of new ones.

Parallel programs place unique requirements on the scheduler. Our solution is based primarily on three techniques: scheduling groups, temporary non-preemption, and the `SIGPREEMPT` signal. We argued for the importance of bottleneck-free signal operations, even when signals occur only in very unusual circumstances. We then went on to present highly parallel algorithms to manage groups of processes that may be signal recipients.

Chapter 5: Memory Management

The class of computers we consider has two key concepts at its heart: MIMD control and shared memory. This chapter, *Memory Management*, is concerned not only with the latter but also with the relationship between the two. The most important conceptual feature of the Symunix-2 memory management design is that the memory abstraction is based strictly on the concept of mappings between files and address spaces. No memory is accessible to a program that isn't part of a file in the file system. There is no *anonymous* backing storage for virtual memory; the files themselves serve this purpose.

It is also important to point out some notable features omitted from the Symunix-2 design. The notions of *memory* and *file* are not merged. The decision to design the memory system around file mappings was motivated by the need for simplicity and flexibility of sharing rather than by a desire to hide ordinary file I/O. Contrary to contemporary trends, the kernel doesn't provide shared address spaces; it is possible, however, to support shared address spaces in user-mode (§7.6.2).

After looking at the evolution of operating system support for shared memory in section 5.1, we present the Symunix-2 memory model and its implementation in sections 5.2 and 5.3. Section 5.4 discusses some ideas for future work and we summarize the chapter in section 5.5.

5.1. Evolution of OS Support for Shared Memory

Operating system kernels for shared memory multiprocessors differ in the address space model they provide: some provide a *shared address space* for a collection of execution threads, and some provide only *private address spaces*. Parallel program support environments have their own models for memory and control of parallelism, where "address space" is more often an implementation concept than a semantic one. In practice there are some advantages and disadvantages to both shared and private address spaces, i.e., there are common situations where each is more convenient.

Operating system designers must balance conflicting goals such as performance, portability, flexibility, compatibility with previous systems, and demand for various features. While it is clear that many alternate approaches may be viable, we shall concentrate here on the Symunix-2 operating system. Symunix-2 strives to do well on a specific target class of machines (§1.3), while still maintaining acceptable performance on a somewhat broader class. Flexible support for differing parallel program environments has also been a key goal.

Symunix-2 evolved from traditional uniprocessor UNIX systems, especially 7th Edition and 4.3BSD. As a UNIX system, Symunix-2 must provide all aspects of the traditional UNIX process environment. Chief among them is the association between a process and an address space. In particular, the `fork` system call creates a new process with a nearly exact copy of the parent's address space. This requirement strongly discourages radically different models such as a single system-wide address space for all processes. More moderate models with many shared address spaces have been adopted by most comparable contemporary operating systems, but the Symunix-2 kernel retains the original private address space design. While much of the rationale for this approach is given in section 4.1, there are additional performance issues to be considered in the implementation of shared address spaces:

(1) For the target class of machines, we seek a bottleneck-free implementation, supporting operations such as page fault, page eviction, and mapping changes in a scalable manner. For example, it is possible for every processor of the machine to be performing one such operation concurrently, even within a single application. Designing a kernel implementation of shared address spaces to handle such a situation without substantial serialization is more difficult than designing one for private address spaces, and the resulting system can be expected to suffer greater overhead. The additional overhead provides no corresponding benefit for serial applications, or when the anticipated flood of concurrent operations doesn't materialize in practice. Of course, the cost and difficulty of emulating a shared address space, when needed, on top of a private address space kernel must also be considered.

(2) Private address spaces can be used to emulate shared address spaces. The emulation can potentially be more efficient than a genuine shared address space, because it can be tailored to meet the semantic requirements of different parallel program environments, without paying for unneeded features. For example, many modern virtual memory systems (including Symunix-2) support *sparse* address spaces having page granularity, with potentially many mapped and unmapped regions, each with a variety of distinct attributes, and the ability to change the mappings dynamically. There are good reasons for supporting such a rich model, but many applications require less. A very common usage pattern is to request many shared pages from the kernel at a time, with contiguous increasing addresses, and never change attributes or release the pages until program termination. The allocated pages form a pool managed by user-level code in response to application requests to allocate and free memory. Such an environment can be efficiently created on top of private address spaces without the difficulties or overhead of supporting more complex operations in a bottleneck-free manner. Such an emulation is described in section 7.6.2 on page 323.

(3) *Partially shared* address spaces can be supported, removing the need for an application, library, or run-time support system to emulate private memory in a shared address space. This situation has been encountered over and over by developers attempting to use traditional UNIX libraries in parallel applications for shared memory multiprocessors.[96] The most notorious problems are the global variable `errno` and the standard I/O library. As a result, massive changes to the standard UNIX libraries have been undertaken, including many strange tricks. Some of this effort is

[96]The author experienced this problem when modifying UNIX on a PDP-11/23 (a uniprocessor) to run parallel programs for the NYU Ultracomputer project in 1982 (see section 1.2 on page 2). As a result, the system supported both shared and private data, as did its successors (a master/slave system, and Symunix-1) running on parallel hardware (§1.2/p4).

unnecessary if shared address spaces are implemented at the user rather than kernel level.

Even in programming environments featuring a shared address space, there is often need for some memory that is private to each kernel-supported process or thread, e.g., control blocks for very lightweight threads, temporary storage, ready lists, interrupt stacks, etc. By mapping these objects to the same virtual address in each process, they may be addressed directly, avoiding a level of indirection. Another special use for partially shared address spaces is to provide access protection for logically private memory, e.g., for debugging.

(4) There are cases where a system that emulates a shared address space on top of private address spaces can be lazier than a genuine shared address space implementation. The additional laziness can translate into greater efficiency. For example, suppose an instruction stream allocates a new range of memory, uses it, and then deallocates it. In a straightforward kernel implementation of shared address spaces, the map-in operation can be performed lazily, but the map-out will require an expensive *shootdown* step on many systems, especially if the kernel-supported thread has migrated to another processor (Black *et al.* [35]). With an emulation of shared address spaces, it is simple for the map-out to avoid the shootdown if the instruction stream didn't migrate to other processes during the period of usage. Of course if a shootdown *is* required under emulation, it may have higher overhead, because of the extra kernel/user transitions required to send signals and make map-out system calls. But all these difference may be insignificant in practice, as the most efficient method of managing dynamic allocation of shared memory is to request rather large blocks from the kernel, manage them in user-mode, and never return them to the kernel until program termination.[97]

Providing a framework in which to evaluate the validity of these arguments is one of the research goals of Symunix-2. One must consider the cost of each operation, weighed by frequency, but a complete assessment cannot be based on quantitative factors alone. We must also evaluate, based on substantial experience, how *well* the system works: reliability, flexibility, portability, and ease of development/maintenance.

In addition to the question of shared versus private address spaces, an operating system must deal with the architectural diversity of memory system features such as cache options (e.g., replacement policies and coherence schemes) and physical memory placement (e.g., nearby or far away). The approach taken in Symunix-2 is to focus initially on the basic features common to machines in the target class, and then try to extend the software model to handle more variation without a fundamental change. This means that each architecture-dependent feature of the model should have a sensible implementation (such as to ignore it) even for machines lacking the corresponding hardware feature.

5.2. Kernel Memory Model

This section describes the conceptual model of memory management as presented by the Symunix-2 kernel to applications. Each process has a private address space that is

[97] Most versions of the standard C `malloc` routine work this way.

manipulated primarily by three system calls: `mapin`, `mapout`, and `mapctl` (§5.2.1). The `fork`, `spawn`, `exec`, and `exit` system calls (§5.2.4) also manipulate the address space, but their operation includes other complex actions as well. The central organizing concept is that the address space is a set of mappings from virtual addresses to locations in files. Files to which such mappings are directed are called *image files*, and they are said to be *mapped in*. If two address spaces include mappings to the same image file locations, the result is shared memory.

In general, any memory access instruction should be equivalent to some sequence of `read`, `write`, and `lseek` system calls directed at the appropriate image file. Likewise, any `read` or `write` system call should be equivalent to some sequence of load and store instructions (assuming the file locations are mapped in).

There are three areas in which an implementation can vary while still complying with this equivalence requirement:

(1) *Other memory-referencing instructions besides load and store.* Similar equivalences apply, but are necessarily implementation-specific. In general, an equivalent sequence of `read`, `write`, and `lseek` calls may also need to include ordinary computation.

(2) *Atomicity.*[98] In this definition of equivalence, we have ignored the issue of atomicity, since so many machine and OS implementation factors are involved. To give maximum latitude to the OS and hardware implementations, Symunix-2 does not require serializability of conflicting concurrent overlapping memory or file accesses. Note that the POSIX standard [113] doesn't guarantee any such serializability, nor do most UNIX implementations provide it, even on uniprocessors. Symunix-1 is unusual in that it *does* provide such atomicity, except for "slow" devices, e.g., terminals. Our decision to drop this feature in Symunix-2 was partially motivated by the introduction of mapped files and the issues discussed in this section. Portable software tends not to assume `read`/`write` system call atomicity.

Semantics of concurrent memory references are a separate issue, normally defined as part of the machine architecture or of a particular parallel programming model (e.g., *sequential consistency*, *weak consistency*, *release consistency*, etc.; see Mosberger's survey [154]). Memory consistency semantics are affected primarily by the order in which serially executed memory access instructions are carried out, and by the *cache coherence* scheme employed, if any.

(3) *EOF and holes.* The effect of a `mapin`, or memory access, directed to a location beyond the image file's End-Of-File, or within a "hole", is not fully defined. (In traditional UNIX systems, a hole is created by seeking beyond EOF and writing; the area in between may not be allocated space on the disk, but is guaranteed to read as zeros.)

An attractively simple approach would be to carry the equivalence between memory access instructions and file access system calls as far as possible. For example, a load directed at a hole would return 0 without allocating any new disk blocks. A store

[98] Here we refer to atomicity in the absence of hardware or software crashes.

directed at a hole would allocate a disk block and initialize the rest of it to zeros. A load directed past the image file's EOF would be treated as an error, but a store would extend the file and advance the EOF. Unfortunately, these semantics lead directly into implementation difficulties and efficiency problems. The problems stem from the byte resolution of a file's EOF: typical address translation hardware has a minimum mapping granularity greater than one byte and simulating byte granularity by trapping references near EOF is very slow. It might seem easiest simply to prohibit `mapin` of a hole, or beyond EOF, but such mappings can still result from a file truncation system call[99] after the `mapin`. File truncation has another problem as well: a requirement for serializability would necessitate invalidation of hardware address mappings, possibly affecting many processors; this is one aspect of the difficult problem known as *TLB consistency* (Teller [191]). Simply outlawing truncation of mapped-in files is attractive, but leaves the system vulnerable to denial-of-service security violations.

There are many possible ways of dealing with the problems of holes and EOF, with different tradeoffs. To allow maximum latitude for the implementation, we specify the mapped file semantics as little as possible. While this is not meant to be a formal specification, the following minimum requirements are needed for an implementation to be usable:

- System integrity must be maintained. (E.g., an in-use page frame must not be reallocated for another purpose as a result of file truncation, without appropriate action to revoke access.)
- At any given time, there must be at most one value associated with an offset into an image file. (E.g., truncating a file and then re-writing it must not result in two "parallel versions" of the file.)
- Neither `mapin` nor memory access instructions may directly modify an image file *unless* the mapping includes write permission. Filling in holes or changing the file size (EOF) are considered modifications in this context.
- The ability to write zeros implicitly to a file by seeking beyond EOF and calling `write` must be preserved. (Whether or not space is allocated for those zeros is up to the implementation.)
- At least for `mapin` requests with write access permission, it must be possible to map in a file over a hole or beyond its EOF point. This is important, because mapping in uninitialized portions of a file is the way new memory allocations are performed.

Within these requirements, significant variation is still possible. For example, the following possibilities all seem reasonable:

- A `mapin` system call for read-only access beyond a file's EOF or spanning a hole may, or may not, be allowed. Different restrictions might apply than when write access is requested.
- Future memory accesses may, or may not, be denied after a file is truncated. Access may continue to be denied after the file is extended again.
- In a read-only mapping, access to a hole or beyond EOF may succeed (return the last value stored or zero if none), be ignored (return zero or garbage), or result in a signal.

[99] An `open`, `creat`, `truncate`, or `ftruncate` system call.

- When an image file is truncated, the data beyond the new EOF may be deallocated immediately,[100] or delayed until the file is finally mapped out. If disk space deallocation is also delayed, this can temporarily exacerbate the distinction between a file's "size" and the disk space allocated to it.[101] Note that if disk space deallocation is delayed, a kind of denial-of-service attack is possible by simply getting a read-only mapping to a file owned by someone else; as long as the mapping exists, the owner's disk allocation can not be decreased.[102]
- The handling of I/O errors and lack of space on the file system are problems that must be addressed by any virtual memory system that makes use of secondary storage. The issues are the same in all such systems, including Symunix-2; the details are generally beyond the scope of this document. An I/O error could happen as a result of a page fault, in which case a signal can be generated synchronously with program execution. Or, the I/O error could be the result of an attempted prefetch or page-out, in which case user notification is more problematic.[103] It is possible to force out-of-space conditions to appear at `mapin` time, by reserving the maximum amount of space, or they can be eliminated by providing an explicit file preallocation capability, either as an extension to `open`, or as a new system call.

Rules like these can become very complicated; for practical purposes, it is wisest for an application to avoid all these problem areas. Nevertheless, *some* semantics must be chosen if the system is to be implemented.

5.2.1. The `mapin`, `mapout`, and `mapctl` System Calls

Mappings are added to an address space by the `mapin` system call. This call has the form

```
int mapin (void *vaddr, size_t size, int imagefd,
           size_t imageoffset, int flags)
```

where *vaddr* is the beginning virtual address of the new mappings, *size* is the size of the address range to be mapped, *imagefd* is the file descriptor of the image file, *imageoffset* is the offset within the image file of the location to be mapped at *vaddr*, and *flags* specifies options that may be *or*'ed together from the values given in Figure 27, on the next page.

Mappings are deleted from an address space by the `mapout` system call. This call has the form

---

[100] TLB consistency actions may be required, similar to those required for copy-on-write; see section 5.3.3 on page 236.

[101] These are reported by the `stat` system call in the structure fields `st_size` and `st_blocks`. Traditionally, the only factors that cause these numbers to disagree are the internal fragmentation of the final block, file system overhead ("indirect" blocks), and holes.

[102] A similar attack is available in *any* UNIX system: as long as a file is open (by any user), its i-node and disk allocation will not be decreased by removing (unlinking) it, although truncation is still effective.

[103] Many currently available commercial UNIX systems treat I/O errors while paging or swapping as fatal to the process; older versions would cause a complete system crash.

Flag	*Meaning*
AS_EXECUTE	execute permission
AS_WRITE	write permission
AS_READ	read permission
AS_CACHE_WT	write-through cache
AS_CACHE_WB	write-back cache
AS_EKEEP	`exec` action: keep
AS_SCOPY	`spawn` action: copy
AS_SDROP	`spawn` action: drop
AS_SSHARE	`spawn` action: share with parent

Table 27: Flags for `mapin` and `mapout`.

Both flags in the middle section request that the mapping be cached; they differ in suggesting write policy for caches supporting both write-through and write-back. Specifying neither cache flag stipulates that the mapping must not be cached. The `exec` and `spawn` actions are described in section 5.2.4.

`int mapout (void *`*vaddr*`, size_t` *size*`)`

where *vaddr* and *size* are as used in `mapin`.

Mappings have attributes, set initially by the *flags* parameter of `mapin`. Some of these attributes may be changed with the `mapctl` system call. This call has the form

`int mapctl (void *`*vaddr*`, size_t` *size*`, int` *flags*`, int` *how*`)`

where *vaddr*, *size*, and *flags* are as used in `mapin`. The *how* parameter specifies the manner in which the `mapin` and `mapctl` flags are combined: by performing *or*, *and-not*, or *assignment.*

The `mapin`, `mapout`, and `mapctl` calls are fairly strict about the addresses file offsets, and sizes provided by the user. For example, any size and/or alignment restrictions must be strictly adhered to by the user-supplied parameters; the kernel won't round addresses down or sizes up. This choice was made simply as a way to discourage "sloppy" practices that are likely to lead to surprises. Also, the user must provide a virtual address for `mapin`; the kernel won't pick one. This restriction, not present in some other contemporary systems, was made to provide flexibility for parallel program run-time support systems. It would be presumptuous (at best) for the kernel to make this kind of decision. We could add an option to `mapin`'s `flags` parameter, to specify that the kernel should choose the virtual address, but we see no good reason to do so.

The only size and address alignment restrictions defined for `mapin`, `mapout`, or `mapctl` are those imposed by an implementation. In most cases, on current hardware, this means that addresses and sizes must be multiples of a single *page size*. The nature of the file referenced by *imagefd* can also impact size and alignment restrictions in an implementation dependent way. Most file systems today require no restrictions beyond respect for a single minimum block size that is a divisor of the page size, so this is not often a problem. It is possible, however, to encounter a system with other requirements. To allow programmers to work in an environment with unknown restrictions, each implementation must provide a function advertising its requirements:

```
void
sizealign (void **vaddr, size_t *size, int imagefd,
           size_t *imageoffset, int flags)
```

This function can revise proposed *vaddr*, *size*, and *imageoffset* parameters for `mapin`, `mapout`, or `mapctl` to accommodate a wide range of implementation restrictions. The role of `sizealign` is purely advisory: the user may alter the modified values or ignore them completely, but `mapin`, `mapout`, and `mapctl` are allowed to be arbitrarily strict, as long as the true restrictions are properly documented and fairly represented by `sizealign`.

Although `mapin`, `mapout`, and `mapctl` form the core of the memory management model, there are some additional services to address certain functional and performance concerns. These are addressed in the following subsections.

5.2.2. The `mapinfo` System Call

When `mapin`, `mapout`, and `mapctl` were designed, it was assumed that processes would have some user-level code to "keep track" of their address spaces. For example, the user, not the kernel, makes the decision of which virtual address to allocate during `mapin`. This approach is simple and trouble-free most of the time, but it does result in a duplication of information between the user and kernel. (The kernel must also "keep track" of the address space, in cooperation with the hardware, to provide address translation.) The `mapinfo` system call is provided to allow a process to obtain the kernel's address space mapping information, either for itself or for another process (in accordance with suitable security provisions). For example, it is reasonable to call `mapinfo` at program start-up time to obtain information about the initial mappings.

```
int
mapinfo (int pid, void *start, void *info, size_t infosiz, int ninfo)
```

The *pid* and *start* parameters indicate the process and starting virtual address of interest, respectively, while the *info* parameter is the address where results should be placed, as an array of `mapinfo` structures. The *infosiz* parameter is equal to `sizeof(struct mapinfo)`, so as to allow new fields to be added over time without affecting binary compatibility with pre-existing programs; the total amount of space provided by the caller is *infosiz* `*` *ninfo* bytes.

5.2.3. Stack Growth

In any system supporting a programming language with a run-time stack, the responsibility for stack growth and stack space management is divided between the compiler, its run-time support library, the operating system, and the hardware. The language run-time stack is commonly kept in contiguous virtual memory, and grown by simply extending it in the proper direction, but there are other alternatives. *Stack growth* is this process of extending the stack and allocating the memory to be occupied by the extension. There are several possible approaches that may be reasonable in certain circumstances:

- *Preallocation.* The maximum stack space may be preallocated when a program begins execution. This is the simplest possible approach, but has the highest potential waste.

- *Automatic growth.* The hardware and operating system can cooperate to produce the illusion of preallocated stack space if memory references to unallocated memory can be trapped and restarted after extending the stack allocation. (This is the same hardware requirement as needed to support virtual memory with demand loading of pages or segments.) This is a popular solution, as there is no execution time overhead when the logical stack size is within the current stack allocation, and wasted space is minimized.[104]
- *Explicit growth.* At every point where the stack may be extended (e.g., in a function or nested block *prologue*), a comparison can be made between the logical and allocated stack sizes, and a system call can be issued to extend the allocation as needed. Depending on machine details, this can require quite a few extra instructions even in the most common case, when the current allocation is sufficient and the system call is unnecessary.
- *Implicit growth.* If the hardware doesn't meet the requirements for automatic stack growth in general, but can recover from at least one particular memory referencing instruction (by skipping it if necessary), there is a simple solution that is almost as cheap as automatic stack growth in the important common case. Each time the stack is to be extended, the furthest location to be referenced on the stack is simply referenced. If the current allocation is sufficient, no exception will occur and further action is unnecessary. Otherwise, an exception will occur and the operating system can interpret this as a special request to extend the allocation accordingly, because the faulting address is "close" to the existing stack.[105]

These are all well established solutions that have been used for many years. The related problem of detecting and handling *stack overflow* is usually ignored or handled as an error when the stack allocation can't be further extended. A common method of detecting overflow is to use a *redzone*, an unallocated region just beyond the current stack memory allocated. As the stack grows past the limit, an exception will occur on the first reference within the redzone. The redzone needs to be at least as large as the largest single growth that will ever be required, otherwise an attempt to grow the stack could "hop over" the redzone and create havoc with an unrelated memory area.

In a traditional serial environment, there is one stack and one heap in each address space. The heap is used primarily for memory not adhering to a stack-like lifetime (last allocated, first deallocated). It is convenient to grow the program's stack and heap towards each other from opposite ends of the address space, possibly with a redzone in between. In a parallel environment, many stacks may need to coexist in a single address range, making such a simple solution inapplicable. Having a very large address space helps considerably because a large maximum space can be set aside for each stack, but it can still be hard to build the system without uncomfortable limitations on the number of stacks and maximum size of each stack.

---

[104] *Stack space contraction* is also an issue. Some systems never reduce the stack allocation, even if the stack itself substantially shrinks. Other systems, such as Symunix-1, shrink the allocation only in restricted circumstances, such as when swapping a process out of memory.

[105] The definition of close is system-dependent. Traditional UNIX systems allocate a very large part of the address space for the stack, and accept any fault in that range as a legitimate stack growth.

An alternative is to place a limit on the size of a single stack frame, and grow the stack non-contiguously. This approach, often called a *spaghetti stack*, also lends itself to supporting a *saguaro* or *cactus* stack organization, where the stack frames of newly forked threads all emanate from the common parent's most recent stack frame.

In a system designed to support a variety of parallel and serial programming languages and models, the simple-minded automatic stack growth so successful in serial environments is inadequate. Symunix-2 offers each process a choice of using either explicit growth or *user-mode automatic growth*, wherein the language run-time system catches the `SIGSEGV` signal generated by a reference to unallocated memory. (Of course, an implementation can also support traditional kernel-mode automatic stack growth, as an alternative for serial programs.)

5.2.4. The **`fork`**, **`spawn`**, **`exec`**, and **`exit`** System Calls

Most address space changes are made separately for each mapped in region, but there are three important situations where a more massive change is appropriate: `fork`/`spawn`, `exec`, and `exit`. The most severe and trivial is `exit`, which destroys an address space, along with the rest of the calling process.

The `fork` system call in traditional UNIX systems duplicates the address space of the caller, to establish a new, nearly identical, child process. In Symunix, `fork` is a special case of `spawn`, which creates any number of children (§4.2/p148). In terms of address space handling, `fork` and `spawn` are identical. Within the address space of a single process, each mapping includes an attribute called the *child action*; it specifies what should happen to the corresponding virtual address range in a freshly created child process. There are three choices:

- *Copy.* Like traditional UNIX behavior for read/write memory, the child gets a copy of the parent's memory. This is done by implicitly creating a new image file, mapping it at the appropriate virtual address, and either performing an immediate copy or establishing copy-on-write (the choice being the kernel's).
- *Share.* Inherited shared memory is provided by duplicating the address mapping, including the same image file and offset.
- *Drop.* It is possible simply to omit a mapping when establishing the child.

The child action is determined by the flags used in `mapin` and `mapctl` system calls (§5.2.1/p215).

Another operation with major address space impact is the UNIX `exec` system call, which replaces the program running in a process. This call is not strictly necessary, in the sense that it is theoretically possible to emulate it by using `mapin`, `mapout`, and `mapctl`, but such emulation would be very tricky and machine-dependent, especially without placing restrictions on the new address space layout. On the other hand, `exec` is straightforward if implemented in the kernel, because it essentially operates in a separate address space. In Symunix-2, each address mapping includes an *exec action*, to specify its disposition when `exec` is successfully executed. There are only two choices:

- *Drop.* Analogous to traditional `exec` behavior, the mapping is removed from the address space, as if a `mapout` were executed.
- *Keep.* It is possible to pass information to the new process image by retaining some mappings.

There is another impact of `exec` on address space management: it sets up the new mappings needed for the program to start. These generally include such things as the *text* (machine instructions), initialized private data, uninitialized private data, initialized shared data, and uninitialized shared data. Initial stack placement and setup may also be required. In addition, `exec` is responsible for gathering and setting up the program's *arguments* and *environment*, provided as arguments to `exec`; they must be placed someplace within the address space.[106]

An immediate question arises for `exec`, given our introduction of the *keep* exec action: what happens if there is a conflict between an old and a new mapping? From a security point of view, it isn't acceptable to honor the old mapping in such a situation (recall the UNIX *setuid* and *setgid* bits). Our preference is to make such an `exec` fail with a suitable error indicator. Clearly the *keep* feature is useful only when the pre-`exec` program knows (or assumes) something about the address space layout of the post-`exec` program; this is simply a manifestation of the fact that some agreement or convention is required between the two programs for *keep* to be useful.

One obvious implementation approach for `exec` is to specify the new mappings, together with stack and argument/environment placement information, in the executable file of the program so the kernel can do everything (using one or more new image files, as appropriate). Another approach is to establish only some of the mappings within the kernel, and let the language *start up* code, the first user-mode instructions to gain control after `exec`, handle the rest of them. The details of whichever approach is taken form a critical interface between the kernel and users (as represented by program development tools and the language startup code). Of course, more than one approach can be used, just as more than one executable file format can be supported.

5.2.5. Image Directories

We have already indicated that, in certain situations, the kernel will need to create a new image file:

- For memory to be copied from parent to child in `fork` or `spawn`.
- For new program segments established by `exec`.

The new image file should be created with a name in the file system, so that it will be accessible by other processes for sharing or debugging. But what name should be used? There are two parts to the answer.

(1) Just as each process has root and current working directories, we introduce the *image directory* as the place where such kernel-created image files reside.

(2) The names selected for kernel-created image files are unspecified, and may be thought of as randomly selected by the kernel. A related mechanism is discussed in section 5.2.7 on page 221.

---

[106] Since the size of these aren't known until run-time, traditional serial UNIX systems put them at the base of the run-time stack (i.e., at a higher address than the stack for machines with downward-growing stacks).

The file name may be obtained via the `mapinfo` system call (§5.2.2). The image directory is specified to the kernel with the `setimagedir` system call.

5.2.6. Core Dumps and Unlink on Last Close

One of the central advantages of defining a memory model based on mapping files into memory is that it provides a simple and consistent way to provide and control access to that memory by other processes. This comes primarily because all memory can be named in the file system name space. A mapping is similar to any other open file reference, such as a file held open by a user, a current working directory, etc.; a file will not be deallocated until:

- the last such reference is dropped and
- the last name for the file in the file system is removed (via the `unlink` system call).

When the last mapping for a file is removed (via `mapout` or as a side effect of `exec` or `exit`) the image file may or may not continue to exist, depending on whether there are other open file references to it, and whether or not it still has a name in the file system.

It is easy to unlink a file immediately after creating and opening it; this is the standard way of obtaining unnamed temporary files in UNIX. Such temporary files can be used as image files, but cannot be shared except by inheritance over `fork` or `spawn` or by use of some mechanism to pass file descriptors between processes.[107] Recall that file space in UNIX systems is not deallocated until a file has no more links and is no longer open by any process.

In traditional UNIX systems, certain signals cause the kernel to produce a *core dump* by writing the process image into a file named `core` in the current working directory. Requirements for parallel systems such as Symunix-2 are more complicated, but core dumps can be supported by retaining image files in the file system (and writing a `core` file with some non-image process information, such as machine state and memory mappings in effect at termination). A simple mechanism to control unlinking of image files is needed.

The solution we adopt is a set of four new options for the `open` system call. When a file is opened, the `O_UNLINK` flag specifies that the file name should be unlinked when the last reference corresponding to this open is dropped. To do this, the kernel must remember the directory and file name from the given path, and verify that it still points to the file in question at the time the last reference is dropped. Recall that the file descriptor returned by `open` may be propagated by various other system calls, such as `fork`, `spawn`, `dup`, `mapin`, `superdup` (§5.2.7/p222), and possibly some IPC mechanisms.[108] Only when the *last* of these references is dropped will the file be unlinked.

The `O_UNLINK` flag is useful by itself, as it allows access to image files by other processes without the risk of leaving them around after program termination. Three more `open` flags extend this mechanism by allowing `O_UNLINK` to be suppressed for certain situations:

---

[107] Many systems, such as Mach, Amoeba, and Chorus, provide the ability to pass file descriptors or object capabilities via their own native IPC facility. Symunix-2 has the `superdup` system call, to be described in section 5.2.7 on page 222 as well as the 4.2BSD UNIX facility for passing file descriptors through UNIX domain sockets.

[108] As discussed in footnote 107.

`O_NO_CORE_UNLINK`
: Suppress the unlink if (dropping the last reference is) due to program termination by a core-dumping signal (e.g., `SIGQUIT`, `SIGSEGV`, `SIGBUS`, etc.).

`O_NO_UERR_UNLINK`
: Suppress the unlink if due to program termination with a non-zero exit status which, by convention, indicates a user-detected error.

`O_NO_0_UNLINK`
: Suppress the unlink if due to program termination with an exit status of zero, indicating normal termination.

While these options were conceived for image files to promote sharing and support core-dumping, they can be used with any file. For example, `O_UNLINK` by itself can be used to provide accessible temporary files (e.g., for logging the progress of a program) that are self-cleaning. `O_UNLINK | O_NO_CORE_UNLINK | O_NO_UERR_UNLINK` can be used to automatically delete a log file except when serious errors occur. `O_UNLINK | O_NO_0_UNLINK` can be used to ensure that a program's output file is retained only if it was successfully completed.

5.2.7. File System Optimizations

Although the design presented so far is functionally sufficient, there are some potential performance issues to be addressed in this section. The solutions proposed here are useful to some extent even in non-parallel systems, and even without the Symunix-2 memory model.

Lazy and Anonymous File Creation

The cost of creating a new file may discourage use of image files in certain ways. For example, memory regions that are virtually discontiguous, shared among a different set of processes, or used for very different purposes, might most naturally be mapped into different image files, but a programmer might be more inclined in practice to lump them together into a single image file to avoid the cost of creating so many files. We would like to use a *lazy* approach in the kernel to preclude a performance-related motivation for such user action.

What is the primary source of overhead during file creation? Normal disk buffer cache strategies can do an effective job of delaying writes for new data blocks, but this is not typically true for I/O related to file system structure and file naming. The directory must first be searched to see if the file already exists, and the new file name must be entered. Generally it is desirable to copy the file's i-node to disk and then flush the directory to disk before allowing further access to the new file, to ensure a recoverable file system structure on disk in the event of a crash.

There are two general ways to make this lazier:

- Perform weaker ordering of writes for file system integrity, so "safe" orderings (such as "write i-node before directory entry"), can be enforced without using synchronous writes.
- Delay specifying the file name, so it need not be looked-up and the directory need not be written out until later, if ever.

These two approaches are independent. The first is strictly an internal implementation issue, while the second requires an extension of the user-kernel interface. The file name look-up can only be delayed if the name itself is unspecified. This can be supported by the `open` system call with a new option flag: `O_ANONYMOUS`. This flag stipulates that the path name refers to a directory, and the file will be created with a system-determined name within that directory. The name only needs to be established in the following circumstances:

- If `mapinfo` must report the file name.
- If the directory is read to EOF (not scanning for a file look-up or creation, but read by a user process, e.g., a shell or the *ls* command). Any anonymous files in the directory must be named, and the names must be returned to satisfy the read.
- If the last open or mapping for the file is deleted, and the "unlink on last close" operation doesn't apply. In this case, the file becomes permanent.

This new option is also applicable as a way to create ordinary temporary files. In most if not all cases, the temporary file would never need to touch the disk, but some overhead would still be required to allow for the *possibility* of name generation. As a further extension, another flag could be used to specify that an anonymous file should *never* be named. This would provide truly anonymous temporary files, but in a more efficient way than the traditional practice of unlinking a temporary file right after creating it. The `O_EXCL` flag (fail if `O_CREAT` is specified and file exists) would have no other use in combination with `O_ANONYMOUS`, so it could be overloaded for this purpose.

Reference Descriptors

Even with anonymous file creation, as described on the previous page, a directory name must still be looked-up before a file can be created. One way to avoid this overhead is to create the file in the current directory, which has a very short name (e.g., `"."`, or even `""` in some systems). This is not only inflexible, but it won't even work in general, because the process may not have permissions to write in the directory. A better alternative has been designed for Symunix-2, *reference descriptors*.

In recent years, some UNIX systems have a peculiar device driver (or special file system type) that allows an `open` system call to do essentially the same thing as a `dup` system call. Typical usage is to pass a string like `"/dev/fd/1"` to `open`, and get a new descriptor that refers to the standard output [22]. We extend this in two ways:

- We generalize the numeric file descriptor pseudo-files, and allow them to appear at the beginning of a longer path. In such a case, the corresponding descriptor must refer to a directory, and the remainder of the path is looked-up relative to that directory.
- We shorten the name `/dev/fd/`*n*, referring to file descriptor *n*, to `/@/`*n*, and implement it as a special case in the ordinary path search algorithm (the infamous `namei` routine in the UNIX kernel). Furthermore, string such as `"/@/`*n*`/myfile"` is a path name for the file `myfile` located in the directory referenced by file descriptor *n*.

The result of these extensions is that additional directories may be referenced with the same efficiency as the current working directory or the root. In particular, a new anonymous file can be created with the call

```
open ("/@/n", O_ANONYMOUS, 0666);
```

where *n* is the string representation of a file descriptor for an open directory. The file can be created with no I/O, and without scanning the directory.

The `superdup` System Call

Open files in UNIX can be shared by two mechanisms:

(1) *Inheritance of file descriptors*. A forked (or spawned) child inherits copies of the parent's file descriptors. The descriptors are related in the same way as if the copy were performed by the `dup` system call. In particular, the two file descriptors share the same seek position (a seek on one affects the other as well).

(2) *Separate opens*. Two unrelated processes may open the same file by name, permissions permitting. The descriptors may have different access permissions, and have different seek positions (a seek on one has no effect on the other).

Some systems have introduced methods of passing file descriptors from process to process, generally as a feature of a more general interprocess communication facility. For example, systems supporting the Berkeley *socket* interface can pass file descriptors over connections in the UNIX domain. Systems with kernel supported threads generally share file descriptors among all threads in the same process, so no explicit passing is required.

A suitably efficient file descriptor passing mechanism can be used to emulate shared file descriptors without direct kernel support. While the result is unlikely to be more efficient than a direct kernel implementation of shared file descriptors, the following arguments suggest that the approach still has merit:

- *Flexibility*. More complex patterns of sharing can be established. For example, some file descriptors may be shared, and some not, or different processes may share different sets of descriptors.
- *Generality*. Efficient file descriptor passing can be used for more than just multithreaded computations (the processes involved need not be executing the same program image). For example, such file descriptor passing can be performed as part of a general access control scheme, or for after-the-fact I/O redirection.
- *Kernel Simplicity*. By omitting shared file descriptors from the kernel's feature list, it is kept smaller, both conceptually and actually. It is not necessary to design a single concurrency control mechanism for file descriptors that works well for all workloads.

A new system call, `superdup`, has been designed for passing file descriptors between processes with matching user IDs in Symunix-2. As its name implies, `superdup` is an extended form of the traditional `dup` system call:

```
int
superdup (int srcpid, int srcfd, int targetpid, int targetfd)
```

Like most UNIX system calls, −1 is returned for errors and `errno` is set accordingly; the target file descriptor is returned on success (normally the same as *targetfd*, but *targetfd* may be −1 to choose the lowest available descriptor in the target process, matching the normal descriptor allocation algorithm used by `open` and `dup`).

Because `superdup` deals with security in such a simple-minded way, it can be performed by a single process; most file descriptor passing schemes require explicit action from both sender and receiver. The `superdup` mechanism even allows the transfer to be performed by a third process.

Aside from the general issue of sharing file descriptors among processes of a parallel program, `superdup` is useful for emulating shared memory in user-mode, as discussed in section 7.6.2 on page 324.

5.3. Kernel Memory Management Implementation

Not only do we wish to provide a flexible kernel/user interface for memory management, but we also seek a flexible implementation to enhance portability and allow experimental exploration of alternative strategies. To this end we have identified the following modularization, depicted in Figure 9, below.

- *Address space management.* This module is responsible for the internal representation and manipulation of each address space, including the higher levels of system calls such as `mapin`, `mapout`, etc. It is divided into two sub-modules, the "upper" one being the bulk of the module and the "lower" one being "configuration dependent", i.e., it interfaces with and depends on the working set management and TLB management modules chosen for a machine.
- *Working set management.* This module is responsible for deciding what user memory is resident in physical memory and what isn't. By changing this module, alternative strategies can be provided, such as demand paging, swapping, full memory residence, or a variety of hybrids. This module is called only from "configuration dependent" code, namely the lower portion of the address space management module. Different working set management modules may have somewhat different interfaces and may place different requirements on TLB management and page frame management modules.
- *TLB management.* This machine-dependent module is responsible for interfacing with the hardware to provide the appropriate virtual address mappings. The term TLB refers to the Translation Lookaside Buffers commonly used in Memory Management Units. Since TLBs are usually private (per-processor) resources, this layer must adopt some strategy for *TLB coherence* to ensure that TLB entries used by different processors are consistent with the address space mappings maintained by the `mapin`, `mapout`, and `mapctl` system calls. The interface to the TLB management module is via the working set management module and various machine-dependent modules (e.g., low level trap handling). The TLB management module may be very aware of the approach chosen for working set management; alternate working set modules may assume different interfaces for TLB management.

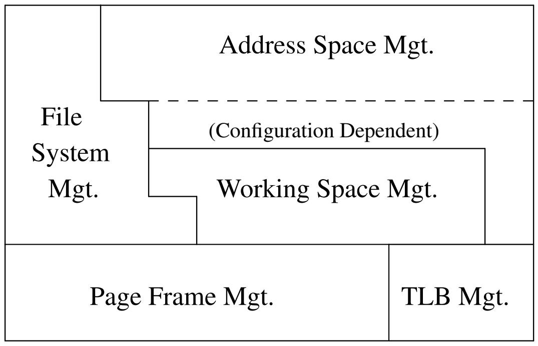


Figure 9: Memory Management Module Layering.

- *Page frame management.* This is primarily responsible for physical memory allocation and deallocation. Its interface with the rest of the kernel is fixed, but there may be cause for using different implementations on different machines, e.g., to use different memory allocation algorithms. MMUs with fixed or variable sized pages are supported.

The modules are further subdivided, and can be classified according to portability:

Machine-independent
: Some modules or sub-modules, as already identified, are machine-independent, i.e., they are written in a way that is applicable to all machines in our target class, and many others as well.

Machine-dependent
: Some modules or sub-modules are clearly machine-dependent, i.e., depend on hardware details specific to a particular architecture.

Configuration dependent
: Some modules or sub-modules are portable, i.e., could be applied to any machine in our target class but, for various reasons, aren't. In some cases, the configuration dependent code is tainted by knowledge of a machine-dependent interface. In other cases, alternate configuration dependent modules are provided to allow use of alternate policies or strategies, e.g., for experimentation, tuning for machine size, etc.

Because of the way in which the user/kernel interface is defined, the entire memory management system is also closely related to the file system. The four modules just identified are addressed more thoroughly by the following four subsections.

5.3.1. Address Space Management

There are many possible ways to implement this module, but the simplest is probably just a linked list of mapping objects for each address space, with a critical section used to protect it. A highly parallel structure is not required because we assume private address spaces (§4.1, §5.1/p210), and the only parallelism *within* a process is due to asynchronous system calls (§4.3.1) and page faults (§4.3.2). The average rate of asynchronous system calls and page faults per process is not expected to grow with the machine size. Although highly parallel solutions for address space management could be devised, there would almost certainly be *some* additional cost in time, space, and/or conceptual complexity. As another simplification, we require the system calls that modify the address space[109] to operate synchronously (i.e., they are only executed by the primary activity—§4.3). This naturally serializes address space modifications without additional overhead.

The address space is represented by a `struct aspc` for each process, and contains the list head for the mapping objects and a context-switching readers/writers lock (§4.5.2):

---

[109] These are `mapin`, `mapout`, `mapctl`, `fork`, `spawn`, `exec`, and `exit`. The ability to run these asynchronously doesn't seem useful. The first three are designed to run without blocking anyway, so it wouldn't normally make any difference.

```
struct aspc {
     struct msegp {    // mseg pointers
          struct msegp *forw;
          struct msegp *back;
     } ptrs;
     csrwlock lock;
};
```

Use of a readers/writers lock instead of a simple binary lock allows page fault processing to be overlapped for multiple activities within a process.[110] The mapping objects themselves are called *msegs* (*M*emory *SEG*ments); a `struct mseg` looks like this:

```
struct mseg {
     struct msegp ptrs;   // pointers
     void         *vaddr; // virtual address
     off_t        size;   // in bytes
     int          flags;  // attributes
     struct inode *ip;    // i-node of image file
     off_t        off;    // offset into file
     int          fdhint; // probable file descriptor
};
```

Both aspc and mseg may also contain machine-dependent information, including any needed to support the working set and TLB management modules. Msegs are split and joined on demand, in response to `mapin`, `mapout`, and `mapctl` system calls, but they are almost completely invisible at the user/kernel interface.[111]

The initial implementation uses a doubly linked list of msegs; a more sophisticated effort might use a tree to speed up searches when there are very many msegs. The mseg structures are dynamically allocated, but the exact mechanism is hidden behind a pair of functions `newmseg` and `freemseg`. (This simple modularity allows experimentation with different free list strategies for msegs.)

Because dynamic resource allocation may fail, address space modifications must be performed carefully. For example, the implementation of `mapctl` preallocates two msegs because, in the worst case, one mseg may need to be split into three. This conservative approach makes it easy to handle allocation failure before mseg manipulations are done (so they needn't ever be undone). In other cases, where there is no small upper bound on the number of msegs required or where configuration dependent code may indicate a failure, care must be taken to undo partial effects of a failed operation.

---

[110] There is no contradiction with the previous assertion that highly parallel techniques are inappropriate. In the first place, page faults don't modify the address space mappings, which makes them easier to handle concurrently. In the second place, this is a matter of I/O (not processor) efficiency: even a low degree of parallelism can improve the performance of slow I/O.

[111] The `mapinfo` system call returns information to the user that is very similar to a list of msegs; see section 5.2.2.

The address space management module provides the following functions:

`void asset()`

Set the processor's address space to be the one for the currently running process. This is normally called when a new activity begins execution and after context-switching (`cswitch`, §4.4.5), but can be omitted if another context-switch is imminent and no references to user space are needed. The current implementation of `asset` does nothing but make machine-dependent calls for TLB and cache management (e.g., to invalidate the TLB or cache, change hardware-supported address space identifiers, etc.).

`int asdup (struct aspc *`*parent*`, struct aspc *`*child*`)`

Duplicate the indicated parent's address space for the indicated child. This is called for `spawn` (by each child) or `fork` (by parent or child). Each of the parent's msegs is examined, and new msegs are created for the child according to the *child action* of the parent's mseg (§5.2.4):

- *Copy.* The child performs an *anonymous mapin*: the new mseg points to a new i-node, created with sufficient size to handle all the parent's *copy* action msegs.
- *Share.* The child's mseg contains the same information as the parent's, thus sharing the underlying memory. The shared i-node's reference count is incremented.
- *Drop.* No corresponding mseg is created for the child.

The return value of `asdup` is 0 for success, and an `errno` value for failure.

`int asexec1 (struct msegp *`*droplist*`)`

Handling of the UNIX `exec` system call is broken into three phases:

- `asexec1`, described here,
- a series of `mapin` calls, to set up the various needed address ranges, and
- `asexec2`, to commit to the new image.

Each of the current process's msegs is handled according to its *exec action* (§5.2.4); msegs that are to be dropped are temporarily moved to *droplist* for eventual disposal or restoration by `asexec2`. (The caller provides *droplist* as an uninitialized `struct msegp`.) Of course this approach means that the process consumes more resources during the time between `asexec1` and `asexec2` than would be required if the dropped msegs were not held to the end, but the implementation is much easier this way. The return value of `asexec1` is 0 for success and an `errno` value for failure. In the case of failure, `asexec2` is called automatically to move msegs back from *droplist* to the normal `struct aspc` list before returning.

`void asexec2 (struct msegp *`*droplist*`, int` *commitflag*`)`

After calling `asexec1` and making an appropriate number of calls to `mapin`, the exec must be completed by calling `asexec2` with the droplist used for `asexec1` and a *commitflag*: *TRUE* to finally drop the msegs on *droplist*, or *FALSE* to move them back to the normal `struct aspc` list. Because all possible failures have been handled by `asexec1` and `mapin`, there is no return value for `asexec2`.

`void asdestroy()`

Free all address space resources for the process; called as part of process termination.

`struct mseg *vtomseg (struct aspc *, void *)`
: Given a user virtual address for the indicated address space, find and return a pointer to the mseg that maps it (`NULL` if the address is invalid).

`int mapin (void *`*addr*`, off_t `*size*`, int `*flags*`, struct inode *, off_t `*off*`)`
: Implement the essence of the `mapin` system call (§5.2.1/p214), returning 0 for success or an `errno` value for failure. Adjacent virtual addresses are checked for possible mseg merging, so `mapin` can consume or release 0 or 1 msegs. Currently, `mapin` fails if any existing mseg overlaps the range given by *addr* and *size*. The strategy taken to deal with holes in the image file (see §5.2) is to have `mapin` fail if the mapping is to be read-only (i.e., `AS_WRITE` is not specified with *flags*), and to "fill in" the hole (with zeros) otherwise.[112]

`int mapout (void *`*addr*`, off_t `*size*`)`
: Implement the essence of the `mapout` system call (§5.2.1/p215), returning 0 for success or an `errno` value for failure. Msegs are split as necessary, so `mapout` can consume 1 mseg or can release 0 or more msegs. Machine-dependent functions are called by `mapout` to handle problems such as TLB or cache coherence, if required. We ignore unmapped subranges within the range indicated by *addr* and *size*, but if the entire range is invalid, `mapout` fails.

`int mapctl (void *, off_t `*size*`, int `*flags*`, int `*how*`)`
: Implement the essence of the `mapctl` system call (§5.2.1/p215), returning 0 for success or an `errno` value for failure. The action of `mapctl` is to alter the affected msegs' flags according to *flags* and *how*, with *how* specifying the function to apply: *OR*, *AND-NOT*, or simple assignment. Machine-dependent functions are called by `mapctl` to handle problems such as TLB or cache coherence, if required. Msegs are split when necessary and adjacent virtual addresses are checked for possible mseg merging, so `mapctl` can consume as many as 2 msegs, or release some number of msegs. We ignore unmapped subranges within the range indicated by *addr* and *size*, but if the entire range is invalid, `mapctl` fails.

`int mapinfo (int `*pid*`, void *`*addr*`, void *`*info*`, size_t `*infosiz*`, int `*ninfo*`)`
: Implement the essence of the `mapinfo` system call (§5.2.2), returning an `errno` value for failure or 0 for success. The process (and hence, address space) to examine is specified by *pid* (real and effective user IDs must match); a *pid* of 0 indicates the current process. An array of *ninfo* `mapinfo` structures, each of size *infosiz*, is provided in user space for filling in by the kernel to describe address ranges beginning with *addr* (all address ranges if *addr* is `NULL`). Upon success, the number of filled in structures (a number between 0 and *ninfo*); is returned to the user as a secondary value.[113] The reason for including *infosiz* as a parameter is to allow future extensions to the system by defining new fields at the end of the `mapinfo` structure; old

---

[112] To simplify the implementation, we currently don't undo the hole filling in all cases of `mapin` failure. This is a resource consumption problem, not a semantic problem, since unfilled holes read as zeros anyway.

[113] A secondary value is stored into a dedicated field of the activity structure, and copied out to the user as part of the low level system call return code.

programs compiled with the old definition will still work.

We ignore unmapped ranges completely in `mapinfo`; they do not cause failure.

`void asfault (struct aspc *, void *`*addr*`, int `*rwx*`, int `*uflag*`, md_asfault_t `*md_arg*`)`

Handle an address translation failure for the user address specified by *addr*; `asfault` is called only by machine-dependent trap handling code. The address space lock must be held (non-exclusively). The original memory reference causing the fault may have been a read or a write or an instruction fetch; this is indicated by *rwx* (4=read, 2=write, 1=execute). The original reference might have been issued from either user- or kernel-mode; this is indicated by *uflag* (1=user-mode, 0=kernel-mode).

Asynchronous page faults are handled by setting the `ACT_OPTIMASYNC` flag in the activity (§4.4.2/p162); this is not done if `SIGPAGE` is masked or ignored, or if *uflag*=0. The primary work of `asfault` is to search for an mseg to translate *addr* and, if that succeeds, call `i2pf` (in the file system management module; see section 6.2.6) to produce a page frame to satisfy the reference. The activity may block for I/O in `i2pf` (and lose primary status, if `ACT_OPTIMASYNC` is set); but that depends on the policies of the working set management module. Finally, `asfault` calls the machine-dependent routine `md_asfault` to carry out whatever final action is required, e.g., loading a TLB entry, setting up hardware page table entries, calling a working set management function, etc. One of the arguments passed to `md_asfault` is *md_arg*; this adds flexibility, allowing the original (machine-dependent) fault handler to pass information through (the machine-independent) `asfault` to (the machine-dependent) `md_asfault`. Not only does `md_asfault` complete handling for successful translations, but it also makes decisions about handling failures. This allows `md_asfault` to take special actions if desired, e.g.,

- Handling of page and file size rounding problems.
- Sending `SIGSEGV` or `SIGBUS` with special machine-dependent arguments.
- Performing automatic stack growth.

Configuration Dependent Hooks

Each of the above-mentioned address space operations (`asset`, `asdup`, `asexec1`, `asexec2`, `asdestroy`, `vtomseg`, `mapin`, `mapout`, `mapctl`, `mapinfo`, and `asfault`) makes calls to one or more configuration dependent functions, to allow flexibility in the interfaces to machine-dependent modules (e.g., TLB management) and the working set management module. For example, `asdup` calls the following configuration dependent functions:[114]

- `md_asdup1`: called before doing anything else, e.g., to allow for initialization of the configuration dependent fields within the child's `struct aspc`.
- `md_asdup2`: called after scanning all of the parent's msegs, allocating msegs for the child, and determining the total size of all *copy* action msegs. This allows any configuration dependent fields in the child's msegs to be set up properly.

---

[114] Although these functions are considered configuration dependent, we give them names beginning with `md_` because configuration dependent choices are necessarily machine-dependent.

- `md_asdup3`: called after performing anonymous mapin for *copy* action msegs. This is the last chance for the configuration dependent code to indicate failure.
- `md_asdup4`: called after completing initialization of *copy* action msegs. The actual memory copying is the responsibility of `md_asdup4`, although it merely passes responsibility to a working set management function, which may choose to copy now or later, e.g., using copy-on-write.

Each `md_asdup`*x* call, except `md_asdup4`, has the opportunity to return 0 for success or an `errno` value for failure. Clearly, the details of the configuration dependent calls depend on the implementation of the main address space management functions, e.g., in this case, `md_asdup2`, `md_asdup3`, and `md_asdup4` all "know" the approach taken to implement the main `asdup` function, such as the use of a single anonymous mapin for all *copy* action msegs, even if not virtually contiguous.

Address Translation

Most address translations are successfully completed by the MMU without software intervention. The rest of the time, `asfault` is called, as described on the previous page. There are four possible reasons for a translation fault:

(1) The referenced virtual address is invalid, i.e., there is no mapping from that address to an image file. This results in a `SIGSEGV` signal or automatic stack growth, as appropriate.

(2) A mapping to an image file exists for the referenced virtual address, but the permissions associated with the mapping do not allow the type of access attempted, e.g., a store to a read-only location. This results in a `SIGBUS` signal[115] or possibly a configuration-dependent action such as *copy-on-write* (§5.3.3/p236).

(3) A mapping to an image file exists for the referenced virtual address, but the relevant page is not in memory. This is a classic page fault, and is serviced by performing I/O.

(4) A mapping to an image file exists for the referenced virtual address, and the page is in memory, but not currently mapped by the MMU. This situation is common on machines using *software TLB reload*. It is also possible, on machines with hardware TLB reload, to use the hardware-supported page table like a giant TLB with software reload. This method has been used in the Mach system.

5.3.2. Working Set Management

The point of all the configuration dependent parts of address space management is to allow different approaches for machine-dependent functions (such as TLB management, §5.3.3) and policy functions, such as working set management; this section addresses the latter.

The ultimate goal for the working set management module is to permit effective and efficient implementations of any reasonable policy for memory residence in virtual memory management. In particular, we will outline two alternative working set management module implementations to illustrate the flexibility of the basic design:

---

[115] The name `SIGBUS` stems from UNIX on the PDP-11, where a "bus error" trap occurs in these situations.

- *A memory resident policy*, in which all virtual memory is always resident in physical memory.
- *A demand paging policy*, based on an explicit working set model (Denning [53]).

There is no fixed interface to the working set management module, it is devised to meet the needs of the configuration dependent portion of the address space management module, and is called from no where else. The working set management modules themselves need not be changed for different machines, although the best choice might be different on different machines. For the two policies we will examine in this section, the main address space management module remains unchanged but the configuration dependent portion is rewritten to make calls to different working set management modules (and possibly different TLB management modules as well). Furthermore, on machines with sufficiently different architectures, the configuration dependent portion may be changed to call yet different TLB management modules. Assuming there are W different machine-independent working set modules, and the system is to run on M significantly different kinds of machines, this ultimately leads to creation of $W \times M$ different TLB management modules and configuration dependent portions of the address space management module. Fortunately, we anticipate neither W nor M growing very large and it is unlikely that every working set module will be appropriate for every machine type.

Memory Resident Policy

This degenerate policy places all virtual memory in physical memory and keeps it there until it is unmapped. The MMU is used for address translation and protection, but there are no page faults.[116] This policy, seemingly anachronistic for computing in 1995, is worthy of exploration for several reasons:

- It is relatively easy to implement.
- It is simple to explain and helps prepare the reader for the more elaborate design to follow.
- It stretches the bounds of freedom for the working set management module in an unusual direction, and thus helps validate the claim that we allow a great deal of flexibility.
- As a degenerate policy, it offers the opportunity to evaluate the performance cost and benefits of more elaborate policies.
- It might be the most efficient policy, especially for a machine with a limited workload.

In the memory resident policy, a `mapin` system call allocates and initializes page frames for all referenced pages that are not already memory resident. Each page frame is reference counted so it can be released when no longer needed (this is part of page frame management, §5.3.4). The reference count is used for any working set policy, not just the memory resident policy (see section 5.3.4 on page 238); what distinguishes the memory resident policy from others is that the page frame reference counts are incremented at `mapin` time and not decremented until `mapout` time.

---

[116] There can still be TLB misses, which on some machines cause traps for reloading by software, but these are not page faults in the traditional sense: no I/O is required to satisfy them. In principle, although we do not describe such an approach, even *copy-on-write* or *copy-on-reference* traps could be implemented in a fully memory resident working set policy.

As already stated, the interface to the working set management module is dependent on the needs of the address space management module. For example, the `asdup` system call's configuration dependent functions were listed in section 5.3.1 on page 229. We provide the following memory resident policy functions in support of `asdup`:

- `wsmem_asdup3`: called by `md_asdup3`, this function calls `i2pf` (§6.2.6) to preallocate space for the new "anonymous mapin" memory required for all *copy* action msegs.
- `wsmem_asdup4`: called by `md_asdup4`, this function performs memory-to-memory copies for all *copy* action msegs; the memory allocated by `wsmem_asdup3` is the target of these copies.

In between these operations, the main `asdup` function completes the initialization of child msegs, notably assigning msegs to subranges of the anonymous image file. Of course, `wsmem_asdup3` can signal a failure if `i2pf` returns an error (e.g., for lack of physical memory or lack of file system space).

Demand Paging Policy

The Least Recently Used page replacement policy, or an approximation thereof, is an important part of many demand paging systems. In principle, we could use the method described in section 3.7.6 on page 136 to implement such a policy:

- Prior to using a page, we could use a variant of *cachesearch* (§3.7.6/p137) to find the corresponding page frame if it is in memory.
- When the desired page isn't in memory, the least recently used frame could be be reassigned, as in `_cache_reassign` (§3.7.6/p140), possibly after writing back to secondary storage if dirty. An unordered list of completely free (clean and unassigned) page frames could also be maintained to satisfy most page faults without having to wait for a dirty page to be written.
- After use, the page frame could be inserted at the most recently used end of an LRU list, as in `cacherelease` (§3.7.6/p140).

There is a serious problem that prevents this from being a practical solution: overhead. A true LRU replacement policy requires that page use be defined by individual user memory references, thus these LRU/search list operations would need to be performed for each such reference. This unacceptable overhead is the reason why true LRU replacement is not implemented for machines with conventional memory management hardware. Most operating systems approximate LRU replacement with an algorithm such as "second chance", "clock", etc. (see standard OS textbooks, such as Tanenbaum [190] or Silberschatz *et al.* [180]); we will describe here an approach that bears a more direct resemblance to true LRU. The essence of our approach is to use software TLB reload to derive approximate usage information for each page. We assume the TLB is significantly smaller than a realistic working set size.

We can choose to implement page replacement globally, with a system-wide LRU/search list, or locally, with an LRU/search list per process. An advantage of local replacement is that we can afford to use simpler (i.e., less parallel) data structures than the ones described in section 3.7.6 on page 136, which are needed for global replacement. The remaining discussion pertains primarily to local replacement, although much of the same approach is applicable to global replacement.

Because each page in a working set is memory resident, it has a page frame with a reference count giving the number of working sets to which it belongs. Each process maintains its own working set with an LRU/search list, using virtual page number as search key. Every page in a working set is represented by a small data structure we will call a *working set item*. Based on `cacheitem` (§3.7.6/p137), a working set item includes a reference count to record the number of times the page is referenced by a TLB or outstanding I/O operation (this count is typically 1, and only rises when asynchronous system calls or page faults are running.

When a memory reference isn't satisfied by the TLB, we follow a modified `cachesearch` algorithm (§3.7.6/p137), hoping to find the page in the working set. If found, we increment the working set item's reference count, remove it from the LRU list to reflect its in-use status, and load it into the TLB. Otherwise, the reference is to a memory location outside the working set, so we call `asfault` (§5.3.1/p229).

When `asfault` finds a valid mapping for a faulted virtual memory address, it calls `i2pf` to produce the page frame; `i2pf` will allocate a new frame (and perform I/O if necessary) if the needed page isn't found in memory by searching. This search is by {device, disk block} pair and is based on the `bcache` module we will describe in section 6.1.2 or other file-system-specific function, as we will explain in section 6.2.6. Allocating a new page frame is performed by calling a page frame management routine, `pf_new` (§5.3.4/p240), which obtains one from the global completely free page frame list or the working set's LRU replacement list (if the completely free list is empty). Either way, `asfault` calls `md_asfault` after `i2pf` returns; `md_asfault` loads the appropriate TLB entry, allocates a working set item with reference count 1, increments the page frame's reference count, and possibly evicts another page. When a page is evicted from the TLB, its working set item's reference count is decremented and, if zero, the working set item is re-inserted into the working set LRU list (at the most-recently-used end). (A page evicted from the TLB may still be in use, e.g., for I/O or by another activity within the same process running concurrently on another processor.)

Because of sharing (a page frame may be mapped into the address spaces of more than one process, or into a single address space more than once), the mapping from LRU/search list items to a page frame will be many-to-1, but there is no need to map back from a page frame to all such items. Page frame structures contain reference counts, used in page frame management to control allocation and deallocation of page frames (§5.3.4/p238). The page frame reference count is incremented when the page enters into a working set and decremented when when leaving the working set.

A page is pushed out of the working set and becomes eligible for reassignment when it is deleted from the least recently used end of the LRU list. If the page is dirty, it must be written to backing storage before being reassigned; we do this by extending the general `cachesearch` and `_cache_reassign` operations described in section 3.7.6 on page 137. If necessary, a dummy item can be used, as described on page 131. After the dirty page is written back, if necessary, the page frame's reference count is decremented and, if zero, the working set item is deallocated and the frame is freed for general reallocation[117] (if not needed immediately for another purpose).

---

[117] The page is freed by calling `brelse` (§6.1.2) or other file-system-specific function, corresponding to the look-up function we will discuss in section 6.2.6.

By a simple modification of the basic search/LRU scheme described in section 3.7.6 on page 136, it is possible to provide an operation we call `wsadjust`, to delete items from the LRU list if they have been on the list more than a specified amount of time, releasing them to the page frame management layer (or initiating write-back on them, if dirty). By augmenting the list items to include a time stamp,[118] we thus implement a close approximation to the working set page allocation policy, often described in the literature but rarely implemented.

Because we don't assume the hardware provides a page reference time stamp in each TLB entry, we have to approximate such a value. We do this by assuming pages resident in TLBs are always in use (and hence, part of the working set by definition), and marking the corresponding working set item with the current process virtual time when a page is evicted from the TLB. This approximation inflates the working set, since it is possible for the TLB to contain entries that aren't frequently referenced (although this effect is reduced by the small size and low associativity of most real TLBs). We can refine the approximation by periodically examining all TLB entries and clearing the *referenced* or *valid* bits; a page that is unreferenced during the previous interval should not have it's time stamp updated.

It is not quite enough to time stamp the pages in the working set, we must also trim the set to exclude those pages not referenced within the working set window. We can do this by calling `wsadjust` for each process at periodic intervals; this can be done either with separate timer events for each process, or with a single timer event and the "all processes" visit list (§4.4.1/p159).

A reasonable demand paging system must have a way to shed load when memory becomes over committed. One or more system activities can be dedicated to this task. Many policies are possible, such as releasing memory of processes blocked for a long time or suspending and "swapping out" low priority processes. Assuming such over commitment is rare, the "all processes" visit list is a reasonable way to implement this; if it is the normal state of affairs, it might be worthwhile to maintain an always ordered list of potential swapping victims, e.g., using an algorithm based on group lock, similar to that suggested in section 4.9 on page 207.

The major problem with the local replacement strategy we have just outlined is memory overhead. Even if the working set LRU and search structures are implemented in a space-efficient way (e.g., with simple serially-accessible linked-list data structures, *not* the highly parallel one described in section 3.7.6 on page 136, which requires $O(N)$ memory for an N processor machine), each process still requires $O(wss)$ working set items pointing to page frames, where *wss* is the working set size in pages. If the number of processes and the average working set size both grow with N, quadratic memory overhead will result. Memory sharing between processes doesn't help; in fact, without memory sharing we would expect the sum of all working set sizes to grow only linearly with N (otherwise thrashing would occur and users would reduce the load).

---

[118] This requires hardware support in the form of a high-precision clock. We need a clock to measure process virtual time, which is easy to derive from a simple free-running real-time clock. Many current microprocessors already have such a clock, or something similar (including the AMD 29050 used in the Ultra-3 prototype), because it is generally useful. A counter of instructions executed or memory references completed would be even better; the latter can be implemented with the FPGAs on the Ultra-3 processor board.

One alternative is to implement a global replacement policy, using a single highly parallel LRU/search structure, as in section 3.7.6 on page 136, to construct and maintain a "global working set". Another possibility is to use a *family working set* approach, where the family, as described in section 4.2 on page 148, corresponds closely to the idea of a parallel program in execution. We expect much less inter-family memory sharing than intra-family memory sharing, thus limiting the expected number of working set items per page frame to a small number, independent of N. As a result, the total overhead should only grow linearly with N. The family is also the logical unit for swapping, rather than the individual process, when it comes time to shed load. Ultimately, however, swapping must be carried out on a per-process basis.

Managing the page use timestamps becomes more difficult with per-family replacement. An ideal *family virtual time* source might be an accurate per-family memory reference counter, incremented for every memory reference of any family member, although this would require varying the working set window size according to family size. A more realizable method is to synthesize a family virtual time source by using a local time source together with Fetch&Max:[119] whenever an activity begins running, the local virtual time source is set to the synthetic family value, and when the latest time value is needed or the activity stops running, the synthetic family value is updated with Fetch&Max.

5.3.3. TLB Management

The most fundamental job of the TLB management module is manipulation of the MMU: handling TLB miss traps, formatting and loading TLB or page table entries, choosing entries to replace, dealing with the change of address space accompanying a context-switch (`asset`, §5.3.1/p227), etc. The implementation of most of these operations is dominated by hardware considerations (you have to work with what the hardware provides), but software usually has significant options. For example:

- Software TLB reload can be used to emulate conventional page table structures or vice versa.
- A small hardware TLB can be augmented with a larger *pseudo TLB*, such that TLB misses that hit in the pseudo TLB are much less expensive than full misses.

TLB management also must deal with some less hardware determined issues, such as maintaining *TLB consistency*. We have shoved much of the problem up to the user level by adopting a private address space approach (§7.6.2), but some issues still remain in the kernel, especially page eviction and copy-on-write, which we discuss in turn.

Page Eviction

With a demand paging working set policy, page eviction becomes a serious potential cause of TLB inconsistencies: some processor might still have a TLB entry for a page that has been evicted from memory.

Many solutions to TLB coherence have been implemented or proposed. The solution we advocate here has been called *read-locked TLB entries* (Teller [191]), and requires no special

[119] Fetch&Max(a,v) replaces a with max(a, v) and returns the old value of a. A software implementation is given by Greenberg *et al.* [96].

hardware support. The page replacement strategy outlined in section 5.3.2 on page 232 is in fact intended to support this TLB consistency approach by precisely identifying those pages not eligible for eviction because they are part of a working set. Because pages are only placed in the LRU list when they are evicted from the TLB (from the last TLB, if a per-family or global replacement strategy is used), we can guarantee that no page translation remains usable in any TLB if it is not in any working set.

We define the *coverage* of a TLB to be the maximum amount of memory mapped by the TLB at any time; this can be applied to a software-reloaded TLB, a software-implemented pseudo-TLB, or an MMU with hardware TLB reload from page tables being used as a giant TLB. To prevent deadlock, we require the total coverage of all TLBs in the system to be less than the total physical memory of the system; for good performance it should probably be much less.

Copy-On-Write

Generically speaking, copy-on-write is an optimization for a copying operation, where the target object is remapped to point at the source and both are marked to disallow modification. When and if modification is attempted, the affected object is copied, mappings are again separated, and the original access permissions are restored. The most common application of copy-on-write is for memory copy operations, with the page as the minimum-sized object.[120] Smaller or improperly aligned copy operations are not optimized with this technique.

We view copy-on-write and related strategies such as copy-on-reference as configuration dependent optimizations at the implementation level, rather than as fundamental machine-independent features.

In Symunix-2, the opportunity for copy-on-write springs from the `read` and `write` system calls, which may be viewed as file copy operations since user-addressable memory exists only in image files. Because of our integrated memory and image file model, this requires careful management of the i-node → page frame mappings[121] as well as the virtual → physical address mappings. Consequently, an implementation of copy-on-write affects not only on the TLB management module, but also cuts across working set management, and page frame management.

There are a large number of ways to implement copy-on-write, but they are typified by three general approaches:

- *Do it the hard way*. In general, a mechanism such as *shootdown* must be used to prevent use of any old references for source or target page frames that may be present in any hardware or software TLB structures, LRU lists, etc., throughout the system.

---

[120] In section 6.4 we consider issues related to using copy-on-write as a file system optimization for on-disk data blocks.

[121] Traditional UNIX systems perform the {i-node, offset} → buffer mapping in two steps, first converting {i-node, offset} → {device, block number}, and then using a system-wide disk block cache to convert {device, block number} → buffer. As we will describe in section 6.2.6, it is possible in most cases to perform the overall mapping in one step, without using the disk block cache.

- *Do it only when easy*. Copy-on-write is considerably easier when neither the source nor the target is currently shared with other processes. However, subsequent sharing (before a modification attempt) can still cause problems that require TLB consistency actions.
- *Don't do it*. This is the easiest to implement and the lowest overhead when the pages in question are almost certain to be modified.

Unless rather large page sizes are supported (so as to amortize the high per-page costs over more data), we generally favor the "don't do it" strategy, but small machines, and especially uniprocessors, are likely to benefit from the "only when easy" approach. Unfortunately, these solutions don't match the usage pattern of message passing, which has motivated the use of copy-on-write in systems like Mach. On the other hand, even better optimization for message passing may result from using a different model; a possibility along these lines is discussed in section 5.4 on page 248.

Nevertheless, copy-on-write may be implemented in Symunix-2 for those `read` and `write` system calls with appropriate alignment and sufficiently many bytes to transfer to make it worthwhile. Such a `read` or `write` system call makes one or more calls to `vtomseg` (§5.3.1/p228); this reduces the problem to copying between files. Ultimately, the decision of whether or not to perform the copy-on-write optimization is made by a call to `pf_copy`, properly a part of the page frame management module (§5.3.4/p242). The purpose of `pf_copy` is to copy page frames (which is why it's a part of page frame management), but it gets enough information to make decisions about copy-on-write and interact with the working set and TLB management modules if necessary; as we have already pointed out, copy-on-write potentially cuts across all these modules.

Some of the information to be considered by `pf_copy` includes the i-node reference counts,[122] mapin counts,[123] and page frame reference counts.[124] Inode locking may be required to prevent conflicting accesses (see section 6.2.4 on page 261). The TLB coherence strategy chosen for a particular configuration will dictate the actions required to maintain consistency of hardware TLBs, pseudo-TLBs, and working set LRU lists. Enough "hooks" are provided by the machine-independent kernel to allow configuration dependent support for a per-i-node visit list of processes (or families) mapping the respective i-node; this can be helpful in reaching all relevant address space structures, but requires constant maintenance, adding to the overhead of `mapin`, `mapout`, `spawn` (or `fork`), `exit`, and `exec`, even when no copy-on-write optimizations are advantageous.

5.3.4. Page Frame Management

Page frame management in Symunix-2 is concerned with issues of allocation and deallocation. The design is driven by the varied needs of higher-level modules and the desire to permit a wide range of implementation alternatives. We support multiple higher-level uses of page frames with a common free list. Particular expected uses include a general purpose disk block cache (§6.1.2) or a file system implementation that does its own cacheing but other uses can also be accommodated (p238).

[122] How many processes currently have each i-node open.

[123] How many processes currently have address space mappings for each i-node.

[124] An upper bound on the number of TLB and LRU list references for each page frame.

Variable Size Pages

We have adopted a variable-size page frame model for Symunix-2, although only some MMUs support mixed page sizes. The advantages of supporting multiple page sizes include:

- *Better TLB coverage without excessive fragmentation.* The total amount of memory accessible without a TLB miss depends on the number of TLB entries and the amount of memory translated by each. A larger page size allows more coverage, but a smaller one reduces internal fragmentation.
- *Variable size buffers.* High performance file systems tend to employ variable size disk blocks, requiring variable size buffers in memory. Because the file system buffer cache is fully integrated with the virtual memory system, it is advantageous to use the same mechanism for each.
- *Reduced overhead.* Larger pages reduce overhead because there are fewer of them. Some significant overhead factors are more dependent on the number of pages than on their cumulative size (especially cumulative page fault time and total page frame allocation time for a large amount of memory).

These factors are unrelated to parallelism, except that large parallel machines (and the applications run on them) tend to require large amounts of memory, generally in proportion to the number of processors.

A disadvantage of variable-sized pages is that the physical memory allocator has to cope with external fragmentation. It is quite possible that a large frame can't be allocated even when there is plenty of space available in smaller chunks.

The page frame management interface is designed so that single-size paging systems are easily handled in one of two ways:

(1) As a degenerate case, with the minimum and maximum frame sizes being equal.

(2) With software emulation of large page frames in terms of small ones.

Page frames in Symunix-2 range in size from `MINPAGESIZE` to `MAXPAGESIZE`, and are restricted to powers of two. Each frame must begin at a physical address that is a multiple of its size, to match simple MMU implementations and to permit use of a buddy system allocation policy.

Interface

A page frame is represented by a `pframe` structure:

```
typedef struct _pframe {
      int           pf_rcnt;      // reference count
      struct bio    *pf_bio;      // for I/O
      union pf_use  pf_use;       // configuration-dependent
      unsigned char pf_flags;
      unsigned char pf_type;
#if MINPAGESIZE < MAXPAGESIZE
      unsigned int  pf_size: LOG_NPAGESIZES;
#endif
      MD_PFRAME                   // machine-dependent stuff
} pframe;
```

The most basic fields within the `pframe` structure are the reference count, flags, type, and size information. The size information is implicit in the degenerate case where `MINPAGESIZE` and `MAXPAGESIZE` are the same; this is the reason for using `#if`. Additional fields are also included in this structure by way of the machine-dependent macro `MD_PFRAME`. It is important to keep the overall size of a `pframe` as small as possible, since a lot of them will be allocated (one per page frame); this is why we declared `pf_size` as a bit-field of `LOG_NPAGESIZES` bits. Further attempts at bit-packing may be fruitful with `pf_flags` and `pf_type`, as well as the fields introduced by `MD_PFRAME` and `union pf_use`. The page frame management functions are listed in Table 28, below.

Function	*Purpose*	*See Page*
ip `pf_new` (*s*, *mins*, *rpf*, *f*, *t*)	Allocate 1 frame, or multiple smaller	240
r `pf_1new` (*s*, *f*, *t*)	Allocate 1 frame only, size *s*	240
v `pf_free` (*pp*, *n*, *f*)	Tentatively deallocate frames	240
v `pf_1free` (*p*, *f*)	Tentatively deallocate 1 frame	240
v `pf_unfree` (*p*)	Reclaim freed frame (advice)	241
v `*pf_getpaddr` (*p*)	Get physical address of frame	241
s `pf_getsize` (*p*)	Get size of frame	241
r `pf_grow` (*p*, *s*)	Attempt frame growth in place	241
r `pf_split` (*p*)	Break frame in 2 parts	241
ip `pf_flush` (*d*)	Write dirty frames (`sync`/`umount`)	242
v `pf_copy` (*ps*, *is*, *os*, *pt*, *it*, *ot*)	Copy frame, possibly copy-on-write	242
v `fake_pf_init` (*fp*)	Initialize for `fake_pf_new`	243
i `fake_pf_new` (*fp*, *s*, *n*, *f*)	Test hypothetical allocation	243

ip = `int` (number of frames>0, failure<0 (negative `errno` value), 0 not used)
i = `int` (success≥0, failure<0 (negative `errno` value))
r = `struct rtrn_pf` (`pframe *` and `errno` value)
v = `void`
s = `unsigned int` (page frame size)
mins = `unsigned int` (minimum page frame size)
rpf = `pframe **` (pointer to result array of page frame pointers)
f = `int` (flags)
t = `unsigned int` (page frame type)
pp = `pframe **`
p = `pframe *`
n = `int` (number of page frames)
d = `dev_t` (`NODEV` for `sync`, otherwise `umount`)
ps, *pt* = `pframe *` for copy source, target
is, *it* = `inode *` for copy source, target
os, *ot* = `off_t` for copy source, target, in i-node
fp = `struct fake_pf *`

Table 28: Page Frame Management Functions.

Flag	*Meaning*
PF_TRANSIENT	Frame will be used only a short time
PF_PERMANENT	Frame will probably be used forever
PF_NOBLOCK	Indicate EWOULDBLOCK instead of blocking

Table 29: Flags for pf_new.

Frames allocated with neither PF_TRANSIENT nor PF_PERMANENT are assumed to be for long-term use, e.g., referred to by an address space mapping. Note that, depending on the working set module, such pages may still be subject to eviction or swapping out of memory; this is a crucial distinction from PF_PERMANENT.

Type	*Meaning*
PF_TYPE_FREE	Frame is free and may be allocated for any purpose
PF_TYPE_BCACHE	Frame is allocated to the buffer cache (§6.1.2)

Table 30: Page Frame Types.

Additional values can be assigned to specify other higher-level uses. One of these values is stored in each pframe's pf_type field, and one (but not PF_TYPE_FREE) must also be passed as a parameter to each pf_new or pf_1new call.

Allocation. The basic allocation function is pf_new, which attempts to allocate a page frame of size *s*, but may return a set of smaller page frames if a single *s*-sized frame is not currently available. Substitution of smaller frames is controlled by the *mins* parameter, which specifies the smallest acceptable fragment size; by setting *mins* = *s*, the request will only succeed if a single frame of the specified size is available. Smaller frames must all be of the same size, so pf_new will allocate

1 frame of size *s*, or
2 frames of size *s*/2, or
4 frames of size *s*/4, etc.,

as long as the resulting page frame sizes are each ≥ max(MINPAGESIZE,*mins*). Additional pf_new parameters specify flags, listed in Table 29, above, and the "type" to be assigned the page frame(s), listed in Table 30. Page frame types are integers used to identify the higher-level purpose of the page frame; they are not interpreted by the page frame module except when calling higher-level functions for page frame reassignment (p241) and pf_flush (p242). The return value of pf_new indicates the number of page frames allocated (and hence their size); negative values indicate failure. The pframe pointers themselves are returned via the *rpf* parameter, a pointer to a caller-provided array.

As a convenience, the call pf_1new(*s*,*f*,*t*) is essentially the same as pf_new(*s*,*s*,*rpf*,*f*,*t*); exactly 0 or 1 page frame is allocated, and the return value includes the page frame pointer.

Free List. Page frames are placed on the free list by the pf_free and pf_1free functions; the former works with an array of *n* pframe pointers, while the latter provides greater convenience for the common case of a single frame. In most cases, when we speak of pf_free, we include pf_1free as well.

The free list plays essentially the same role as the LRU list in section 3.7.6 on page 136 and pf_free behaves like _cacherel (p141): without the PF_FREE_REALLY flag, frames go on the free list in such a way that they may be reclaimed for their previous use. Before a frame may be reassigned by pf_new, a function of the higher level module (determined by the frame's pf_type field) is called for approval. Higher level modules, such as the buffer cache, are considered to "own" their freed frames until such approval is given or pf_free is called with PF_FREE_REALLY.

A higher level module may advise the page frame management module that a frame is definitely not available for reassignment by calling pf_unfree, but giving such advice is not required. Likewise, a valid implementation of pf_unfree may do nothing or may remove the frame from the free list.

Use of Reference Count. The pf_rcnt reference count field in the pframe structure is set to 1 by pf_new but not otherwise used by the page frame management module; it is provided simply because of expected need by higher level modules. Use of pf_rcnt to determine when to call pf_free is completely up to the higher level modules and is not required.

Physical Address and Size. Once a pframe is obtained, the functions pf_getpaddr and pf_getsize may be used to obtain the physical address of the page frame and its size, respectively; this information may come from explicit fields within the pframe[125] or from other sources. For example, pf_getsize returns a constant if MINPAGESIZE = MAXPAGESIZE, and pf_getpaddr may return a calculated address if pframe structures are statically allocated in an array, one-for-one with actual minimum-sized page frames; these are implementation issues.

Growing and Splitting Page Frames. The interface allows for the possibility of growing and splitting page frames. A successful call to pf_grow will return a (possibly new) pframe pointer for a double-sized page frame. The direction of growth depends on the frame's physical address:

```
if (pf_getpaddr(p) & pf_getsize(p))
        growth downward
else
        growth upward
```

A successful call to pf_split returns a pframe pointer for the upper half of the frame; subsequently the two pframes are completely independent. Of course, the implementation may cause either of these operations to fail for any reason, but we expect that an aggressive file system buffering strategy will use them to reduce copying overhead and internal fragmentation.

[125] Possibly expanded from MD_PFRAME.

Flag	*Meaning*
PF_UNINIT	Uninitialized
PF_DIRTY	Must be written back to disk
PF_ERROR	An I/O error has occurred

Table 31: Page Frame Flags.

These flags are provided in the expectation that they will be useful to higher level modules; they are not generally used by the page frame module. An exception is PF_DIRTY, which is examined by pf_flush. Additional flags may also be defined for use by higher level modules.

Page Frame Role in Input/Output. The page frame management module has no real knowledge of I/O, disk block addresses, etc. Those concepts are under the jurisdiction of other modules, but we provide some assistance and cooperation:

- Each pframe can point to a bio structure, to be described in section 6.1.1; it is used by the block I/O module.
- The pframe structure includes a union, pf_use, with a member addressing the needs of each higher level module. In particular, the general buffer cache module, described in section 6.1.2, adds device, disk block, and hash table information (typically three "words" of memory).[126]
- The flags in the pframe structure (Table 31, above) can specify that a frame is "dirty" or that an I/O error has occurred. These concepts are of little significance to the page frame module.
- The pf_flush function assists in performing the sync and umount system calls by using the pf_type field of each potentially affected page frame to select and call a higher level module function. If the argument to pf_flush is NODEV, it indicates a sync operation and the appropriate higher level function is called for PF_DIRTY pages only. Otherwise, the higher level function is called for all pframes (with the device number as parameter), allowing it to perform any necessary write-back operations and release each page frame associated with the indicated device (by calling pf_free with PF_FREE_REALLY). Additional low-level mechanisms are provided to execute the operations in parallel via RBC if appropriate (§3.6), and await the completion of any I/O; the higher level module merely has to provide a function to handle a single pframe.

Copy-On-Write. As already described in section 5.3.3 on page 237, the pf_copy function is provided as a "hook" for the machine-dependent implementation to perform copy-on-write if appropriate. A minimal implementation will simply do a real copy, ignoring all parameters except the source and target page frames (which must have the same size). A somewhat more aggressive implementation will closely interact with the file system, address space, working set, and TLB management modules to perform copy-on-write under favorable conditions.

[126] This union is defined by a machine-dependent header file that determines machine configuration details.

Hypothetical Allocation. A single number can serve to represent an amount of memory in a system with a single page size. This is important for such quantities as the amount of memory currently available, the maximum amount of memory that may be allocated in a particular system, the virtual and physical memory sizes of a process, etc. Many of these quantities are still useful in a variable size page environment, but one particular use is invalidated: determining if the physical memory requirements of a process can ever be met. It is important that a process not be created or grown if it will never be able to obtain sufficient page frames to run.

This determination is the responsibility of the working set module but, without the ability to rely on single numbers for the maximum allocatable number of page frames in the system and the total page frame requirement of a process, we must provide an alternative. This is the purpose of `fake_pf_init` and `fake_pf_new`. These functions, which operate on a `fake_pf` structure, allow *trial allocations* to be performed. These trial allocations have no effect on real allocations via `pf_new`, as indicated by the prefix “fake”. A `fake_pf` structure includes an array of $1 + \log(\texttt{MAXPAGESIZE}) - \log(\texttt{MINPAGESIZE})$ counts, to indicate how many frames of each size would be available if all non-permanent frames in use were released. (This is one purpose of the `PF_PERMANENT` flag; another is to give the implementation the flexibility of performing such allocations in a different way, such as from a special arena.) Fake allocations are performed by `fake_pf_new`, simply reducing the appropriate values in the array so as to reflect what a real buddy allocator would do (starting with the requested size and “splitting” larger blocks if necessary). The `fake_pf` structure must be initialized by `fake_pf_init`; for these uses, the `fake_pf` structure is normally allocated on the stack. There is no need for a `fake_pf_free` operation. The return value of `fake_pf_new` is a simple boolean indicating success or failure of the hypothetical allocation.

It is also important not to allow `PF_PERMANENT` allocations to prevent a process currently in the system from ever running. Specifically, such allocations must not reduce the total allocatable memory below the minimum requirements of any single process in the system. The way this is handled is configuration dependent, and may vary for different working set management approaches. A reasonable choice is to use a `fake_pf` structure to represent the minimum requirements of each process and another to represent the maximum of all processes. Increases in the maximum can be handled by using group lock (§3.5.6). Decreases are more difficult; there are several possibilities, including:

- Never decrease the amount that must be available for processes; let the value be a high-water mark instead.
- Periodically recalculate the maximum.
- Trigger recalculation only when a process reduces its requirements from the maximum.

Some care must be taken to avoid excessively serializing other operations, such as `fork`, `spawn`, `exit`, `mapin`, and `mapout`.

Implementation Issues

We have already indicated several implementation issues during the course of describing and motivating the interface for page frame management:

- The number of page sizes supported has wide impact on the implementation.
- The use of the functions `pf_getsize` and `pf_getpaddr` to provide implementation flexibility.

- The use of `MD_PFRAME` within the `pframe` structure to allow for additional, implementation-specific, fields.
- The free list can be implemented in many ways because of the definition of `pf_free`, `pf_unfree`, and the mechanism of calling a higher level function to approve page frame reassignment.
- Full support for `pf_grow`, `pf_split`, and `pf_copy` is optional, giving additional flexibility to the implementation.

There is a spectrum of possible free list organizations, depending on when "freed" frames are approved for reassignment. At one extreme, each `pf_free` call can seek such approval immediately, resulting in almost no opportunity for reclaims; at the other extreme, freed frames can stay in an LRU list as long as possible, until `pf_new` needs to reassign them. In between are a range of solutions combining an LRU list of unapproved frames with another list of "truly" free, approved, frames. The latter list contains frames freed with the `PF_FREE_REALLY` flag, and possibly some approved frames taken from the LRU list. Allocation of a page frame by `pf_new` is performed from the approved list if possible, resorting to the LRU list (and obtaining the required approvals) only if necessary. A variety of strategies may be employed to move frames gradually from the LRU list to the approved list, with the goal of satisfying most requests without consulting the former.

There are many possible allocation schemes for the approved list. For single page size systems, a list such as one of the set algorithms in section 3.7.5 is an obvious and appropriate choice for our target class of machines. There is a substantial body of literature about variable size allocation algorithms, but the page frame model we have presented is a *buddy system* model, so we concentrate on that. The advantages of a buddy system are speed and simplicity; these are achieved at some sacrifice in utilization due to restrictions on block size and alignment.

Buddy System Algorithm

A buddy system memory allocator provides blocks of size 2^m, 2^{m+1}, ..., 2^{m+n-1}, where n is the number of block sizes supported. Not only are block sizes restricted, but starting addresses are also: a block must begin at an address that is a multiple of its size. (A block of size 2^i must start at an address with the i least significant bits equal to zero.) Any block with size greater than 2^m may be split into two half-sized *buddies*.

For each allowed size 2^i, a list, *bfree*(i), keeps track of free blocks. The first step in allocation is to check the appropriate free list; if that fails, a larger block can be allocated and split if available. The (serial) algorithm is as follows:[127]

[127] We relax our pseudo-code notation in this section, deviating further from C, in order to succinctly represent the algorithms at a more abstract level. We retain some of the C operators, notably `&` (bitwise "and"), `|`, (bitwise "or"), `^` (bitwise exclusive-or), and `~` (unary bitwise complement).

```
buddyalloc(size)                    // Serial version
        j ← ⌈ log size ⌉
        if j ≥ m + n
                return FAILURE
        b ← dequeue (bfree(j))
        if b ≠ FAILURE
                return b
        b ← buddyalloc(2^(j+1))
        if b ≠ FAILURE
                enqueue (bfree(j), b | 2^j)
        return b
```

We assume the dequeue function simply returns the address of a free memory block, or an invalid address we call *FAILURE*. Likewise, enqueue accepts such an address as its second argument. Of course a real implementation might be more efficiently coded in iterative form, but we find this recursive form convenient for presentation.

In a buddy system allocator, failure to find a free block of the desired size is handled by trying to get a larger block and split it into two buddies. In the pseudo-code just presented, this is the subject of the last four lines. The expression $b \mid 2^j$ is the address of the *upper buddy* and b is the address of the *lower buddy*; this works because any block allocated must begin on an address that is a multiple of its size, hence the jth bit of the address b is guaranteed to be zero.

Just as a buddy system must sometimes split a block into its two buddies, the deallocation algorithm is responsible for merging buddies together again:

```
buddyfree(b, size)          // Serial version
        j ← ⌈ log size ⌉
        bb ← b ^ 2^j
        if j < m + n − 1 && remove (bfree(j), bb)
                buddyfree (b & ~2^j, 2^(j+1))
        else
                enqueue (bfree(j), b)
```

This time, bitwise exclusive or (ˆ) must be used to compute a buddy's address, since we don't know whether we're looking for an upper or a lower buddy. A central aspect of the deallocation algorithm is the "remove" function, which searches a free list for a particular block, removes it if found, and returns a boolean result to indicate success or failure. When merging two buddies, b & $\tilde{}2^j$ is the address of the lower buddy, which is also the address of the merged block (min(b, bb) would work as well).

The first obvious step to parallelizing these algorithms is to use a parallel list structure for the *bfree* lists, such as those described in section 3.7. In this case, the list may use any ordering discipline, but must support interior removal.

An important free list issue is how the list item data structures representing the memory blocks are implemented. An attractive method is to place the item structures in the memory blocks themselves, but this creates the requirement that the items are never accessed when the blocks aren't free (because they exist only when the blocks are free). As a result, the items-in-blocks approach makes the design of a list algorithm supporting interior removal somewhat tricky. (In particular, the algorithms described in section 3.7 are not suitable for the items-in-blocks approach.) In this section, we assume items are statically allocated apart from the blocks, one item for each minimum sized (2^m) physical memory block.

The simplest way to "parallelize" the serial buddy algorithm is simply to protect it with a busy-waiting lock (§3.5.3). Our first attempt, in Symunix-1, was just a little more ambitious: we implemented the free lists with parallel access queues, similar to `dafifos`, described in section 3.7.1 on page 82. This worked, but sometimes the system would run out of large blocks even if nothing was allocated. The problem was that sometimes two buddies would get on the free list individually, instead of being merged into a larger block. For example, imagine two buddies being deallocated concurrently; each fails to find its buddy via interior removal, so the opportunity to merge is lost. Symunix-1 actually ran real user workloads with this algorithm for awhile, but sometimes it would get into a deadlock-like state, where all processes were swapped out, but memory was too fragmented to swap the most needy process in. A quick band-aid was applied to detect this state, scavenge the buddy free lists, and perform all possible buddy merges. Under heavy load, it happened every day or so.

The next improvement, suggested by Eric Freudenthal [85], was to prevent the simultaneous deallocation of upper and lower buddies by using a readers/readers lock (§3.5.5) for each block size:

```
buddyfree(b, size)        // Parallel version
        j ← ⌈ log size ⌉
        bb ← b ^ 2^j
        if b < bb
                bwrr_xlock(bfrr(j))        // lower buddy
        else
                bwrr_ylock(bfrr(j))        // upper buddy
        found ← j < m + n − 1 && remove (bfree(j), bb)
        if not found
                enqueue (bfree(j), b)
        if b < bb
                bwrr_xunlock(bfrr(j))
        else
                bwrr_yunlock(bfrr(j))
        if found
                buddyfree (b & ~2^j, 2^(j+1))
```

The busy-waiting readers/readers lock for blocks of size 2^j is $bfrr(j)$, and holders of X and Y locks may not execute concurrently. By synchronizing in this fashion, there is no race between two buddies being freed concurrently.

Although this approach effectively prevents any "lost merges", and was heavily used by Symunix-1 users, a race in block allocation still causes very poor memory utilization in some circumstances. Imagine a time when there are no small blocks and only a few large blocks available. Suddenly, many processors simultaneously request small blocks. Since there aren't any, all such processors go looking for larger blocks. Because there aren't enough large blocks for all processors, some requests will fail, even though there is plenty of memory available. (See section 8.3.1 on page 334 for an example of this kind of scenario.)

We now describe an enhanced buddy allocation algorithm devised to solve the problem by forcing processors to cooperate rather than compete on concurrent requests. The basic idea is to keep a count, for each block size, of failed requests that need to split a larger block. Processors making even requests go looking for larger blocks to split, while processors making odd requests wait to get the buddies thus produced. For each block size, we use

- An integer, *bfailed*(j), to count the failed requests for each size. The first failed request is numbered 0.
- A busy-waiting counting semaphore, *bmerge*(j), for the odd requests to wait on.
- An unordered list, *bspare*(j), to hold the spare buddies produced by even requests that will satisfy odd requests.
- A readers/readers lock, *barr*(j), to exclude requests and splits from interfering with one another.

It is fairly easy to see how this can work if requests always came in pairs, but how do we handle an odd number of requests? When a larger block is split, we decrement *bfailed*(j) by 2; if this drives it to -1, then we know there were an odd number of requests, so we set it to 0 and put the spare block on the normal free list (*bfree*(j)). Otherwise, the spare buddy goes on *bspare*(j) and we signal *bmerge*(j) to allow the waiting request to complete.

```
buddyalloc(size)                         // Parallel version
        j ← ⌈log size⌉
        if j ≥ m + n
                return FAILURE          // size too large
        bwrr_ylock (barr(j))
        b ← dequeue(bfree(j))
        if b ≠ FAILURE
                bwrr_yunlock (barr(j))
                return b
        x ← fai(bfailed(j))
        bwrr_yunlock (barr(j))
        if x mod 2 = 1                   // odd requests must wait
                bws_wait(bmerge(j))
                b ← poolget(bspare(j))
                return b
        b ← buddyalloc(2^(j+1))          // get larger block
        bwrr_xlock (barr(j))
        x ← faa(bfailed(j),–2)
        if x = 1                         // unmatched request, buddy not needed
                bfailed(j) ← 0
                if b ≠ FAILURE
                        enqueue (bfree(j), b | 2^j)
        bwrr_xunlock (barr(j))
        if x ≠ 1                         // matched request, buddy needed
                if b ≠ FAILURE
                        poolput (bspare(j), b | 2^j)
                bws_signal(bmerge(j))
        return b
```

The effectiveness of this algorithm at reducing unnecessary allocation failures is shown in section 8.3.1 on page 336.

5.4. Future Work

Impact of Kernel Thread Support

We have repeatedly resisted the temptation to add shared address spaces to Symunix-2 (§5.3.1/p225). It would, however, be of scientific interest to quantify the performance impact of this design issue.

Working Set Mangement

In section 5.3.2 we presented a framework for construction of alternative working set management strategies. This area seems ripe for further evaluation, and is well suited to quantitative analysis. In particular, we would like to know the impact of global vs. per-family vs. per-process page replacement policies under various real workloads.

Message Passing

As mentioned in section 5.3.3 on page 237, the potential benefit of copy-on-write may not always be available, especially for message passing styles of interaction. An alternative with more potential than copy-on-write is to adopt the semantics of *move* instead of *copy*. The poor-man's way of doing this requires no kernel interface changes: unmap the message from the sender's address space before telling the receiver to map it in (the image file can be open by both processes, and a small queue implemented in user-mode shared memory can be used to pass information about such files).

A fancier approach is to provide optional forms of `read` and `write` that are allowed to destroy their source as a side effect;[128] this can be used directly to move data from one (image) file to another. New operations such as these have very little to do with the implementation of memory management if implemented in an "only when easy" style, e.g., use copying if either source or target image file is mapped-in more than once or if the target is already memory resident.

Checkpoint/Restart

A *checkpoint* is an operation executed to preserve the state of a computation in such a way that it may be *restarted* at some later time. In principle, any program can be modified to explicitly perform checkpoint and restart operations, but the most interesting checkpoint/restart facilities are transparent, being implemented entirely by some combination of the language compiler, run-time system, and operating system. There are many problems to solve in implementing checkpoint and restart transparently, mostly concerning a program's relationship with the "outside world": the file system, communication links with people,

---

[128] The source can be "destroyed" by converting it to be a *hole* in the file. Recall that UNIX systems have traditionally supported holes in files, regions that occupy no secondary storage and read as zeros. Writing to a hole causes secondary storage to be allocated. Holes are traditionally created by seeking beyond the current EOF position of a file and writing; the EOF position is the "high water mark" of writes to the file.

processes, or machines, active memory sharing with other programs, and the passage of real time. The state of each of these things is difficult or impossible to properly checkpoint and restore. Nevertheless, with some restrictions, a checkpoint/restart system can be valuable for many long-running programs as a form of fault tolerance.

Because all user-accessible memory in Symunix-2 exists as image files in the file system, it should be a good base for a checkpoint/restart mechanism. Full consideration of all the issues is far beyond the scope of this dissertation, but for the purpose of preserving the state of user-accessible memory, the "image file" model seems well suited. A checkpoint solution is "simply" a matter of stopping the processes comprising an application program, copying the image files, recording the remaining processes states (machine registers, open file information, address space organization, process identity, etc.), and allowing them to continue. In other words, a checkpoint is a superset of the operations required for a "core dump" (§5.2.6).[129] A restart can be achieved by generalizing the basic concept of the `exec` system call to accept the information saved by the checkpoint and recreate the original arrangement. Clearly there are many issues here that must not be overlooked or trivialized, such as proper synchronization while checkpointing the state of a cooperating collection of processes, but the Symunix-2 design offers no unique obstacles, and the presence of image files is convenient. The resulting saved state will naturally represent the memory sharing relationships among the checkpointed processes, however complex they may be. The Symunix *family* concept (§4.2/p148) appears to be a convenient unit for checkpointing multi-process programs, but any complete set of cooperating processes will do.

There are several possible ways to implement checkpoint:

- *In the kernel*. In a sense, this is the easiest because the kernel has access to all existing information about the processes involved and has authority to carry out all actions directly. Implementing a major function, such as checkpoint, in the kernel lacks flexibility, however.
- *In user-mode*. It is especially attractive to consider implementing checkpoint as a user-mode signal handler for a new signal type, say, `SIGCKPT`. This makes the entire implementation and all the policy decisions user-replaceable, with the potential of taking advantage of application or run-time specific information (e.g., perhaps this is a good time to perform garbage collection).
- *External*. A user process, separate from those being checkpointed, could be responsible for checkpoint; like the kernel solution, this has the advantage of putting the implementation in one place rather than scattered among the affected processes. The disadvantage is that the agent has somewhat more limited control and must operate with potentially clumsy interfaces to control and examine the complete state of target processes.

Each approach has advantages and disadvantages, and each probably requires some extensions to the basic operating system design already presented.

---

[129] The important special case of "checkpoint and terminate" is quite similar to making a core dump; it has practical importance for long-running applications or daemons in response to an operator-initiated system shutdown. It should also be possible to design a checkpoint/restart system such that a core dump is similar to a degenerate checkpoint, one that a debugger can fix up and restart.

As a special case worthy of optimization, consider how to efficiently handle a sequence of checkpoints, where only the most recent one is deemed worthy of saving. By alternately "swapping" each image file with a corresponding checkpoint file, we can minimize the number of image file pages that must be written to secondary storage to bring the image files up to date. When those writes are completed, the image files and checkpoint files can be swapped, and the program allowed to continue. The "swap" operation also involves passing all in-memory buffers from the old image file to the new one, and possibly copying some non-memory-resident pages from the old one to the new one; this is similar in effect to setting up a copy-on-write relationship between the files,[130] but possibly more efficient. This new swap operation can't be implemented without kernel changes, since it relates strongly to the internal representation of files and buffering.

5.5. Chapter Summary

The memory and address space management framework presented in this chapter is unconventional but, at the heart, based on a familiar model: an abstraction of a hardware memory management unit. Being built upon a virtual machine (the Symunix process), it has considerable more power and flexibility than a real MMU, however.

We have taken advantage of the extra flexibility, both by specifying functionality in a UNIX context (such as relying on the UNIX file system for backing storage), and by unifying and extending several distinct features of the traditional model, such as various kinds of memory sharing, flexible core dumping, and the behavior of `fork`, `spawn`, and `exec`. The new model is also rich enough to support new functionality, such as checkpoint/restart.

The primary goals of this design were avoidance of serial bottlenecks and flexibility for user-mode run-time systems. The resulting system makes no decisions about how to use various parts of the user's address space; all address space allocation decisions are the responsibility of user-mode software, not the kernel. Since our address spaces are private, per-process, entities, we impose no serialization or coherency actions for shared address space coherency. Our approach is to provide a memory model rich enough to support a wide variety of memory sharing patterns and coherence strategies.

We also presented a modular design for the implementation of this memory management model, with sharp attention paid to the machine- dependent/independent boundaries. At the lowest level, we gave a new highly parallel algorithm for a memory block allocator based on the binary buddy system.

[130] We will also address the issue of copy-on-write for files in section 6.4.

Chapter 6: Input/Output

Many architectural and hardware issues of I/O management and performance are independent of parallel processor design. The major exception is bandwidth, due to the large capacity and high performance possible in parallel designs. A full discussion of all these issues and their operating system implications goes well beyond what this dissertation can accomplish. Instead, we concentrate on some generic synchronization and resource management issues for our target class of machines, independent of actual input and output data transfers.

The chapter is organized as follows: Section 6.1 presents our approach to buffer management, consisting of modules for block-oriented I/O and buffer cache management. File systems are addressed in section 6.2, and considerations for device drivers are covered in section 6.3. After discussing some ideas for future work in section 6.4, we summarize the chapter in section 6.5.

6.1. Buffer Management

Traditional UNIX kernels maintain a system-wide cache of disk blocks in memory, called the *buffer cache* (Bach [15], Ritchie and Thompson [170]). The size of the buffer cache is generally fixed at system boot time. Each buffer in the cache consists of a structure used to describe a disk block and to record the status of an I/O operation, together with the actual memory to hold a copy of the disk block. Older systems use fixed-sized buffers, but many newer ones have a pool of page frames, used as necessary to make buffers of various sizes matching disk block sizes.[131] In all traditional UNIX systems, the bulk of memory is managed separately from the buffer cache as page frames for allocation to user processes.

In an effort to utilize physical memory better, more modern systems have merged the virtual memory and buffer management systems, so the same underlying pool of page frames is available for either purpose, managed by a single set of policies.

While Symunix-1 uses the traditional buffer cache structure with fixed size buffers, Symunix-2 follows a more modern approach, supporting an overall memory model of mapping files into memory (§5.2). The remainder of this section pertains to the Symunix-2 design.

[131] Note that using non-contiguous page frames for a buffer relies on the hardware to perform address translation for I/O transfers or to support an efficient *scatter/gather* operation.

Because every page frame is a potential disk block buffer, we have split the functionality of the traditional UNIX `buf` structure (Bach [15]) into three separate structures and implemented them with separate modules within the kernel:

`struct pframe`
: As described in section 5.3.4 on page 238, the page frame module maintains a `pframe` structure for each allocated page frame in the system. The `pframe` structure contains the minimum information necessary to describe the page frame and interact with the other two structures, described next.

`struct bio`
: The block I/O module provides support for device drivers of block-oriented storage devices, such as disks. It should not be confused with the notion of *block device* in UNIX terminology (Thompson [196]); our `bio` module is also used for *raw* devices. A `bio` structure is assigned to a page frame for the duration of an I/O operation and also provides a convenient point for synchronization, since the `pframe` structure itself contains no locks or other synchronization mechanisms (because it is important for the `pframe` structure to be small).

`struct bcache`
: The buffer cache module provides for look-up of page frames associated with [device, block number] pairs, in the manner described in section 3.7.6 on page 136. The page frame module supports the notion of a "higher level use" for each page frame (§5.3.4/p240); the `bcache` module is an example, used by UNIX *block device* drivers and file systems. Of course, use of the `bcache` module is optional, and it is also possible to implement less generic cacheing strategies for such purposes. A `bcache` structure is associated with a page frame as long as the page frame is allocated to the `bcache` module.

Within the `pframe` structure, the `pf_bio` field points to a `bio` structure (or `NULL`, if none is associated with the page frame), and the `pf_use` field is a union containing a structure for each possible higher level use (e.g, `bcache`).

6.1.1. The `bio` Module

The work of actually performing I/O is the responsibility of various device drivers; the `bio` module merely provides a generic structure to represent a block I/O operation and some subroutines to handle allocation, deallocation, and synchronization.

The `bio` structure primarily consists of fields drawn from the `buf` structure of traditional UNIX (Bach [15], Leffler, *et al.* [134]):

- A union of free list and active list pointers. In our case, the former is a `poolitem` and the latter is just a pair of pointers; any such active list is a private concern of the device driver, and we assume highly parallel structures are not required.[132]
- Device and block number, specifying the target of a read or write operation.

[132] If this assumption were not valid, additional list item types would be added to the union, as described in section 3.7.1 on page 80.

- Pointer to the associated `pframe` structure (`bio_pf`).
- Transfer size and residual remaining after error.
- Function to call upon I/O completion.
- Flags to alter the behavior of I/O operations, e.g., read vs. write, whether or not the operation should be waited for, and whether the page frame should be deallocated (`pf_free` with `PF_FREE_REALLY`) on I/O completion.

In addition, the `bio` structure includes a reference count (`bio_rcnt`) to help control when the `bio` ↔ `pframe` relationship can be broken, a context-switching readers/writers lock (`bio_rw`) to synchronize access, and a context-switching event for awaiting I/O completion.

Functions provided by the `bio` module are listed in Table 32, below. Allocation and deallocation of `bio` structures is handled by the `bioget` and `biorelease` functions. The `bioreserve` function provides a way to ensure that the next request for a `bio` structure can be satisfied without blocking; a pointer to the reserved structure is held in a dedicated per-processor pointer, which is referenced preferentially by `bioget`.

Careful coordination is required in `bioget` and `biorelease` to avoid races and unnecessary serialization. For our target class of machines, we use a technique related to that of section 3.7.6. A hash table of busy-waiting readers/writers locks provides synchronization, but the table is static; we don't add or delete anything to it:

```
bwrwlock bio_hashrw[NBIOHASHRW];  // table of r/w locks
```

The `pf_bio` field within each `pframe` is the focus of `bioget`:

Function	*Purpose*
b `bioget(`*pf*`, ` *t*`, ` *q*`)`	Get `bio` structure for `pframe`
v `biorelease(`*b*`, ` *t*`)`	Release `bio` structure
s `bioreserve()`	Ensure next `bioget` will succeed
i `bioerror(`*pf*`)`	Check for I/O error
i `biowait(`*b*`)`	Wait for I/O completion
v `biodone(`*b*`)`	Called by drivers when I/O is done

s = `spl_t` (`bioreserve` does `splbio` as side effect; see section 3.4)
b = `struct bio *`
pf = `pframe *`
t = `int` (`BIO_NOLOCK`, `BIO_SHARED`, or `BIO_EXCL`)
q = `int` (boolean; *TRUE* = use "guaranteed" r/w lock function (§4.5.2/p183))
v = `void`
i = `int` (an `errno` value)

Table 32: `bio` Module Functions.

```
struct bio *
bioget (pframe *pf, int locktype, int qlock)
{
      bwrwlock *bh = &bio_hashrw[BIOHASH(pf)];
      spl_t s = bioreserve();
again:
      bwrw_rlock (bh, splcheckf(s));
      struct bio *bp = pf->pf_bio;
      if (bp != NULL) {
            // bio structure already allocated
            vfai(&bp->bio_rcnt);
            bwrw_runlock (bh);
            vsplx(s);
            lock bp->bio_rw according to locktype and qlock
      }
```

If a `bio` structure is already allocated, we simply register ourself by incrementing its reference count *before* releasing our hash bucket lock. The `locktype` and `qlock` parameters are used to control locking of the `bio` structure: if `locktype == BIO_NOLOCK`, no such lock will be obtained; we get a reader lock for `BIO_SHARED` and a writer lock for `BIO_EXCL`. When `qlock` is `TRUE`, we use a "quick" reader/writer lock function, if we need a lock (§3.5.4/p64).

```
      else if (bwrw_rtow (bh)) {
            // need allocation, exclusive access obtained
            bp = address of new bio structure;
            pf->pf_bio = bp;
            bp->bio_pf = pf;
            additional *bp initialization
            lock bp->bio_rw according to locktype (no blocking)
            bwrw_wunlock (bh);
            vsplx(s);
      }
```

The heart of the algorithm is the call to `bwrw_rtow`; when the upgrade is successful, we allocate a new `bio` structure (with `bio_rcnt` already equal to 1). When the upgrade fails, we start over:

```
      else {
            // hash/lock collision; retry
            bwrw_runlock (bh);
            goto again;
      }
      return bp;
}
```

Note that `bioreserve` is called at the beginning of `bioget` to protect the remainder of `bioget` from blocking (at the point indicated by `bp =` *address of new* `bio` *structure*). This strategy would not preclude `bioget`'s caller from holding a busy-waiting lock too, since `bioreserve` will not block if a `bio` structure is already reserved; such a caller of `bioget` would simply call `bioreserve` before obtaining its own busy-waiting lock.

Assuming the hash function (`BIOHASH`) and hash table size (`NBIOHASHRW`) are adequate, very little lock contention should result from random collisions. The major source of lock contention should be concurrent operations on the same page frame. As we saw in section 3.5.4, the readers/writers operations used here cost exactly one shared memory reference each, in the absence of lock contention.

The focus in `bioget` is on the pointer from a `pframe` to a `bio` structure; the focus in `biorelease` is on the reference count:

```
void
biorelease (struct bio *bp, int locktype)
{
      pframe *pf = bp->bio_pf;
      bwrwlock *bh = &bio_hashrw[BIOHASH(pf)];
      switch (locktype) {  // release lock of indicated type
      case BIO_NOLOCK: break;
      case BIO_SHARED: csrw_runlock (&bp->bio_rw); break;
      case BIO_EXCL:   csrw_wunlock (&bp->bio_rw); break;
      }

      spl_t s = splbio();
retry:
      bwrw_rlock (bh, splcheckf(s));
      if (fad (&bp->bio_rcnt) > 1) {
            // not last reference; done
            bwrw_runlock (bh);
            vsplx(s);
      }

      else {
            if (!bwrw_rtow (bh)) {
                  // lock contention; try again
                  vfai (&bp->bio_rcnt);
                  bwrw_runlock (bh);
                  (*splcheckf(s))();
                  goto retry;
            }
```

```
            // exclusive access obtained
            if (bp->bio_rcnt == 0)
                 pf->pf_bio = NULL;
            else
                 bp = NULL;
            bwrw_wunlock (bh);
            if (bp != NULL)
                 deallocate *bp;
            vsplx(s);
       }
  }
```

Proper use of `bioget` and `biorelease` guarantee a non-negative reference count, so `pf->pf_bio` will not change value even after releasing the lock and executing the `goto` statement.

For our target class of machines, we have designed `bioget` and `biorelease` so that the common cases of many concurrent operations on the same page frame can be executed with no serial bottleneck. Furthermore, the overhead of managing the `pframe` ↔ `bio` association is very low. The probability of unrelated lock contention is dependent on the quality of the hash function, which can be made perfect at a maximum space overhead of one `bwrwlock` structure per page frame.[133]

6.1.2. The `bcache` Module

The buffer cache implementation for our target class of machines is essentially nothing more than that outlined in section 3.7.6 on page 136. The `pframe` structure is augmented with the following additional information (§5.3.4/p242):

```
struct pf_use_bc {
     struct _pframe *next;      // hash bucket pointer
     dev_t dev;                 // device number
     daddr_t bno;               // disk block number
};
```

Because we are dealing with variable sized blocks, we impose some restrictions for simplicity:

- The blocks must be properly aligned, i.e., the starting block number (on the device) must be a multiple of the size (in basic device blocks). This restriction simplifies and improves the hash function.
- The use of overlapping blocks is not supported. This means that small blocks must be removed from the cache before a larger overlapping block is requested, and vice versa.

The functions of the `bcache` module are listed in Table 33, on the next page. In most cases, their functionality is essentially the same as the traditional UNIX kernel functions of the

[133] As shown in section 3.5.4 on page 61, a `bwrwlock` is as small as one integer, 4 or 8 bytes on most current machines.

Function	*Purpose*
rr getblk(*d*, *b*, *s*, *a*, *t*)	Find/allocate pframe by dev/block-number
re bread(*d*, *b*, *s*, *t*)	getblk and read if necessary
re breada(*d*, *b*, *s*, *rab*, *ras*, *t*)	bread plus read-ahead
v bwrite(*pf*, *t*)	Write, wait, and release
v bawrite(*pf*, *t*)	Write, don't wait, release when done
v bdwrite(*pf*, *t*)	Release and mark dirty
v brelse(*pf*, *t*)	Release with no I/O implied
i bcache_reassign(*pf*)	Called by pframe module for reassignment
i bcache_pf_flush(*pf*, *d*)	Called by pframe module for sync and umount
v babort(*d*, *b*, *s*)	Unmark dirty and free pframe
v bcache_biodone(*bp*)	Called by biodone on I/O completion

d = dev_t (device number)
b = daddr_t (block address on device)
s = int (size of block in bytes)
a = int (boolean; *TRUE* = allocate pframe if not found
t = int (lock type: BCACHE_NOLOCK, BCACHE_SHARED, or BCACHE_EXCL).
rr = struct reference_pf (pframe * and reference count before increment)
re = struct rtrn_pf (pframe * and errno value)
rab = daddr_t (read-ahead block address)
ras = int (read-ahead block size in bytes)
pf = pframe *
bp = struct bio *
v = void
i = int

Table 33: bcache Module Functions.

same name (Bach [15]).

Page frame reference counts are incremented by getblk, bread, and breada, and decremented by bwrite, bawrite, bdwrite, and brelse. Interaction with the page frame module's free list is managed by the bcache_reassign and bcache_pf_flush functions (§5.3.4/p241, §5.3.4/p242). I/O required by bread, bwrite, etc., is initiated by calling the bio module (§6.1.1) and the relevant device driver, with essentially the same interface as in traditional UNIX kernels (Bach [15]).

6.2. File Systems

The most common, and hence most performance-critical, file system operations are open, close, stat, read, and write for ordinary files, especially when physical I/O can be avoided due to cacheing in memory. There are, however, other operations that must not be neglected; these include *block* and *character* device operations, and unusual operations such as mount, umount, and sync.

6.2.1. Mountable File Systems

In a UNIX system, the overall file system is generally constructed by joining independent mountable file systems together with the `mount` system call. Many systems support more than one file system type, including remote as well as local ones, but we will focus here on general problems of local file systems only. Many of the same issues and solutions are applicable to remote file systems too.

The evaluation of file path names relies on a data structure describing mounted file system relationships; in traditional kernel implementations, this is the *mount table* (Bach [15]). Usage of the `mount` system call is rare, usually limited to system start-up time, so a single readers/writers lock (the *mount lock*) is sufficient for synchronization; only `mount` and `umount` require exclusive access to the mount table (the readers/writers lock algorithm we use (§4.5.2) has the important property of not serializing readers in the absence of writers). The simplest strategy, used in Symunix, is to obtain and hold the reader lock for the duration of each path name resolution, but this is feasible only if we assume path name resolution time is always short, as in systems with only local file systems.[134] An alternative is to obtain and release the mount lock separately for each path name component resolved; this increases overhead somewhat.

6.2.2. File System Resources

Certain data or resources are maintained on a per-file-system basis and thus require appropriate mechanisms to synchronize their use. The details depend on the internal structures of each particular file system, but examples include free space, usage quota information, flags, modification times, and transaction logs. All the techniques outlined in chapter 3 are available to implement the needed synchronization.

Free Space

Symunix-1 has a file system based closely on that of 7th Edition UNIX. In this design, the list of free disk blocks is maintained with some block numbers kept in the *superblock*, some more stored within the last block referenced in the superblock, and so on. The synchronization method chosen for Symunix-1 was simple mutual exclusion, on a per-file-system basis, for any allocation or deallocation request. This solution clearly does not scale upwards to systems in our target class. A straightforward improvement to avoid bottlenecks could be based on group lock (a context-switching variety, similar to that to be described in section 7.3.5 on page 304), but other considerations, such as the desire to perform more careful block placement, favor alternative organizations for Symunix-2.

An in-memory bitmap can be implemented with a variation of the compact set data structure presented in section 3.7.5 on page 126. The algorithm can be augmented to support a form of interior removal, thus allowing for preferred block placement; when this is unsuccessful, the set algorithm's `getarb` function can be used to obtain a nearly random placement.

[134] Times are short in practice, but computing a bound would be difficult and the result would be quite large.

There are many possible ways of handling variable size block allocation. The bitmap-based set algorithm scheme just mentioned allows the bitmap to be examined directly, so it is possible to scan the status of blocks near a particular location. Fetch&And allows multiple contiguous blocks (up to the number of bits in a word) to be allocated atomically by setting a string of bits to zero. Of course concurrent operations may interfere with one another, causing one or more processes to retry, possibly elsewhere. The algorithm can also be further augmented to allow faster location of bitmap words containing sufficiently many contiguous set bits.

Another general approach is to implement a buddy system for allocation of free disk space. Memory consumption is a problem with the type of algorithm presented in section 5.3.4 on page 244; a list item structure is required for each potential minimum-sized block. (Memory consumption is also a problem with bitmap-based solutions, but the constant factors are different: one, or even several, bits per block compared to several words per block.) Since only a very unusual allocation pattern would actually make use of all those structures, it makes sense that many fewer will be needed in practice.

One way to reduce physical memory usage for the free space allocator is to use virtual memory, with a portion of the device itself being used for backing storage. Another possibility is to manage the list items for blocks dynamically (allocating one on every block split, freeing one on every merge). This can be done using a pool (§3.7.1/p82) together with a search structure (§3.7.6), using the block number and size as key, but some strategy must be chosen for handling failure to allocate a list item structure when splitting a block. A simple approach is to discard the free half of such a block, keeping track of it in an on-disk list (stored within the otherwise "lost" blocks).

Free Inodes

In Symunix-1, being based on 7th Edition UNIX, a small number of free i-node numbers are kept in the superblock. When this runs out, linear scan of the on-disk i-nodes is performed to replenish it. As with free space (p258), we use simple mutual exclusion to ensure correctness, but clearly this solution doesn't scale to meet the needs of our target class of machines. Most of the solutions outlined for free space allocation in Symunix-2 are applicable to i-nodes as well. Even the issue of variable-size block allocation can be relevant for some file system designs, where i-nodes are intermixed with data blocks and adjacent placement is attempted.

6.2.3. The `file` Structure

The main purpose of a `file` structure in UNIX is to hold the *current seek position* for an open file. Traditional UNIX semantics require all file descriptors derived from a single `open` system call to refer to the same position for `read`, `write`, and `lseek` system calls; such file descriptors may be derived by `dup` or by inheritance over `fork`. Because of these semantics, the seek position cannot be stored as a per-process attribute, such as the file descriptor itself, or as a per-open-file attribute, such as the file permissions. There is a many-to-one relationship between file descriptors and file structures, and between file structures and i-nodes.

The role of the file structure has changed very little since the early days of UNIX (Ritchie [169]). The most significant change we impose is using a pool (§3.7.1/p82) for allocation instead of linearly scanning the table of file structures or using a simple linked list. Each file structure contains a reference count to determine its own deallocation when no more file descriptors refer to it.

It is important to realize that the seek position stored within a file structure may be modified concurrently by more than one process executing `read`, `write`, or `lseek` system calls. The effect of such concurrent operations is not clearly defined in traditional UNIX implementations. (The selection of which bytes are transferred to and from which file positions by each involved process and the final value of the seek position are not necessarily sequentially consistent.) Symunix-1 imposes sequential consistency by serializing all `read` and `write` system calls at their file structures;[135] this limits parallelism for `read` and `write` calls only when they operate on file descriptors derived from a common `open`. A more careful implementation could eliminate most of this serialization using a readers/writers lock or a group lock, but an attractive alternative is to fall back upon the traditional inconsistent semantics, expecting the user to provide any necessary synchronization. This is the approach taken in Symunix-2. Note also that the meta system call mechanism in Symunix-2 allows `read` and `write` to be performed without depending on a previous seek (§4.3.1/p155), so it is possible to avoid the problem completely. In this case, as an additional optimization, one might like to defer the overhead of allocating the file structure until its seek position is actually needed. However, considering the low cost of two pool operations (as little as 17 shared memory references; Table 14, §3.7.1/p83), the "optimization" may not be worthwhile. Eliminating the file structure from the kernel altogether, in favor of a user-mode implementation of shared seek positions using shared memory (invisible to the applications programmer), is philosophically attractive but likely slower (and therefore hard to justify, given the entrenched nature of the existing semantics).

6.2.4. Inode Access

The search structures of section 3.7.6 (without LRU cacheing) were originally developed to manage in-memory i-nodes for Symunix-1. Given a device and i-number, the algorithms locate or allocate an i-node structure in memory; likewise they automatically remove such structures when no longer needed.

Once an in-memory i-node is obtained, further access and manipulation must be synchronized using other mechanisms.[136] The most common types of access and manipulation are `read` and `write`, but there are others as well. We perform most synchronization with a context-switching readers/writers lock in each in-memory i-node structure; Table 34, on the next page, lists the Symunix-2 operations requiring synchronization at the i-node and the type of lock used for each (shared or exclusive). Note that, of those operations requiring an exclusive lock, only directory update would normally be used with high frequency and, hence, represent a potential serial bottleneck.

The need for an exclusive lock during directory update stems from established UNIX semantics: file names within a directory must be unique. This means that whenever a file name is created, the system must make sure that no other entry within the same directory has the same name (in fact, normal behavior in UNIX, as provided by the traditional `creat` system call or the 3-argument form of `open` with the `O_CREAT` flag, is to truncate and open

---

[135] For some inexplicable reason, `lseek` calls are not also serialized.

[136] Actually, as described in section 3.7.6 on page 130, additional synchronization begins at the point within the search algorithm called "*final acquisition steps*".

Operation	*Locking*
`access`	shared
`chdir`, `chroot`	shared
`chmod`, `fchmod`	exclusive
`chown`, `fchown`	exclusive
`close`	shared †
directory look-up	shared
directory update	exclusive +
`exec`	shared
`flock`	exclusive
`fsync`	shared
`link`	shared
`mapctl`	shared
`mapin`	shared
`mapout`	shared
`open`	shared *
`read`	shared
`stat`	shared
`truncate`, `ftruncate`	exclusive
`unlink`	shared
`utimes`	exclusive
`write`	shared •

† `close` requires an exclusive lock when it is the last close (§8.3.2/p349).
+ directory update requires only a shared lock when removing an entry (e.g., `unlink`, `rmdir`).
* `open` requires an exclusive lock when truncating a file.
• `write` requires an exclusive lock when extending a file or filling a hole.

Table 34: Inode Locking in Symunix-2.

These operations affect ordinary files, directories, or symbolic links. Special device files may have other requirements met by the associated driver, line discipline, etc. Operations not listed require no i-node locks at all.

any such pre-existing file in preference to creating a new file or failing). When updating a directory for another purpose besides file creation, e.g., `unlink`, `rmdir`, some forms of `rename`, or when creating an *anonymous* file in Symunix-2 (§5.2.7/p221), there is no such requirement and a shared lock may often be used instead (just as when writing to an ordinary file; see the next page). Still, the exclusive access strategy for file name creation is a potential bottleneck, and could affect some workloads. Avoiding it would require implementing directories with a more elaborate data structure than the traditional unordered list and employing a parallel update strategy.

A significant difference between Symunix-1 and Symunix-2 is that the older system uses an exclusive lock for all file writes. Such an approach is not a problem for most serial applications, as it only limits parallelism for operations on a single file. On the other hand, it is quite reasonable to expect highly parallel applications to operate with a high degree of parallelism on a single file, in which case the exclusive access approach could be a serious limitation. For this reason, Symunix-2 uses a shared i-node lock for most cases of file writing. To simplify the implementation, an exclusive lock is still required whenever a file write requires allocation of a new file system block. Block allocation generally occurs only when appending to a file, but can also result from filling in a hole.[137]

We consider it unlikely that hole-filling will ever be a serial bottleneck, at least not unless the system is extended with features to support and encourage more extensive use of holes. Appending data to a file, on the other hand, could conceivably be a problem for some workloads. Note, however, that parallel applications can organize their I/O to avoid concurrent file appending; some reasonable workarounds include:

- Preallocate file space. Sometimes the total amount of file space needed is known in advance.
- Use buffering to ensure that all file appends are large; this makes it easier to take advantage of techniques such as *stripping*, increasing parallelism even for serial data streams.
- Have each process or thread within the application append to a different file, and possibly merge files at the ends of major computations.

6.2.5. Path Name Resolution

When converting a file path name, such as `/usr/gnu/lib/groff/tmac/tmac.etxr`, into an i-node reference, each of the specified directories is searched in turn, beginning with the root directory, `/` (or the current directory, if the specified path did not begin with `/`). Each directory is locked with a shared lock to prevent changes while it is being searched. As each successive path name component is matched, the associated i-node must be located, brought into memory, and locked. The path is effectively expanded by mount points and symbolic links. The target i-node is locked for shared or exclusive access as indicated in Table 34. To avoid deadlock, each directory i-node is unlocked before the next (child) i-node is locked.[138]

It is the nature of the search algorithm presented in section 3.7.6 to increment the reference count of the object (in this case, i-node) located or allocated; this fact insures that a successor in the path cannot be deallocated between the time the predecessor is unlocked and the successor can be locked. This strategy of not holding more than one lock during path resolution can result in non-serializable results when looking at whole system calls, i.e., each `open` is not atomic, a property shared with all traditional UNIX implementations. It is possible to concoct a case where a file may be opened under a path name that it never had; Figure 10, on the next page, shows such an example. Note, however, that the undesired result is already possible in serial UNIX, doesn't violate system security or integrity, and is simple to avoid (i.e., don't rename directories in situations where concurrent access is possible).

---

[137] In traditional UNIX systems, a "hole" is created in a file by seeking beyond the end of the file and writing. A hole can be read, and appears to contain zeros, even though it has never been written and occupies no disk space. Writing to a hole fills it in, resulting in disk space being allocated.

[138] There is an exception to this "lock only one i-node at a time" rule for resolving file path names: as explained in section 3.7.6 on page 130, an i-node must be locked while the in-memory copy is initialized. This cannot cause deadlock as the i-node is newly allocated in memory, and the search algorithm guarantees that the lock can be obtained without blocking.

> Assume the following files exist: `/a/b/c/d`, containing the words "file d", and `/a/b/c/e`, containing the words "file e". Now consider what might happen when the two processes perform the following two procedures concurrently:
>
> | **Process 1** | **Process 2** |
> |---|---|
> | | `rename /a to /aa` |
> | `open /a/b/c/d` | `rename /aa/b/c/d to /aa/b/c/dd` |
> | | `rename /aa/b/c/e to /aa/b/c/d` |
>
> I.e., the three `renames` all overlap the execution of `open`. There are three possible outcomes: The `open` can access the file containing "file d", the file containing "file e", or can fail to access any file. If `open` was atomic, the "file e" result would not be possible.

Figure 10: Example Effect of Non-Atomic `open`.

6.2.6. Block Look-Up

A critical step in any attempt to access the data in a file is to translate from a logical position within the file to a physical location on a secondary storage device.[139] Because secondary storage devices typically support access to data in units of some *block size* only, this translation itself consists of two steps: determining the logical block involved (and the offset within it) and translating the logical block to a physical one. This latter translation is a key design element of the file system implementation. Most traditional UNIX file systems support a scheme based on that of seventh Edition UNIX (Thompson [196]), possibly extended to allow use of a smaller block size for the tail end of a file (e.g., as in the Berkeley "fast file system" described by McKusick *et al.* [146]). There are, however, many other possibilities.

Once the desired physical block is identified, a buffer for it must be located or allocated in memory;[140] this can be done by making use of the buffer cache (§6.1.2), or by some other

[139] In most systems, more than one mountable file system may share a physical storage device, such as a disk. In addition, many systems also have the ability for a single file system to span multiple devices. For reliability or improved availability, some systems support replication as well. All of these realistic features are usually handled by first translating from a position within the file to an offset from the beginning of the logical file system, and then later applying additional translations to account for these additional mappings. For simplicity of presentation, but without loss of generality, we assume each file system occupies exactly one disk device.

[140] This is not strictly required in all cases; the `read` and `write` system calls always specify an address within the user's address space, which could be used directly if it meets all system-specific requirements for size and alignment. Doing so has the advantage of avoiding the cost of copying or performing page-level remapping tricks, but only at the expense of extra I/O operations that might be caused by the lack of cacheing, read-ahead, and delayed writes. A sophisticated approach might use this technique (possibly along with page-mapping tricks to allow some read-ahead and delayed writing), especially if there is reason to believe no sharing or re-use will occur. Evaluation of such schemes is beyond the scope of this document.

mechanism specific to a file system's implementation (see this page).

In Symunix-2, the block look-up function is performed by the `i2pf` function:

```
struct rtrn_ipf
i2pf (struct inode *ip, off_t off, off_t size, int mode)
```

Given a file, indicated by *ip*, an offset within it, indicated by *off*, together with *size*, and *mode*, `i2pf` locates or allocates a block (or blocks) on the storage device, locates or allocates a page frame, and reads the data if required. The *mode* parameter is constructed by the bitwise OR of the following constants:

- `I2PF_READ` The data will be read, so I/O will be needed if the page isn't already in memory.
- `I2PF_WRITE` The data will be written, so the old data need not be read from the device if the size and alignment are sufficient to fully cover the entire page frame.
- `I2PF_SEQ` Subsequent access will be sequential, so read-ahead might be helpful.

Exactly one of `I2PF_READ` or `I2PF_WRITE` must be used.

The return value of `i2pf` is a structure containing an `errno` value (0 for success), a pointer to the page frame (`NULL` for failure), and the offset into the file corresponding to the beginning of the page frame. From this the caller may deduce the desired position within the page frame. (Recall from section 5.3.4 on page 241 that `pf_getpaddr` may be called to determine the starting physical address of a page frame.)

A pointer for only a single page frame is returned, containing at least the initial portion of the specified data. The caller of `i2pf` must call `pf_getsize`, described in section 5.3.4 on page 241, to determine the true extent of the data available in the page frame.

The functionality of `i2pf` combines the effects of the traditional UNIX kernel functions `bmap` and `bread`/`getblk` (and related functions, as described by Bach [15]). The advantage of `i2pf` is that it gives flexibility to the file system implementation. The additional flexibility comes from calling a file-system-specific function to do most of the work, and potentially manifests itself in two ways:

(1) The file system implementation may allocate variable-sized extents on the storage device, and may use variable-sized page frames, if available.

(2) The file system is not forced to use the general buffer cache mechanism. For example, it may be advantageous to build a block look-up data structure directly associated with each open i-node; this may involve only indexing rather than searching, thus reducing overhead. Taking this approach, however, makes simultaneous consistent accesses via the traditional UNIX buffered device interface impossible (but the only practical impact of this is probably the inability to run `fsck` on mounted file systems, in our view a mess we should be glad to be rid of).

6.3. Device Drivers

Device drivers in Symunix follow the UNIX model, by being a collection of routines in the kernel that are called to perform various standard functions. A driver can usually be divided into two interacting parts:

- The *top half* is called by processes (activities, in Symunix-2), in response to specific needs, such as opening, closing, reading, writing, or performing control functions. The top half executes in the context of the process for which the operation is being performed.

- The *bottom half* is called by interrupt handlers to take care of device needs, such as completion of a requested operation, arrival of a character (or message, etc.), or occurrence of some kind of fault. The bottom half, like all interrupt handlers, executes in no particular process context.

Of course, there may be some functions in common between the top and bottom, and data structures are often maintained and accessed by both. On a uniprocessor, synchronization between the top and bottom is solved by judicious use of interrupt masking. On a multiprocessor, it may be that the top and bottom can execute on different processors, so additional synchronization is necessary. If the processor to handle an interrupt is predictable (e.g., always the same processor or always the processor with a physical connection to the physical device in question), a simple technique is to restrict the top half to run only on that processor. Some schedulers maintain per-processor ready lists, which can be used for this purpose. Alternatively, as in Symunix, context-switching locks (§4.5) can be used to provide an appropriate level of mutual exclusion for the top half, and busy-waiting locks (§3.5) can provide it for the bottom half. Whenever the top must exclude the bottom, it must use both interrupt masking and busy-waiting.

Even on a uniprocessor, it is often necessary for multiple processes to synchronize. Traditional UNIX kernels use the `sleep`/`wakeup` mechanism for context-switching synchronization; section 4.5 describes the approach we use in Symunix. There is a special consideration for device drivers that applies equally to Symunix as it does to traditional kernels: because the bottom half doesn't execute in the context of any particular process, context-switching synchronization is inappropriate there. This is largely in the nature of interrupt handling.

Per-Processor Devices

On some machines, physical devices are attached to specific processors in such a way that other processors have no direct access. The asynchronous procedure call mechanism, described in section 3.6, can be used to solve this problem. (In Symunix-1, the traditional kernel `timeout` function is used for this purpose, with a time delay of 0. Our version of `timeout` takes parameters to specify the processor, time delay, function, and argument.) An alternative method of accessing per-processor hardware is to use per-processor ready lists, as already mentioned, if the scheduler provides them.

Soft Interrupts

An important issue in I/O performance is interrupt latency. While hardware factors are also present, the most significant contributor to interrupt latency is usually software masking of interrupts (§3.4). An important technique in Symunix is to use *soft interrupts* (§3.4/p46); these are just like regular interrupts, but are initiated directly by software instead of by hardware. A typical Symunix device driver has both hard and soft interrupt handlers. The hard interrupt handler is very short, taking care of the most time critical device functions, such as copying data between device-specific and in-memory buffers, and scheduling a soft interrupt (`setsoft`*t*, in section 3.4 on page 45). The soft interrupt handler, which executes only when no hard interrupt is pending or masked on the same processor, takes care of more time consuming functions, including waking up the top half if necessary. On most occasions when the top half needs to block interrupts, only the *soft* interrupts must be blocked, thus reducing hard interrupt latency. In some cases a non-driver portion of the kernel, such as generic tty or file system support, must also block interrupts; these are also soft.

As a result of this soft interrupt strategy, Symunix-1 on the Ultra-2 prototype is able to sustain more than one[141] instance of `uucp` running on 9600 bps serial lines with a very low rate of character overruns, even though the hardware lacks character FIFOs; this was not achievable on many contemporary commercial machines (many machines today can only do it by virtue of hardware FIFOs).

6.4. Future Work

Copy-On-Write for On-Disk Data

Copy-on-write optimizations have long been used to provide the semantics of copying with reduced cost (Bobrow *et al.* [36]). Generally copy-on-write is performed for in-memory pages of a virtual memory system. We wish to consider the possible impact and advantages of performing these optimizations for on-disk data as well.[142]

Perhaps the reason copy-on-write has not been more seriously considered for on-disk file data is the lack of a *copy* system call for files in most traditional systems, e.g., UNIX: without a copy operation, there is nothing to optimize. Symunix-2, however, has two such operations: they are called `read` and `write`, and result from the memory model in which all user-accessible memory is established by mapping files into address spaces (this model is the subject of section 5.2). Every `read` is a copy from the indicated source file to the one or more *image files* mapped at the specified buffer range. Every `write` is a copy from the image file(s) mapped at the specified buffer address range to the specified target file. If the files all reside on the same mountable file system and the copy operations are suitably sized and aligned, an opportunity for copy-on-write optimizations exists, both in memory and on disk. In this section we consider only the latter possibility because of its relative novelty.

Allowing for copy-on-write at the disk file level has many implications, but one is especially significant: the natural assumption that a disk block is associated with at most one position in one file is no longer valid. Additional on-disk data structures must be maintained by the file system to keep track of reference counts for each block. The simplest, but by no means only, possibility is a simple array of counts, one for each block in the file system.

It will be simplest if only ordinary file data blocks are subject to copy-on-write; since it seems unlikely that this optimization will apply to i-nodes, directories, indirect blocks, etc. We also suggest reserving sufficient disk space to hold all copied blocks; otherwise a `read` or `write` might fail due to an out-of-space condition even if the target file space was preallocated.

6.5. Chapter Summary

In this chapter, we looked at how we can avoid serial bottlenecks in the input/output portion of the kernel. We are concerned not so much with the architectural issues of scalable I/O bandwidth, as the data structure issues involved in managing that I/O, caching the resulting

---

[141] We never tried more than two in a rigorous way.

[142] Here we consider only file system issues, because Symunix-2 doesn't use any non-file disk space, e.g., "swap space" dedicated as backing storage for virtual memory.

information, and synchronizing access to it.

We began by separating the the block I/O system into strictly input/output and buffer cache management portions. While some other OS designs have sought to have the file system subsumed by the virtual memory system, we have in a sense gone the other way: our VM system depends undeniably on the file system, for which we allow a variety of possible implementations.

We then focused on file systems and how to implement them without unacceptable serialization. Sometimes traditional UNIX semantics require us to jump through hoops, and sometimes they make our job easier. Our solution involves several facets:

- Use of a readers/writers lock for the global mount table.
- Management of free file system space based on a set algorithm or the buddy system, both of which are highly parallel.
- Use of a parallel search structure to organize in-memory i-nodes. This algorithm is structured in such a way (with i-node locks and reference counts) that it also forms an integral part of the path name search algorithm.
- We discussed several alternatives for implementing the traditional UNIX seek position, with varying degrees of serialization, none of them severely limiting for highly parallel applications.
- Synchronization of most file or directory accesses based on a per-i-node readers/writers lock. Most common operations require only the non-exclusive form of this lock. We identified certain directory update operations as being potential bottlenecks; careful workload-based study on large machines running these algorithms will be required to determine the seriousness of any such bottleneck.

Chapter 7: User-Mode Facilities

Other chapters have focused on the operating system itself, the parts that an unprivileged programmer cannot change or avoid. Only occasionally have there been references to the role of unprivileged code or how it might work. In this chapter, we try to balance the presentation and fill in those gaps. The facilities we cover fall into two broad categories:

(1) Basic support for parallel programming.

(2) The migration of functionality out of the kernel.

Moving things out of the kernel has long been an important theme of operating systems research, but attention is usually focused on using server processes to provide traditional OS functionality. In this chapter we address a less glamorous form of migration from kernel to user-mode: increasing reliance on language run-time support code and other libraries that simply build higher level facilities by using low level primitives provided by the kernel. Our approach is driven by performance and user flexibility, rather than minimizing kernel size or maximizing portability.

We present a "softer" design in this chapter, less specific than in the rest of this dissertation. This is because one of our positions is that user-mode facilities must be customizable to meet the needs of diverse programming environments. We envision a wide range of solutions, because there is no single best one.

Table 35, below, gives a guide to our treatment of many important primitives for user as well as kernel-mode use. Some of the basic primitives described for kernel use can be used in

Primitive	*Kernel-mode*	*User-mode*
Fetch&Φ	§3.1	§3.1
Test-Decrement-Retest	§3.2	§3.2
histograms	§3.3	§3.3
interrupt masking	§3.4	§4.8.1
busy-waiting synch.	§3.5	§7.2
interprocessor interrupts	§3.6	§4.8
lists	§3.7	§7.4
context-switching synch.	§4.5	§7.3

Table 35: Primitives for Kernel- and User-Mode Execution.

user-mode with no modification at all, while some require minor adjustment. In addition, some other primitives can only be made available for user-mode execution in a way that differs significantly from their closest kernel-mode relatives.

After describing a minimalist approach to parallel programming environments in section 7.1, we address user-mode issues in synchronization and list algorithms in sections 7.2, 7.3, and 7.4. Section 7.5 is devoted to the field of thread management and 7.6 deals with memory management issues. Finally, we summarize the chapter in section 7.7.

7.1. Basic Programming Environment

In this section we describe how to build a crude but useful programming environment by making a minor extension to the C programming language and providing a library of routines for other functions. This is similar to the design that was implemented for Symunix-1. A similar approach has been used with FORTRAN. In sections 7.5 and 7.6 we describe more sophisticated C-based parallel programming environments, but this crude environment is illustrative of the basic techniques employed and is actually useful, in some cases, even if more sophisticated environments are also available.

A new keyword, `shared`, implements a *storage class modifier* for statically-allocated variables; it causes them to be stored in a memory segment to be shared among all cooperating processes (the processes that make up the *family*–see section 4.2 on page 148). Because these shared variables have static allocation and lifetime, they are initialized only once, at program startup. For example, if

```
shared int a, b[10], *c;
shared struct stat d;
```

appears outside any function, it declares four variables with static allocation and global scope: an integer (`a`), an array of ten integers (`b`), a pointer to an integer (`c`), and a `stat` structure (`d`). This declaration would be illegal inside a function, where the storage class would be `automatic`, but

```
shared static int e;
```

works because it specifies the `static` storage class.

Variables not declared `shared` are private; they are initialized at program startup if they have static allocation, and they are copied to a new child process, just as ordinary variables are in a serial C program that does a traditional UNIX fork.

Memory consistency can be left up to the hardware. In the case of the Ultracomputer prototypes, cacheability is under software control. Symunix-1 provides an address alias for the shared segment in memory: two virtual addresses will access each shared memory location available to the user, one is cacheable and the other isn't. The hardware provides two ways to invalidate the cache: one for the whole cache, and one for a single word; both are directly executable from user-mode and both execute quickly.[143] Symunix-2 has a much more

---

[143] In both Ultra-2 and Ultra-3, the cache tag chips are cleared in a single cycle for whole cache invalidation. Invalidating a single line is a side-effect of performing a Fetch&Add with a cacheable address; a single `vfaa(addr,0)` will do the job, without side effects. On Ultra-3, the `vfaa` doesn't stall the processor unless *AFENCE* mode is in effect.

flexible memory model, which also allows such cacheability aliases to be established.

The Ultra-3 prototype also has the ability to perform memory references out of order. It offers a choice of three *fence modes*:[144]

- *AFENCE*. Wait for all previous data references to complete before issuing another.
- *NCFENCE*. Wait for all previous non-cacheable data references to complete before issuing another.
- *NOFENCE*. Issue all references immediately (subject to hardware resource constraints).

Ultra-3 also supports two *fence operations* to await the completion of either all data references or all non-cacheable data references.

These cacheability and access ordering features were designed into the hardware so the software could implement whatever consistency model is desired.

Everything else necessary for parallel programs can be done without changes to the language itself. ANSI standard C [5] includes the `volatile` storage class modifier, which is (barely) adequate to specify access ordering constraints for shared variables. Library functions (or macros) can easily implement the Fetch&Φ functions of section 3.1. Parallelism can be directly controlled by the user with the facilities of section 4.2, such as `spawn`, `wait`, and the signals `SIGCHLD` and `SIGPARENT`. A minimal set of primitives is not difficult to implement for user-mode use.

7.2. Busy-Waiting Synchronization

Section 3.5 described the basic busy-waiting synchronization mechanisms used in the kernel: simple delay loops, binary semaphores, counting semaphores, readers/writers locks, readers/readers locks, and group locks. The situation isn't much different when operating in user-mode. The same set of basic mechanisms are just as useful, and the same implementation techniques are just as relevant. However, it is reasonable for the selection of available mechanisms to be somewhat richer in a user-mode library, and there are some implementation details that must differ between kernel- and user-modes. These differences are discussed in the remainder of this section.

7.2.1. Interrupts and Signals

Because the kernel is endowed with certain hardware privileges, it can mask hardware interrupts at will (and generally at low cost). This is important not only because unmasked interrupts can potentially cause deadlock in combination with busy-waiting, but also because busy-waiting with interrupts masked tends to increase interrupt latency. Outside the kernel, interrupts are available only in a more abstract form: signals.[145] Signals can be blocked in user-mode in much the same way as interrupts can be blocked in kernel-mode; the mechanism described in section 4.8.1 allows it to be done quite cheaply. *Check functions* (§3.5/p50)

[144] The IBM RP3 [162] appears to have been the first to introduce the "fence" term and has similar modes.

[145] Some systems provide alternate interrupt mechanisms for user-mode programs, designed to have lower overhead and latency. As long as the cost of masking interrupts in user-mode is fairly low, everything we say here about signals is applicable.

can reduce signal latency caused by busy-waiting when signals are masked, but many applications are relatively insensitive to signal latency, so their importance may be correspondingly lessened in user-mode.

Even though hardware interrupts are not directly visible to user-mode programs, they can still cause significant random delays in program execution. This is because of the time taken to execute interrupt handlers in the kernel (even tiny ones). The occurrence of such delays while locks are held can aggravate lock contention. *Non-blocking* and *wait-free* algorithms [105] solve this and other problems, but don't necessarily provide scalability (e.g., the time for N processes to perform an operation concurrently is generally linear in N). A possible hardware solution would be to provide *temporary interrupt masking* in the hardware, designed to do what temporary non-preemption does (§4.6.4), but only for extremely short critical sections (§4.9/p205).

7.2.2. Preemption

Preemption, the rescheduling of a processor to another activity, can be another source of unpredictable delays. The delay can be orders of magnitude greater than that caused by interrupt handlers. Symunix-2 provides scheduling groups with several alternate policies (§4.6.3), including a non-preemptive policy that can let an application avoid preemption completely, but its use is necessarily restricted. Temporary non-preemption (§4.6.4) can help in many cases, and has no restrictions other than low maximum duration.

Temporary non-preemption should be invoked whenever working with busy-waiting locks in user-mode. Section 8.4 gives some performance results for this kind of usage within user-mode queue routines.

7.2.3. Page Faults

We have assumed the kernel is fully memory resident, but this is not generally the case for user-mode programs.[146] Page faults are similar to processor preemption, in that both are generally unpredictable. The unpredictability is exacerbated by systems that don't provide user code with much information about page residence status or page eviction policies and by programming languages and tools that provide no simple way to control the assignment of variables to pages. But, most importantly, it is generally regarded as a virtue that programmers not concern themselves with such details.

A page fault that occurs while holding a lock can seriously hurt performance. Moreover, an asynchronous page fault will cause deadlock if a lock needed by the `SIGPAGE` handler is already held by the same process. There are two approaches that can help: avoiding page faults, and lessening their impact.

Avoiding Page Faults

Many systems with virtual memory provide a mechanism to lock or "wire down" parts of memory, making them ineligible for eviction. Assuming that ordinary preemption has been

[146] It isn't even always true for kernels. Several commercially available operating systems, including IBM's AIX, include the feature of demand paging for parts of the kernel itself. In addition, most micro-kernel systems have the ability to demand page most of the server software.

addressed, as discussed in section 7.2.2, it is only necessary to wire down the pages to be touched while holding a busy-waiting lock. At least in principle, these pages could be "unwired" again when the lock is released, or perhaps when the process is not running. There are, however, several problems that limit this approach:

- *Identifying which pages to wire down*. Simply identifying the pages containing the instructions that will be executed is difficult. It is not conceptually hard to do a conservative job, and pathological cases involving indirect function calls don't often arise in practical code with locks, but programming languages and environments don't provide tools to do so. Identifying all the data pages that might be touched is even harder. Pointers may need to be followed, and data objects can be stored in statically or dynamically allocated memory. It is often not possible to identify a small set of pages before obtaining the lock, and a conservative approach would require wiring down vast sections of the address space, possibly more than the physical memory can accommodate.
- *Cost*. The cost of making frequent changes to the list of wired down pages may be high, compared with the expected cost of locking and data manipulation.
- *Availability*. On some operating systems, the mechanism for wiring down pages may be missing, privileged, or otherwise limited in applicability.

Nevertheless, wiring pages into memory offers the potential of completely avoiding page faults when holding busy-waiting locks. Appropriate software design can reduce the above-mentioned problems.

Lessening Page Fault Impact

The first step is to avoid deadlock in asynchronous page faults. This is easily accomplished by using the low-overhead method of masking signals described in section 4.8.1; when a page fault occurs and the `SIGPAGE` signal is blocked, the fault is handled synchronously, without user-level intervention.

The remaining problem is to avoid excessive busy-waiting when a page fault occurs while a lock is held. There are several possible approaches:

- *Non-blocking* and *wait-free* algorithms are well suited for this situation. Such algorithms have the property that other processors need not wait indefinitely for a stalled one, even when it is executing critical code. On the other hand, these algorithms are not bottleneck-free, so an operation may take O(p) time for p processors.
- A hybrid of busy-waiting and context-switching synchronization[147] can be used. With this form of synchronization applied to locking, a processor that cannot immediately obtain a lock will busy-wait for a short amount of time, and then perform a context-switch if still not able to proceed. This technique puts a bound on busy-waiting time. Anderson used this scheme in conjunction with "scheduler activations" [7].
- With kernel assistance, and/or knowledge of the machine's MMU structure, it is possible to guarantee that a small number of pages remain memory resident for a short time (say, the duration of a temporary non-preemption period). The pages needed can be identified and referenced immediately after acquiring a busy-waiting lock. If any of them are not memory resident, a special `SIGPAGE` handler can release the lock and cause the sequence

[147] Ousterhout called this "two-phase blocking" [160].

to begin again after the fault is serviced. This kind of approach is similar to the non-blocking algorithms, in that the critical code is organized in a way similar to a transaction, but the final "commit" need not be atomic. Of course, the theoretical advantages of non-blocking algorithms are lost, and there are many machine-dependent details.

The biggest problem with all these solutions is that they can't magically service the page fault instantly. If a missing page is needed, for example, to complete a user-mode context-switch, then no such context-switches will be done until the page is brought into memory again. This is why we put so much emphasis on avoiding such page faults by selective wiring down of pages.

7.2.4. The Problem of Barriers

Barriers have not fit in well with the Symunix kernel structure, but have a history of use in general parallel programming. We implemented barriers in a library on Symunix-1, based on an algorithm of Dimitrovsky [62, 65]. A barrier data structure is declared as `bwbarrier_t` *b*, and initialized before use by calling `bwbarrier_init(`*bp*`,`*nump*`)`, where *bp* is a pointer to a barrier structure and *nump* is the number of processes necessary to complete synchronization. Each process calling `bwbarrier(`*bp*`)` is delayed, if necessary, until the *nump*th has also called `bwbarrier`. The algorithm used has the property that `bwbarrier` may safely be called in a loop (there is no danger of a fast process getting around the loop before a slow one gets going again after the barrier).

Barriers are not locks, so it is not normally appropriate to mask interrupts, signals, or preemption; there is nothing small to mask them *around*, as there often is with a lock. A typical situation is a loop with one or more barriers in it, and a total execution time too long to tolerate masking interrupts, but an inter-barrier time very short compared to the preemption interval. The combination of barriers and preemption (or any other delay, such as interrupt, page fault, or signal handling) can be devastating for performance: it reduces the rate of progress from barrier to barrier to that of the slowest process for each interval. The effect is especially bad for preemption, where the faster processes contribute most of their energy to further slowing down the already-slow processes, using the very cycles the slower processes need so desperately. There are two general approaches to dealing with this: avoiding or reducing delays, and lessening their impact.

Avoiding or Reducing the Incidence of Delays on Barriers

Asynchronously generated signals are not used at all in some applications, and for most algorithms using them, such signals may be masked without difficulty, even for a long time. Long term allocation of processors, such as provided by a Symunix-2 scheduling group with a non-preemptive scheduling policy (§4.6.3), is helpful to avoid scheduling delays, but doesn't address short delays caused by interrupts. Support for long-term masking of interrupts may or may not be feasible, depending on the machine design. It may be possible to direct all I/O interrupts to other processors (as in the Sequent Balance [19, 80] and Symmetry [139] computers, and also in some Intel Pentium-based multiprocessors [200]).

Synchronous exceptions fall into several categories:

- Software emulation of certain functions not provided by hardware, such as the integer division instructions defined but not implemented for the AMD 29050 [2] (used in the Ultra-3 prototype), or floating point traps used to implement denormalized numbers or other difficult features of IEEE standard binary floating point arithmetic [112] on several new microprocessor designs. These are often difficult to avoid, because they are manifestations of otherwise minor machine details.

- Page faults, which might possibly be avoided by wiring down all relevant pages (as in section 7.2.3 on page 272). Even without true page faults, the cost of TLB reload may be significant when inter-barrier intervals are short.
- Memory access exceptions that are not errors, such as stack growth on subroutine call, discussed in section 7.6.1. In principle, these can be avoided by ensuring that enough real memory has been allocated in advance for the stack area of the address space. This is easier than wiring down non-stack pages. The only difficulty is deciding how much stack might be needed, and that can be determined through experimentation (assuming no real-time constraints).

Of course, even if all these techniques are effectively used, processors will still not execute in perfect lock-step, due to dynamic differences in execution paths and stochastic delays in accessing memory.

Lessening the Impact of Delays on Barriers

The technique of the fuzzy barrier (Gupta [97]) can be used to increase tolerance of random delays, but often only to the extent of a few instructions; page faults exceed that by several orders of magnitude.

If the number of application threads exceeds the number of available processors, the cost of waiting at barriers can be limited by using context-switching to another thread instead of busy-waiting. This is true whether thread management is performed in the kernel, in user libraries, or some combination of the two, although the cost varies according to the implementation. It is still true that a thread encountering a synchronous delay (such as page fault) cannot reach the barrier until the problem is successfully resolved, but if there are enough threads that don't get page faults and the average inter-barrier execution time is large enough, much of the delay can be hidden. Large asynchronous delays (such as preemption) are handled pretty well by context-switching barriers.

In sections 7.3.5 and 7.3.5, we give examples of some of these techniques.

7.3. Context-Switching Synchronization

Section 7.2 described busy-waiting synchronization methods for use in user-mode. Busy-waiting is the cheapest possible form of synchronization when no waiting or only very short waiting is necessary. There are times, however, when longer waiting times may occur, even if only under rare circumstances. In such cases, busy-waiting can be very expensive, because it consumes processor resources without contributing to useful work. To make matters worse, the processor cycles consumed may be needed to complete the work being waited for.

Context-switching synchronization has most of the opposite properties of busy-waiting:

- The minimum waiting time is not extremely short, due to context-switching overhead. (System call overhead is also a factor, but user-mode thread management can avoid it; see section 7.5.)
- No extra processor resources are consumed while waiting.

7.3.1. Standard UNIX Support for Context-Switching Synchronization

UNIX has traditionally provided the `pause` system call[148] to allow a process to await the arrival of a signal. Systems adhering to the System V Interface Definition [12] usually provide a semaphore mechanism which can be used for general synchronization, but it has some unattractive characteristics for highly parallel applications:

- *Complexity.* A dizzying set of features makes the mechanism somewhat hard to use and introduces otherwise unnecessary serialization.
- *Overhead.* Before a semaphore can be used, the `semget` system call must be used to translate a user-provided "key" into a "semaphore identifier". Allocation within the kernel is performed at this point if necessary. A semaphore is not deallocated until the last reference is deleted with the `semctl` system call.
- *Resource management.* It is quite possible for the system to run out of semaphores, causing `semget` calls to fail. In addition, all deallocations are explicit; there is no way to recognize unused semaphore structures.
- *Name space.* The semaphore "key" space is flat, consisting of integers. Management of this space (i.e., deciding which keys are used for each purpose) is anarchic.

An improved semaphore mechanism we call *ksems* will be presented in section 7.3.4. Notwithstanding the disadvantages of System V semaphores just mentioned, some, but not all, of our techniques for using ksems can be applied to System V semaphores if necessary, and are therefore also useful on conventional systems.

System V semaphores do offer at least one potentially useful feature that is unavailable with our ksem design: the automatic adjustment of semaphore values on unexpected process termination. However, use of this feature is incompatible with all of the useful ksem-based techniques we advocate.

7.3.2. Challenges and Techniques

There are two big challenges for context-switching synchronization of user programs:

(1) *Avoiding overhead in the common case*, where no blocking or unblocking is required. Kernel intervention is needed for blocking and unblocking, but we would like the common case to be handled without the kernel. We will show how this can be achieved by splitting the implementation between user-mode and kernel-mode. The only kernel-mode facility required is a simple context-switching counting semaphore (such as provided by System V semaphores or Symunix-2 ksems).

(2) *Avoiding serial bottlenecks.* Serialization is usually the name of the game for binary semaphores, but many other forms of synchronization allow more parallelism. These include counting semaphores, readers/writers, readers/readers, and group locks, barriers, and events.[149] We will show how this additional parallelism can be realized by using appropriate user-mode algorithms together with a careful kernel implementation, such as we describe for ksems.

---

[148] The `sigpause` or `sigsuspend` variant is also available in most recent UNIX versions. (These system calls atomically alter the signal mask and pause until an unmasked signal is received.)

[149] See §4.5, §3.5.6, and §7.2.4.

We use a two level approach:

- The synchronization method is implemented with all coordination variables in user space and manipulated directly.
- Kernel-provided semaphores are used to provide waiting only. The problem is addressed as a producer/consumer problem, where the resource being managed is "opportunities to be unblocked".

We will now give two examples.

Suppose we want counting semaphores. We could use kernel-provided semaphores directly, but that certainly wouldn't avoid kernel overhead, even in the common case. Instead, we implement the "real" counting semaphore directly in user-mode, and use a kernel-provided semaphore only for blocking:

```
P (int *s)                          V (int *s)
{                                   {
     if (fad(s) <= 0)                    if (fai(s) < 0)
          KP(s??);                            KV(s??);
}                                   }
```

Here `P` and `V` are the traditional semaphore operations, `KP` and `KV` are the kernel-provided versions, and *s*?? is some parameter that tells the kernel which semaphore to use (i.e., the one we associate with `*s`; this example is reproduced in more complete form in section 7.3.5 using ksems). The kernel's semaphore value must be initialized to 0, but the user semaphore `*s` can be initialized to any non-negative value. This code ignores the possibility of integer overflow or underflow; ignoring that, it is simple and easy to verify. Clearly it meets the requirements of challenge 1, avoiding kernel intervention unless blocking or unblocking is necessary. If the kernel is implemented with highly-parallel scheduling and semaphore waiting lists, we will also have met challenge 2.

Now let us consider the example of readers/writers locks. Wilson [205] gives an algorithm we already used in section 4.5.2; a simple transformation adapts it for our current purposes as well:

```
struct rwlock {
     int c;          // counter
     int rr;         // running readers
};
```

```
rlock (struct rwlock *rw)
{
     if (fad (&rw->c) <= 0)
          KP (r??);
}

runlock (struct rwlock *rw)
{
     if (fai (&rw->c) < 0 &&
         fai (&rw->rr) == BIG-1) {
          // last reader wakes writer
          rw->rr = 0;
          KV (w??);
     }
}

wlock (struct rwlock *rw)
{
     int x = faa (&rw->c, -BIG);
     if (x == BIG)
          return;
     if (x > 0 &&
         faa (&rw->rr, x) == BIG-x)
          rw->rr = 0;
     else
          KP (w??);
}

wunlock (struct rwlock *rw)
{
     int x = faa (&rw->c, BIG);
     if (x <= -BIG) {
          // Wake next writer
          KV (w??);
     }
     else if (x < 0) {
          // Wake all readers
          KV (r??, -x);
     }
}
```

This particular algorithm gives priority to writers. We use two kernel-provided semaphores, represented here by *r*?? (for waiting readers) and *w*?? (for waiting writers). Again, they must be initialized to the value 0. The value of `rw->c` should be initialized to `BIG` (larger than the maximum number of concurrent readers) to signify a completely open lock, and `rw->rr` should be initialized to 0. We extend our temporary notation for kernel-provided semaphore operations by allowing a second argument to `KV`, used for specifying an increment greater than 1. This allows a suitable kernel implementation to perform the wakeups in parallel (see section 7.3.4 on page 280 and section 4.6.9; this is also based on Wilson [205]). Again we have met both challenges.

Algorithms for other forms of synchronization are given in section 7.3.5.

7.3.3. Hybrid Synchronization

Context-switching synchronization algorithms such as those just given are fine when expected lock holding times are always long. What about situations where the holding times are highly variable? This can arise, for example, due to preemption or page faults. When the waiting time will be short, busy-waiting is best. When the waiting time will be long, context-switching is best. What if you don't know which to expect? One solution is to use *hybrid* algorithms (dubbed "two-phase blocking" by Ousterhout [160]) which busy-wait for a bounded time, then revert to blocking.

A simple technique for constructing a hybrid algorithm from a basic context-switching version (such as the counting semaphore and readers/writers lock algorithms already presented in this section) is to add a bounded busy-waiting loop before the code that might block. For example, here is a hybrid counting semaphore algorithm based on the example in section 7.3.2 on page 277:

```
P(int *s)                           V(int *s)
{                                   {
      int delay = BOUND;                  if (fai(s) < 0)
      while (delay-- > 0)                       KV(s??);
            if (s > 0)              }
                  break;
      if (fad(s) <= 0)
            KP(s??);
}
```

The only difference is the additional loop at the beginning of P. Since the original algorithm is correct, this one is too, because the additional code is purely advisory.

This technique for constructing hybrid algorithms is applicable to a broad class of context-switching algorithms. Besides a correct original (pure context-switching) algorithm, the only requirement is that there be a way to test for probable success without modifying anything. However, there are some anomalies that can occur:

- A process can block even without waiting for BOUND iterations.
- Even if the original algorithm and the kernel semaphores are fair, the hybrid algorithm is at best only probabilistically fair. Furthermore, any remaining claim to fairness is lost if the exponential backoff technique of section 3.5.1 on page 53 is applied to the bounded busy-waiting loop.

It isn't clear that these anomalies cause any real problems.

7.3.4. Ksems

We described System V semaphores and some of their weaknesses in section 7.3.1. We now describe a simpler mechanism, which we call *ksems*, specifically designed to support highly-parallel applications as well as synchronization of unrelated processes. Ksems don't have very many features because they are designed primarily to support the kind of usage exhibited so far in this section: user-mode synchronization that avoids kernel intervention, and hybrid synchronization. The fact that ksems may be substituted for some common uses of System V semaphores is of secondary importance.[150]

The most difficult part of resource management for ksems is left to the user. The semaphore value is an ordinary integer, allocated and initialized by the user.[151] The kernel is responsible for waiting lists and actually performing the P and V operations on ksems.

---

[150] It is possible to provide only a partial emulation of System V semaphores "on top of" ksems; in particular, automatic semaphore value adjustment on abnormal termination can't be done reliably without direct kernel involvement. Atomic operations on more than one semaphore and operations to alter a semaphore's value by other than ±1 are also difficult to emulate in terms of `ksemp` and `ksemv`.

[151] Perhaps the type should be `faa_t`, as mentioned in section 3.1 on page 40, or a new type, say `ksem_t`, to give the implementation a bit more flexibility.

While there is nothing to stop the user from accessing or modifying the semaphore value directly, it is not often desirable (we give an example of useful ksem modification by the user in section 7.3.5 on page 297). Such accesses cannot damage the system and their effect is well defined.

The two semaphore operations, P and V, are represented as system calls:

`ksemp (int *`*addr*`)`
: The ksem value at address `*`*addr* is decremented. Return is immediate if the old value was positive, otherwise the process blocks. The unique device/i-node/offset to which *addr* maps is used as a key to search for the waiting list when blocking;[152] a new waiting list is allocated if not already present. The value returned by `ksemp` is 0 unless a caught signal interrupts the blocked process, in which case the returned value is −1 and `errno` is set to `EINTR`.

`int ksemv (int *`*addr*`, int `*inc*`)`
: The ksem value at address *addr* is incremented by *inc*, which must be positive. Return of 0 is immediate if the old value wasn't negative, otherwise the waiting list is found and up to $\min(-old, inc)$ activities are awakened. An empty waiting list is deallocated. The number of activities not found for awakening is returned, i.e., $\max(0, \min(-old, inc) - nw)$, where *nw* is the actual number of waiting activities on the list.

The ksem design, as thus presented, has a number of desirable properties:[153]

- To the extent the user can allocate sufficient memory, the number of available ksems is inexhaustible.
- There are no identifiers, descriptors, or capabilities to deal with when using ksems.
- Access control for ksems is identical to access control for memory. If a piece of memory can be shared, ksems can be placed in it (of course, write permission is required).
- The maximum number of waiting lists required is equal to the maximum number of activities allowed in the system.[154]
- Unused ksems are automatically deallocated when the image file containing them is deallocated (see section 5.2.6 for how this can be done automatically on process termination).

---

[152] Recall from section 5.2 that all user-accessible memory in Symunix-2 is the result of mapping files into the user's address space. The ksem mechanism can be adapted for other systems, with different memory models, as long as there is some value derivable from *addr* that remains constant during the life of the ksem. An ordinary physical memory address won't work on most systems unless the page is *wired down* in memory for the full duration of any ksem operation on it, including the full time during which any process is on the waiting list. Using the physical address in this way is probably practical, but depends on many details beyond the scope of this dissertation.

[153] None of these properties hold for System V semaphores.

[154] If an asynchronous version of `ksemp` isn't provided (via `syscall`, see section 4.3.1), then the maximum number of waiting lists required is decreased to the maximum number of processes allowed in the system. The value of asynchronous lock acquisition may seem dubious, but the ability to handle signals without disrupting `ksemp` might have some merit. For simplicity of presentation, we generally ignore the possibility of asynchronous `ksemp` in our examples, and "process" can generally be used in place of "activity" if that makes the reader more comfortable.

There can be no waiting lists associated with deallocated ksems, because a waiting process must have a reference to the image file, hence the latter couldn't be deallocated.

- When `ksemv` unblocks more than one activity, it can take advantage of the scheduler's ability to do so in sub-linear time (see section 4.6.8).
- By returning the number of activities not awakened, `ksemv` allows user-mode compensation for premature unblocking and conditional blocking operations (as we will show in our example semaphore code in section 7.3.5 on page 296). Without this return value, we could not support the two level approach to synchronization described in section 7.3.2 on page 277 (unless we ruled out signals and conditional blocking operations); all P and V operations would need to invoke the kernel.

Ksem Implementation

The major difficulties in implementing ksems for the target class of machines are:

(1) *Parallel Hash Table.* A highly parallel structure must be devised to support searching and allocation of waiting lists.

(2) *Synchronization between* `ksemp` *and* `ksemv`. In the wakeup routines of section 4.5.2, we busy-waited until an activity (or a "hole") could be deleted. Such a strategy won't work for ksems because the user has direct access to the ksem value and can change it arbitrarily. For example, if the user changes the ksem value from 1 to −1, a subsequent `ksemv` could wait forever without finding anything to delete from the waiting list.

(3) *Parallel, Memory-Efficient Wait Lists Without Holes.* A highly parallel wait list algorithm must be used. We would like to use one of the algorithms in section 3.7.1, but two problems prevent this:

- *Memory usage*. The number of ksem waiting lists and items will both grow with the number of processors in a machine. As discussed in section 3.7.1 on page 79, a poorly chosen list algorithm can lead to explosive memory usage. We also need interior removal, but the memory efficient lists in Table 14, in section 3.7.1 on page 83, don't support interior removal.
- *Holes*. The bound on the total number of waiting lists required for the whole system relies on their dynamic allocation and deallocation by the kernel. The use of holes as placeholders for past interior removals will cause some lists to appear nonempty, perhaps forever, thus destroying the bound. Even if we had an algorithm that could recognize an "only holes" list as being "empty", we still have to worry about the gradual accumulation of holes, possibly causing other problems, such as integer overflow (see section 3.7.1 on page 80).

Meeting all these requirements for an efficient parallel-access list in the style of section 3.7.1 appears difficult or impossible.

(4) *Premature Unblocking.* In section 4.5, premature unblocking is accommodated by leaving an explicit hole in the waiting list, and taking a compensating action when it is encountered during deletion (e.g., `_icsrw_runblock` and `_icsrw_rwakeup` (§4.5.3/p186, §4.5.2/p182)). In this case, we cannot use such a strategy because explicit holes would prevent otherwise empty lists from being deallocated. In addition, since users have direct access to the ksem value, they can decrement it; if the resulting value is below 0, it is like inserting a *virtual* hole on the waiting list. Virtual holes are reported to the user by the return value of `ksemv`; when an activity is

prematurely unblocked, we want to inform the user in the same way.

(5) *Parallel Wakeup.* The `ksemv` function must have a way of awakening multiple activities in sub-linear time. In section 4.6.8 we inserted a multi-item onto the system ready list to point at a waiting list. We wish to use a similar approach for ksems, but since we can't keep holes in the ksem waiting list, we have a problem for the implementation of premature unblocking: an activity must not be disturbed if it has already been logically transferred to the ready list. (This was not a serious problem in section 4.5.3 on page 186 because of the way holes are handled there.)

For the synchronization problem between `ksemp` and `ksemv`, we use a busy-waiting group lock (§3.5.6) in the kernel to prevent overlapping execution of the value modification and wait list actions in `ksemp` and `ksemv`. Any number of `ksemp` calls may execute concurrently, or any number of `ksemv` calls, but the two types must exclude each other in time.[155] By segregating the two ksem operations, `ksemv` never has to wait for a `ksemp` to complete list insertion and `ksemv` never needs to allocate a waiting list, as a non-allocated waiting list is equivalent to an empty one.[156]

We use an approach similar to that described in section 3.7.6 to allow parallel searching for ksem waiting lists. Synchronization and reference counting are simplified due to the use of a group lock to segregate different kinds of ksem operations, as just described. In particular, the reference count of section 3.7.6 is unnecessary, as we must also keep a count of

Function	*See Page*	*Purpose*
`ksemp`	286	Semaphore P (wait)
`ksemv`	287	Semaphore V (signal)
`ksem_unblock`	292	Premature unblocking
`ksem_get`	294	Search for & allocate `ksem_wait` struct
`ksem_put`	295	Deallocate `ksem_wait` struct
`kswl_wakeup`	288	Wake up blocked activities for `ksemv`
`kswl_put`	289	Insert activity at rear of waiting list
`kswl_put_pos`	289	Insert activity at specific position
`kswl_get`	290	Delete activity from front of waiting list
`kswl_get_pos`	291	Remove activity from specific position

Table 36: Ksem Implementation Functions.

[155] A readers/readers lock could be used, but the algorithm given in section 3.5.5 gives absolute priority to one class over the other, which is probably not desirable in this context. Group lock also lends itself to a nice implementation of interior removal for `ksem_unblock`, on page 292.

[156] A group lock or readers/readers lock could have been used to avoid the need for busy-waiting in section 4.5, but would be less efficient when the blocking and waking-up operations are not racing each other. Furthermore, such an approach favors use of list algorithms that are very tightly integrated with the context-switching synchronization algorithms, weakening software modularity.

waiting activities.

For the activity waiting list, we enhance Dimitrovsky's hash table-based queue algorithm [63, 65]. (Thus, there are *two* hash tables to be described here: one for finding a waiting list, and one to implement the actual lists of blocked activities.) Holes in a waiting list, which would otherwise be created by premature unblocking, are avoided by filling a vacated position with the last activity on the list.[157] This solution is feasible primarily because inserts and deletions are segregated.

We want to implement `ksemv` with an increment greater than 1 efficiently by inserting onto the system ready list a multi-item pointing at the ksem waiting list. To prevent premature unblocking from effectively putting holes on the system ready list, we maintain two head pointers for the waiting list: one for ordinary deletions (from the ready list), and one for use in `ksemv` when some activities must be "committed" to the system ready list. Activities committed to the ready list but still physically part of the ksem waiting list are not subject to premature unblocking.

Further details are given in the following code segments. Table 36, on the previous page, lists all the ksem-related functions.

[157] This is a violation of FIFO semantics, but is still starvation free. As discussed in section 3.7.1 on page 78, FIFO semantics are not always needed. Unfortunately, we don't currently know of a parallel-access FIFO list algorithm supporting interior removal without holes. A linked list with a group lock and a recursive doubling algorithm seems a likely candidate, but we have not worked out such a solution in detail.

```
     // structure for a ksem waiting list, dynamically allocated
struct ksem_wait {
     dev_t            dev;       // device containing image file
     struct inode     *ino;      // image file
     off_t            off;       // offset of ksem in image file
     struct ksem_hash *hbucket;  // hash bucket for dev/ino/off

     struct ksem_wlist {         // wait list itself
          int          nwait;    // number of blocked activities
          unsigned int tail;     // position for insert
          unsigned int head;     // position for delete
          unsigned int commit;   // position for wakeup commit
          int          unblocks; // number of concurrent unblocks
     } wl;

     union {
          poolitem     pi;       // ksem_wait free list
          struct _kwp {          // for hash bucket linked list
               struct _kwp *forw;
               struct _kwp *back;
          } kwp;
     } ptrs;
};

     // structure for a ksem hash table bucket
     // this is the hash table for finding a waiting list
struct ksem_hash {
     struct _kwp        kwp;     // head of bucket list
     bwrwlock           rw;      // to coordinate hash table access
};

     // hash bucket for activity wait list
     // all ksem wait lists share the same hash table
struct ksem_wlhash {
     struct ksem_wlitem *first;  // head of bucket list
     bwlock            lock;     // serialize pointer manipulation
};

     // wait list item for an activity
     // included in activity structure as a_li.kswi
struct ksem_wlitem {
     struct ksem_wlitem *next;    // linked list pointer
     int                kindex;   // index in ksem_wait array
     unsigned int       position; // position in waiting list
};
```

```
struct ksem_wait   *ksem_wait;      // array of all ksem_wait structs
struct ksem_hash   *ksem_hash;      // the ksem_wait hash table array
struct ksem_wlhash *ksem_wlhash;    // the activity hash table array
unsigned int       nksem_hash;      // size of ksem_hash table
unsigned int       nksem_wlhash;    // size of ksem_wlhash table
bwglock            ksem_all_glock;  // separate p, v, unblock ops
pool               ksem_pool;       // free list of ksem_wait structs

// hash functions
#define KSEM_HASH(dev,ino,off)  (((dev)*(ino)*(off))%nksem_hash)
#define KSEM_WLHASH(ind,pos)    (((ind)*(pos))%nksem_wlhash)
```

The `ksem_wait` structures are allocated as an array at boot time (with the same name as the structure tag, `ksem_wait`), but are immediately placed individually into a free list (`ksem_pool`) for dynamic allocation/deallocation as needed for ksems. Most fields of the `ksem_wait` structure needn't be initialized at that time, but the fields in `wl` should be set to all zeros.

A hash table, `ksem_hash`, is used to locate a ksem's wait list using the unique `dev`/`ino`/`off` key and the method of section 3.7.6. Hash collisions are handled with a doubly-linked list for each hash bucket, and insertions and deletions are coordinated with a busy-waiting readers/writers lock. The `kwp` pointers in the hash bucket must be initialized to point back at themselves, e.g.,

```
for (i = 0;  i < nksem_hash;  i++)
     ksem_hash[i].kwp.forw = ksem_hash[i].kwp.back =
                             &ksem_hash[i].kwp;
```

A second hash table, `ksem_wlhash`, is used to implement the lists upon which blocked activities actually wait.[158] The index of a `ksem_wait` structure within the `ksem_wait` array provides a convenient identifier for the wait list, and is used by the hash function, `KSEM_WLHASH`, to ensure each wait list uses a distinct sequence of buckets in the `ksem_wlhash` table. The `first` pointer within each `ksem_wlhash` structure must be initialized to `NULL`. For machines outside our target class, it may be advisable to replace this second hash table (together the `ksem_wlist` structure, the `ksem_wlitem` structure, and the routines with names beginning with `kswl_` in Table 36) with versions more appropriate to the machine in question.

A single group lock, `ksem_all_glock`, is used to ensure mutual exclusion between all `ksemp`, `ksemv`, and `ksem_unblock` calls. If desired, an array of group locks could be used, with selection performed by hashing of either ksem addresses or `ksem_hash` indices; this would reduce group lock contention.

Although not strictly necessary, we mask soft interrupts while using the group lock; this reduces the effective execution speed variance of cooperating processors at the cost of somewhat increased soft interrupt latency. Masking hard interrupts would offer even less

[158] It might be better to implement this in common with another hash table, such as one used for ordinary waiting lists (see section 3.7.1 on page 82).

variance but might not be acceptable, depending on hard interrupt latency requirements.

```
	// ksemp system call
ksemp (int *addr)
{
	int oldv, r=0;

	vsplsoft();
	bwg_lock (&ksem_all_glock, splcheck0);
	bwg_sync (&ksem_all_glock);		// wait for phase 2
	if (!user_fad (addr, &oldv)) {		// decrement ksem
		bwg_unlock (&ksem_all_glock);
		return -EFAULT;			// bad user address
	}
	if (oldv > 0)
		bwg_unlock (&ksem_all_glock);	// no blocking needed
	else {					// must block
		struct ksem_wait *kw = ksem_get (addr, 1);
		a->a_waitfor.kw = kw;
		a->a_waittype = ACT_KSEM | ACT_MAYCANCEL;
		int pf = a->a_pflags; // private flags; no locking required
		bwl_wait (&a->a_lock, nullf);
		int f = a->a_flags;
		if (f & ACT_CANCEL) {
			a->a_flags = f & ~ACT_CANCEL;
			bwl_signal (&a->a_lock);
			bwg_unlock (&ksem_all_glock);
			r = -EINTR;
		}
		else {
			struct activity *n = _cs_common (a, pf);
			kswl_put (kw, &a->a_li.kswi);
			bwl_signal (&a->a_lock);
			bwg_unlock (&ksem_all_glock);
			cswitch (n ? n : resched(0));
			if (a->a_pflags & ACT_DISTURBED) {
				a->a_pflags &= ~ACT_DISTURBED;
				r = -EINTR;
			}
		}
	}
	vspl0();				// unmask interrupts
	return r;
}
```

We show the return value as 0 for success and a negative value (traditional `errno` value) for errors. We rely on the function `user_fad` to perform the Fetch&Decrement operation in the kernel for a location specified by a user virtual address; it has to deal with memory management issues and the possibility of an attempted invalid access.[159] If blocking is required, the function ksem_get, to be described on page 294, locates or allocates a ksem_wait structure. Most of the code following ksem_get has the same pattern as the blocking functions of section 4.5. The _cs_common function, which mostly handles possible asynchronous activity creation, is shown in section 4.5.2 on page 178. Actual waiting list insertion is handled by a separate function, kswl_put, which will be described on page 289.

We saw that the group lock's phase 2 is used for ksemp; phase 1 is used for ksemv. This choice of ordering for group lock phases isn't very important, but when some of each operation on the same ksem are involved in a single group, this order enables the ksemps to benefit from the ksemvs.

```
      // ksemv system call
ksemv (int *addr, int inc)
{
      int oldv, r=0;
      if (inc <= 0)
            return -EINVAL;                 // inc doesn't make sense
      vsplsoft();
      bwg_lock (&ksem_all_glock, splcheck0);
      if (!user_faa (addr, inc, &oldv))     // increment ksem
            r = -EFAULT;
      else if (oldv < 0) {
            int trywake = MIN (-oldv, inc); // how many we try to wake
            int nawake;                     // how many we actually wake
            struct ksem_wait *kw = ksem_get (addr, 0);
            if (kw == NULL)
                  nawake = 0;               // no one was waiting
            else
                  nawake = kswl_wakeup (kw, trywake);
            r = trywake - nawake;           // number of virtual holes
      }
      bwg_unlock (&ksem_all_glock);
      vspl0();
      return r;
}
```

The user is prohibited from using ksemv to decrement the ksem. The normal return value of ksemv is a non-negative integer giving the number of virtual holes encountered. Traditional errno values, such as EFAULT, which are positive numbers, can easily be distinguished since the value returned is their additive inverse.

[159] user_fad must be implemented atomically with respect to any Fetch&Φ operations executed directly in user-mode. Recall from section 3.1 on page 41 that some machines may require software emulation of some Fetch&Φ operations. For the sake of portability to such machines, we should provide additional user-visible operations, such as ksem_faa and ksem_fas; these would be the same as ordinary faa and fas functions on machines with HARD_faa and HARD_fas defined, but would be carefully implemented to be compatible with the kernel's user_fad routine on other machines.

The key operation, incrementing the ksem, is performed by the `user_faa` function, which operates on a location specified by a user-supplied virtual address.[160] If the old value is negative, a wakeup is logically necessary. We compute the number of activities we would like to wake up, based on the user-supplied `inc` and the old ksem value. There may actually be fewer activities to awake due to past removals. The `ksem_get` function is called to locate a waiting list, but no allocation is performed if one can't be found (an unallocated waiting list is equivalent to an empty one). We call `kswl_wakeup` to do the work of waking up blocked activities and deallocating the waiting list if necessary:

```
      // wake up as many as trywake blocked activities
      // call ksem_put to deallocate waiting list if empty
      // return actual number awakened
int
kswl_wakeup (struct ksem_wait *kw, int trywake)
{
      int nawake;
      int nw = faa (&kw->wl.nwait, -trywake);
      if (nw < trywake)           // not enough waiting?
           vfaa (&kw->wl.nwait, MIN (trywake, trywake-nw));
      if (nw > trywake)           // too many waiting?
           nawake = trywake;
      else if (nw <= 0)           // none waiting?
           nawake = 0;
      else                        // some, but not too many, waiting
           nawake = nw;
      if (nawake > 0) {           // wake up nawake activities
           extern mqitem *kswl_get (mqitem *);
           mqiitem *mqii = (mqiitem *)poolget(&mqiipool);
           mqireinit (&mqii->ptrs.mqi);
           mqii->ptrs.mqi.f = kswl_get;
           mqii->p = kw;
           mqii->rcnt = nawake;
           vufaa (&kw->wl.commit, nawake);
           mkqready (&mqii->ptrs.mqi, PRI_KSEM, nawake);
      }
      if (nw <= trywake)          // deallocate
           ksem_put (kw);
      return nawake;
}
```

The most crucial step in `kswl_wakeup` is reserving some waiting activities to wake up. This is done by decrementing the waiting list count `wl.nwait` with `faa`, and compensating with `vfaa` if the value was driven below zero. Because of the group lock, `wl.nwait` cannot be concurrently raised above 0.

---

[160] The same comments about atomicity apply as for the `user_fad` function (see footnote 159, on page 287).

The actual wakeup is performed by inserting an indirect multi-item on the system ready list with `mkqready` (§4.6.9). The activities will be individually deleted from the ksem waiting list at some time in the future, when the indirect multi-item is encountered on the system ready list. Since the system ready list functions don't know anything about ksems, the function `kswl_get`, on the next page, will do the actual deletion and take care of final disposition of the indirect multi-item.

The actual waiting list algorithm is primarily embodied in the two routines `kswl_put` and `kswl_get`, which are based on Dimitrovsky's hash table queue algorithm [63, 65]:

```
      // insert activity on actual ksem waiting list
void
kswl_put (struct ksem_wait *kw, struct ksem_wlitem *item)
{
      kswl_put_pos (item, kw - ksem_wait, ufai (&kw->wl.tail));
}

      // insert activity at given position on waiting list
void
kswl_put_pos (struct ksem_wlitem *item, int kindex, unsigned int position)
{
      item->kindex = kindex;
      item->position = position;
      struct ksem_wlhash *hb;
      hb = &ksem_wlhash[KSEM_WLHASH (kindex, position)];
      spl_t s = splsoft();
      bwl_wait (&hb->lock, splcheckf(s));
      item->next = hb->first
      hb->first = item;
      vfai (&kw->wl.nwait);
      bwl_signal (&hb->lock);
      vsplx(s);
}
```

The bucket list for waiting activities is singly-linked and unordered. We omit overflow prevention code for `kw->wl.tail` under the assumption that the hash function takes the position modulo a power of two that is smaller than the integer word size (e.g., the hash table size should be such a power of two). We aren't showing complete initialization code, but the buckets' `first` pointers should all be initialized to `NULL`.

The complement to `kswl_put` is `kswl_get`, except that the latter has two peculiarities:

- To meet the needs of the system ready list, it is called with an `mqitem` pointer and also returns an `mqitem` pointer.
- It is responsible for deallocating the indirect multi-item (`mqiitem`) allocated in `kswl_wakeup`. This is done with a reference count, set in `kswl_wakeup` and decremented in `kswl_get`.

```
          // delete next activity from waiting list
          // called with indirect multi-item from system ready list
     mqitem *
     kswl_get (mqitem *mqi)
     {
          struct ksem_wait *kw = (struct ksem_wait *)mqi->p;

          // get item off list
          struct ksem_wlitem *item = kswl_get_pos (kw, ufai (&kw->wl.head));
          assert (item != NULL);      // ksem_all_glock prevents failure

          // dispose of indirect mq item after last deletion
          mqiitem *mqii = (mqiitem *)((char *)mqi -
                          offsetof(mqiitem, ptrs.mqi));
          if (fad (&mqii->rcnt) == 1) {
               poolireinit (&mqii->ptrs.pi);
               poolput (mqiipool, &mqii->ptrs.pi);
          }

          // convert item ptr to mq item pointer in activity
          struct activity *act = (struct activity *)((char *)item -
                                 offsetof(struct activity, a_li.kswi));
          return &act->a_li.mqi;
     }
```

The first part of `kswl_get` is just a call to a function, `kswl_get_pos`, on the next page, to perform the actual hash table deletion. Even though the group lock is not held during `kswl_get`, it has done its job: the bucket list search cannot fail because of the way `kswl_put`, `kswl_wakeup`, and `ksem_unblock` work together in different group lock phases.

```
	// delete item at given position in list
struct ksem_wlitem *
kswl_get_pos (struct ksem_wait *kw, unsigned int position)
{
	int kindex = kw - ksem_wait;
	struct ksem_wlhash *hb;
	hb = &ksem_wlhash[KSEM_WLHASH (kindex, position)];
	struct ksem_wlitem *item, **prev;
	spl_t s = splsoft();
	bwl_wait (&hb->lock, splcheckf(s));
	// search for item in hash table
	for (prev = &hb->first, item = *prev;
		item != NULL;
		prev = &item->next, item = *prev) {
		if (item->position == position && item->kindex = kindex) {
			// item found, unlink it
			*prev = item->next;
			break;
		}
	}
	bwl_signal (&hb->lock);
	vsplx(s);
	return item;
}
```

As with `kw->wl.tail` in `kswl_put_pos`, we do not need explicit overflow protection on `kw->wl.head` here.

If it weren't for the possibility of premature unblocking, the ksem waiting list for activities would be much easier to implement in a highly parallel fashion. We have seen the "normal" insertion and deletion operations, now we are ready for premature unblocking itself. The function `ksem_unblock` is called only from the `unblock` function shown in section 4.5.3 on page 185:

```
        // prematurely unblock activity
        // activity is already locked and soft interrupts are masked
        // returns true/false
int
ksem_unblock (struct activity *act)
{
        struct ksem_wait *kw = act->a_waitfor.kw;
        struct ksem_wlitem *item = &act->a_li.kswi;
        int kindex = item->kindex;
        unsigned int position = item->position;
        int h = KSEM_WLHASH (kindex, position);
        struct ksem_wlhash *hb = &ksem_wlhash[h];

        bwg_lock (&ksem_all_glock, nullf);   // phase 1: ksemv
        bwg_sync (&ksem_all_glock);          // phase 2: ksemp
        bwg_sync (&ksem_all_glock);          // phase 3: unblock removal
        unsigned int oldtail = kw->wl.tail;
        if (udiff (kw->wl.commit, position) > 0 ||
            udiff (oldtail, position) <= 0) {
                // item awakened or committed: unblock not possible
                bwg_unlock (&ksem_all_glock);
                return 0;
        }
        struct ksem_wlitem *item2 = kswl_get_pos (kw, position);
        assert (item2 == item);               // can't fail
        vfai (&kw->wl.unblocks);

        int seq = bwg_sync (&ksem_all_glock); // phase 4: hole filling
        int unblocks = kw->wl.unblocks;
        if (udiff (oldtail - unblocks, position) > 0) {
                do {
                        unsigned int rposition = ufad (&kw->wl.tail) - 1;
                        item2 = kswl_get_pos (kw, rposition);
                } while (item2 == NULL);
                ksem_put_pos (item2, kindex, position);
        }

        if (seq == 0) {
                bwg_sync (&ksem_all_glock);     // phase 5: serial cleanup
                kw->wl.tail = oldtail - unblocks;
                kw->wl.unblocks = 0;
                if (faa (&kw->wl.nwait, -unblocks) == unblocks)
                        ksem_put (kw);
        }
        bwg_unlock (&ksem_all_glock);
        return 1;
}
```

One of the first things to consider about ksem_unblock is the possibility of deadlock: the activity is already locked, and ksem_all_glock is also to be locked, while ksemp does things the other way around.[161] Despite this apparent mismatch, there is no deadlock because ksemp and ksem_unblock can't be operating on the same activity at the same time.

Because we maintain head, tail, and commit only modulo $2^{wordsize}$, the comparisons must be done carefully; this is the purpose of the udiff function (§3.5.6/p72). As used here, there is still a race: the test will give the wrong indication if kswl_wakeup is able to increment the commit field almost $2^{wordsize}$ times between the time the activity to be unblocked is locked and the time of the first udiff call in ksem_unblock. This seems unlikely, but the chances could be improved further if desired by increasing the sizes of head, tail, commit, and position.

After removing the target activity from the waiting list, we have to see if a hole would be created. A hole would not be created, for example, if the removed activity were the last on the waiting list. In general, some of the concurrent unblocks directed at the same ksem will need to fill their holes and some won't. Since we fill holes from the rear of the list, we must cope with the possibility that the rear item is also being unblocked. We handle this by performing hole filling in a new group lock phase. The call to kswl_get_pos will fail if, and only if, the item at position rposition was removed in phase 3. The solution we adopt here is to fill our hole with the next rear-most item. This solution has worst case time proportional to the number of concurrent unblocks. An alternative is to build a list of good "hole fillers", similar to that done for "move-ups" in the visremove function in section 3.7.3 on page 105: a practical solution, but we omit the details because we've presented so many already. Other possible improvements we won't take the space to describe are the use of fuzzy barriers (§3.5.6/p69) and optimizations for the common case when the group size is 1.

Focusing our attention again on the dynamic allocation of ksem_wait structures, we see that although ksem_get is similar to the search function presented in section 3.7.6 on page 129, there is no need for a reference count beyond the wl.nwait field: ksem_put is only called when a waiting list is empty.

---

[161] Note that ksemv doesn't lock any activities directly, although this is an accident of the implementation: if we weren't so worried about serializing wakeups, we'd have written ksemv to delete activities from the waiting list one at a time and wake up each one individually, an action that requires the activity lock.

```
      // search for wait list
      // aflag indicates allocation for ksemp
ksem_get (int *addr, int aflag)
{
      dev_t dev;
      struct inode *ino;
      off_t off;
      struct _kwp *kwpp;
      struct ksem_wait *kw;

      derive dev, ino, and off from addr;
      int h = KSEM_HASH(dev,ino,off); // appropriate hash function
      struct ksem_hash *hb = &ksem_hash[h];
again:
      bwrw_rlock (&hb->rw, nullf);
      for (kwpp = hb->kwp.forw;
            kwpp != &hb->kwp;
            kwpp = kwpp->forw) {
           kw = (struct ksem_wait *)((char *)kwpp -
                          offsetof (struct ksem_wait, ptrs.kwp));
           if (kw->dev == dev && kw->ino == ino && kw->off == off)
                break;
      }
      if (kwpp != &hb->kwp || !aflag) {
           // wait list found or allocation prohibited
           bwrw_runlock(&hb->rw);
           return kw;
      }
      // not found, perform allocation with exclusive bucket lock
      if (!bwrw_rtow(&hb->rw)) {
           bwrw_runlock(&hb->rw);
           goto again;              // failed to get exclusive lock
      }
      kw = poolget(&ksem_pool);        // allocate from free list
      kw->dev = dev;
      kw->ino = ino;
      kw->off = off;
      kw->hbucket = hb;
      kw->ptrs.kwp.forw = &hb->kwp;
      (kw->ptrs.kwp.back = hb->kwp.back)->forw = &kw->ptrs.kwp;
      hb->kwp.back = &kw->ptrs.kwp;
      bwrw_wunlock(&hb->rw);
      return kw;
}
```

The highly parallel search strategy is based on hashing: assuming a good hash function, hash collisions will be rare, the bucket list will be short, and the only significant concurrency within a bucket will be due to searches for the same wait list. If the sought-after waiting list has already been allocated, a shared lock is obtained quickly and without serialization using the algorithm presented in section 3.5.4 on page 62. If not, the lock is upgraded in such a way that, among many concurrent upgrade requests, one will probably succeed and the others will fail. It is then a simple matter for the “winner” to perform the allocation and release the lock, allowing the “losers” to proceed without further serialization. If the upgrade fails because of two independent ksems experiencing a hash collision, extra retries will be performed.)

By using the group lock to segregate `ksemp` from `ksemv` and `ksem_unblock`, we know an empty list will stay empty while we deallocate it, which is done by calling `ksem_put`:

```
        // deallocate ksem_wait structure
        // ksem_all_glock is already held
void
ksem_put (struct ksem_wait *kw)
{
        struct ksem_hash *hb = kw->hbucket;

        bwrw_wlock (&hb->rw, nullf);
        struct _kwp *f = kw->ptrs.kwp.forw;
        struct _kwp *b = kw->ptrs.kwp.back;
        f->back = b;
        b->forw = f;
        bwrw_wunlock (&hb->rw);
        poolireinit (&kw->ptrs.pi);
        poolput (&ksem_pool, &kw->ptrs.pi);
}
```

Once the exclusive bucket lock is obtained, the `ksem_wait` structure is moved from the hash table bucket list to the pool of available waiting lists. Note that it would not generally be safe to deallocate a waiting list’s memory in such a way that it could be used for something else. This is because the waiting list itself may still be in use, containing activities that have been logically awakened, but not yet run from the system ready list.

7.3.5. Ksem Usage Examples

With ksems, users can write context-switching (and hybrid) coordination algorithms. In section 7.3.2 on page 277 we gave context-switching examples for counting semaphores and readers/writers locks using only a generic kernel-supplied semaphore, so we were vague about details. Now that we have described ksems, we can fill in the details. Figure 37, on the next page, provides a mini table of contents for the examples we present.

The simplest way to use ksems is to allocate one for each user-level waiting list. This is the approach we take in the following examples. If the space thus consumed is too great, it is possible to manage ksems dynamically at the user level, allocating them from a hash table on demand and deallocating them when empty. Such additional complexity may be worthwhile, but is beyond the scope of this dissertation.

Synchronization Mechanism	*See Page*
Counting semaphores	296
Readers/writers	298
Events	300
Barriers	302
Group lock	304

Table 37: Ksem-Based Synchronization Examples.

The examples we give use hybrid synchronization based upon a macro, `HBUSY(`*cond*`)`, which waits until *cond* is true or an implementation-defined maximum waiting time has been exceeded. (`HBUSY` might be implemented similar to the kernel's `BUSY_WAIT` macro, described in section 3.5.1.) The calls to `HBUSY` can be omitted to produce pure context-switching versions. There are many minor improvements applicable to these algorithms as well, such as two-stage implementations and check functions (§3.5);[162] we will spare the reader from such details, for the sake of brevity.

Counting Semaphores

The first example is counting semaphores, slightly fleshed out from the version given in section 7.3.2 on page 277:

```
struct csem {
      int v;      // User semaphore value
      int ks;     // Ksem; initialized to 0
}
```

```
int                                   void
P (struct csem *s)                    V (struct csem *s)
{                                     {
      HBUSY (s->v > 0);                     while (fai (&s->v) < 0 &&
      return fad (&s->v) > 0 ||                    ksemv (&s->ks, 1) > 0)
             ksemp (&s->ks) == 0;                continue;  // found hole
}                                     }
```

We made the user semaphore into a structure, added the ksem `ks`, and made changes to allow `P` to fail upon receipt of a user-caught signal[163] and to support conditional P. Condi-

---

[162] Of course check functions are of interest only for busy-waiting and hybrid versions, not pure context-switching ones.

[163] More specifically, the user's signal handler returns, which causes `ksemp` to return −1, with `errno` equal to `EINTR`. We are assuming `ksemp` is implemented with the `metasys` flag `SCALL__SYNC`; there are other possibilities as well (see section 4.3.1 on page 152).

tional P is a version of the standard semaphore P operation that returns *TRUE* if it succeeds without blocking, and *FALSE* otherwise; it is mostly used to prevent deadlock.

```
        // conditional P; returns true/false
int
try_P (struct csem *s)
{
        if (s->v <= 0)
                return 0;
        if (fad (&s->v) <= 0) {
                vfad (&s->ks);          // insert virtual hole
                return 0;
        }
        return 1;
}
```

For `try_P` we have apparently violated our own ksem design, by directly modifying the ksem value, `s->ks`, in user-mode. In this case the modification is equivalent to `ksemp` followed immediately by a system call cancelation. The initial test of `s->v` is akin to the initial test of the test-decrement-retest paradigm, TDR (§3.2), and is necessary to prevent overflow of `s->v` and underflow of `s->ks`. One should note that while `try_P` avoids blocking, and in fact avoids any system calls, it does cause a subsequent `V` to incur the cost of an additional system call if `try_P` fails.

Fairness. As pointed out in section 7.3.3, the `HBUSY` macro makes our counting semaphore algorithm on the previous page unfair, even if the ksems themselves are fair. One alternative is to add a ticket-based layer to the algorithm to make the behavior more nearly fair. Consider the following data structure:

```
struct fair_csem {
        unsigned int ticket;    // Next ticket to issue
        unsigned int serving;   // Biggest ticket being served
        struct csem uf[NCSEM]; // Unfair csems for waiting
}
```

We will show how the fields `ticket` and `serving` can be used with the array of unfair counting semaphores to lessen or eliminate unfairness.

```
void
fair_csem_init (struct fair_csem *s, unsigned int v)
{
        int i;
        s->ticket = 0;
        s->serving = v;
        for (i = 0;  i < NCSEM;  i++) {
                s->uf[i].ks = 0;
                s->uf[i].v = v / NCSEM + (v % NCSEM > i);
        }
}
```

```
int
Pfair (struct fair_csem *s)
{
     return P (&s->uf[ufai(&s->ticket) % NCSEM]);
}

void
Vfair (struct fair_csem *s)
{
     int i;
     do {
          i = ufai(&s->serving) % NCSEM;
     } while (fai(&s->uf[i].v) < 0 && ksemv(&s->uf[i].ks,1) > 0);
}
```

It would almost be possible to write `Vfair` in terms of `V`, e.g., as

```
V (&s->uf[ufai(&s->serving) % NCSEM]);
```

but premature unblocking wouldn't be handled right (except in the degenerate case when `NCSEM` is 1).

When `NCSEM` > 1, ticket holders are divided into groups as widely-separated as possible, thus reducing the number of competitors for the unfair semaphores, and improving fairness. It is difficult to precisely characterize the relative fairness of this ticket-based algorithm compared to the original. It is completely fair if both of the following restrictions hold:

- The value of `NCSEM` is at least as large as the maximum number of cooperating processes (i.e., at most one process is waiting at a given semaphore).
- We ignore premature unblocking.

Other situations will produce intermediate degrees of fairness.

Aside from the extra shared memory reference on each operation, the major disadvantage of this algorithm is the space required for the unfair semaphores. These may be acceptable in some situations, but probably not if both `NCSEM` and the number of fair semaphores must grow with the size of the machine.

Readers/Writers Locks

This example is an enhanced version of that presented in section 7.3.2 on page 277. The same approach is taken as for counting semaphores, on page 296.

```
struct rwlock {
     int c;          // counter
     int rr;         // running readers
     int rks;        // readers ksem
     int wks;        // writers ksem
};
```

```
int
rlock (struct rwlock *rw)
{
     HBUSY (rw->c > 0);
     return fad (&rw->c) > 0 ||
            ksemp (&rw->rks) == 0;
}

int
try_rlock (struct rwlock *rw)
{
     if (rw->c <= 0)
          return 0;
     if (fad (&rw->c) <= 0) {
          vfad (&rw->rks);    // insert virtual reader hole
          return 0;
     }
     return 1;
}

void
runlock (struct rwlock *rw)
{
     if (fai (&rw->c) < 0 && fai (&rw->rr) == BIG-1) {
          // last reader wakes writer
          rw->rr = 0;
          if (ksemv (&rw->wks, 1) > 0)
               wunlock (rw);    // virtual writer hole
     }
}

int
wlock (struct rwlock *rw)
{
     HBUSY (rw->c == BIG);
     int x = faa (&rw->c, -BIG);
     if (x == BIG)
          return 1;
     if (x > 0 && faa (&rw->rr, x) == BIG-x) {
          rw->rr = 0;
          return 1;
     }
     return ksemp (&rw->wks) == 0;
}
```

```
int
try_wlock (struct rwlock *rw)
{
      if (rw->c != BIG)
            return 0;
      int x = faa (&rw->c, -BIG);
      if (x == BIG)
            return 1;
      if (x > 0 && faa (&rw->rr, x) == BIG-x) {
            rw->rr = 0;
            return 1;
      }
      vfad (&rw->wks);
      return 0;
}

void
wunlock (struct rwlock *rw)
{
again:
      int x = faa (&rw->c, BIG);
      if (x <= -BIG) {
            // Wake another writer
            if (ksemv (&rw->wks, 1) > 0)
                  goto again;         // virtual writer hole
      }
      else if (x < 0) {
            // Wake all readers
            x = ksemv (&rw->rks, -x);
            if (x > 0 &&              // virtual reader holes
                 faa (&rw->c, x) < 0 &&
                 faa (&rw->rr, x) == BIG-x) {
                  // last reader wakes writer
                  rw->rr = 0;
                  if (ksemv (&rw->wks, 1) > 0)
                        goto again;   // virtual writer hole
            }
      }
}
```

The last 9 lines of `wunlock` are just the `runlock` algorithm, expanded inline with a `goto` for tail recursion and generalized to release multiple locks. (Since `runlock` calls `wunlock`, it is important that `wunlock` not actually call `runlock`: to do so would invite possibly deep recursion).

Events

We can easily implement an event mechanism similar to that used in the kernel (§4.5/p166). The operations supported are *wait* for an event, *signal* that an event has occurred, and *reset* an event. Once an event has been signaled, further signals are ignored. Once an event has been reset, further resets are ignored.

The algorithm we present here works by having a counter that is incremented by 1 by a waiting process and set to a large constant (with Fetch&Store) by a signaling process.

```
struct event {
      int c;        // counter
      int rmerge;   // merge resets
      int ks;       // ksem for waiting
};
```

The counter `c` should be initialized to 0 for an initially-reset event, or to `BIG` for an initially-signaled event. The value of `BIG` must be larger than the maximum number of waiting processes (including the cumulative number of prematurely unblocked processes), but small enough that each processor can increment it by 1 without overflow.

```
      // wait for event
      // return false if prematurely unblocked
int
ewait (struct event *e)
{
      HBUSY (e->c >= BIG);
      return e->c >= BIG ||
             fai (&e->c) >= BIG ||
             ksemp (&e->ks) == 0;
}
```

It is important that `ewait` test the counter before incrementing it; otherwise the counter might overflow (imagine a process calling `ewait` from within a loop with no `ereset` occurring for a long time).

```
      // signal that an event has happened
void
esignal (struct event *e)
{
      int c = fas (&e->c, BIG);
      if (c > 0 && c < BIG)
           ksemv (&e->ks, c);
}
```

By using Fetch&Store instead of Fetch&Add, `esignal` doesn't have to worry about overflow; however, this assumes that Fetch&Increment and Fetch&Store are atomic with respect to each other (the mechanism of section 3.1 on page 43 could be used to make this more robust). The return value from the Fetch&Store is used to detect redundant `esignal` calls as well as to determine the number of processes to wake up with `ksemv`.

```
      // reset event
void
ereset (struct event *e)
{
      if (fas (&e->rmerge, 1) == 0) {
            if (e->c >= BIG)
                  e->c = 0;
            e->rmerge = 0;
      }
}
```

The reset operation sets the counter to 0, but only if the event was in the signaled state. To ensure this, concurrent `ereset` calls are merged, so that only one has effect, using the `rmerge` field of the event. Once the one has been identified, there is no possibility of the event changing state before the counter is reset to 0 by the statement `e->c = 0`.

Barriers

We can implement a barrier using two counters and two ksems. Dimitrovsky presented a busy-waiting barrier algorithm using only a single counter [62, 65], where two different values (N and $2N$, for N participants) are used to implement the barrier threshold for alternate barriers. We cannot use such a strategy to construct a context-switching barrier based on semaphores, because semaphores are oriented around the value 0 only. Instead, we will use a two-counter approach, similar to Dimitrovsky's presentation [62] of Rudolph's busy-waiting barrier [174].

```
struct barrier {
      int phase;      // indicates which counter to use
      int c[2];       // counters
      int k[2];       // ksems for waiting
};
```

All of the structure members should be initialized to 0 before use. A barrier is accomplished by incrementing a counter. All but the last process to do so then wait on the corresponding ksem; the last one signals the others to proceed:

```
        // wait at barrier
        // n is the number of participants
void
barrier_wait (struct barrier *b, int n)
{
        int myphase = b->phase;
        if (fai (&b->c[myphase]) == n-1) {  // last at barrier
                b->c[myphase] = 0;
                ksemv (&b->k[myphase], n-1);
        }
        else {
                HBUSY (b->c[myphase] == 0);
                if (b->c[myphase] == 0)
                        vfad (&b->k[myphase]);    // spinning paid off
                else
                        ksemp (&b->k[myphase]);   // must block
        }
        b->phase = (myphase+1) % 2;
}
```

We have declined to handle premature unblocking in this version, both for the sake of brevity and because there is no clear need for it.[164] We have chosen to specify the number of participating processes with an argument to every `barrier_wait` call; this is somewhat cheaper than storing it in the structure (as in the Symunix-1 busy-waiting barrier described in section 7.2.4), but more error-prone, since the effect of calling `barrier_wait` with different values of `n` is not well defined.

This algorithm uses a more complicated approach to hybrid synchronization than the other synchronization forms we've presented. The problem is that all participants in a barrier must perform an action before any can proceed, thus it is not helpful for any to delay before acting. But if we are to gain by busy-waiting after affecting the barrier, we need a cheaper way to compensate for `ksemp` than by calling `ksemv`; fortunately, the definition of ksems allows this to be done by simply decrementing the semaphore directly. To convert this algorithm to a pure context-switching version, simply remove the four lines beginning with the call to `HBUSY`.

There is a subtle issue with the placement of the statement that modifies `b->phase`. The code sample above shows the statement as placed by Dimitrovsky [62], and causes the location to be updated to the same value by all participating processes, i.e., it will work well

[164] Premature unblocking is often an error indication, for which a reasonable response is to return an error indicator to the caller. This is all well and good for lock-type synchronization methods, but it isn't clear what should happen when this occurs at one of many barriers in a sequence. How should the "failing" process "clean up" before returning? How should the remaining barriers and cooperating processes behave? These questions have much to do with the semantics of programming with barriers and little to do with the detailed mechanics of premature unblocking. The group lock, which we will deal with on page 304, presents a barrier-style framework in which some of these questions are more easily addressed.

on a CRCW PRAM or a real machine with combining of stores. An alternative is to put the statement immediately before the statement `b->c[myphase] = 0`; this causes the location to updated by only one process, but lies on the critical path to releasing the waiting processes. By using `COMBINE_STORE` (see section 3.1 on page 42), we can write the code to handle either situation:

```
....
if (fai .... ) {
     if (!COMBINE_STORE)
          b->phase = (myphase+1) % 2;
     ....
}
else {
     ....
}
if (COMBINE_STORE)
     b->phase = (myphase+1) % 2;
```

Since the macro `COMBINE_STORE` reduces to a constant, the new if statements will be evaluated by the compiler, which will generate code for only one or the other `b->phase` assignments with no run-time overhead.

Group Lock

In section 3.5.6 we described a busy-waiting algorithm for group lock; Table 38, below, lists the functions we now present for group lock using ksems. The algorithm is based on tickets for group formation, and implements a conservative form of *fuzzy barrier*. In the original definition of fuzzy barrier (Gupta [97]), a single barrier is split into two operations: no process may proceed past the second part until all processes have reached the first one. Computations that can safely occur before or after the barrier may be placed between the two parts, thus providing an opportunity to hide the latency of the last process reaching the barrier and reduce overall waiting time. This is illustrated in Figure 11 (**A**), on the next page.

In the group lock algorithm we now present, a more conservative barrier implementation is used: a process is delayed upon executing either part 1 or part 2 of the barrier if any process has not yet begun to execute the other part. As a special case, the algorithm behaves as if the `glock` operation itself qualifies as a barrier, part 2. This is illustrated in Figure 11

Function	*See Page*	*Purpose*
`glock`	306	Enter group
`gsize`	308	Return group size
`gf1barrier`	309	Fuzzy barrier, part 1
`gf2barrier`	310	Fuzzy barrier, part 2
`gunlock`	310	Leave group

Table 38: User-Mode Context-Switching Group Lock Functions.

(**B**).

The group lock algorithm we present here allows processes within the group to execute different numbers of barriers before leaving the group, but every part 1 must be paired with a part 2. In other words, it is forbidden to leave the group *between* executing part 1 and part 2. If all processes were required to execute the same number of barriers before leaving the group, the algorithm could be altered to achieve a slight performance gain.

For simplicity, this ksem example does not support premature unblocking. In order to support premature unblocking, decisions would have to be made about how to handle failure at a barrier.

A group lock is represented by a structure:

```
struct glock {
     int ticket, size, exit;  // dynamic group control
     int c1, c2;              // barrier counters for parts 1 & 2
     int w0, w1, w2, ws;      // ksems for waiting
};
```

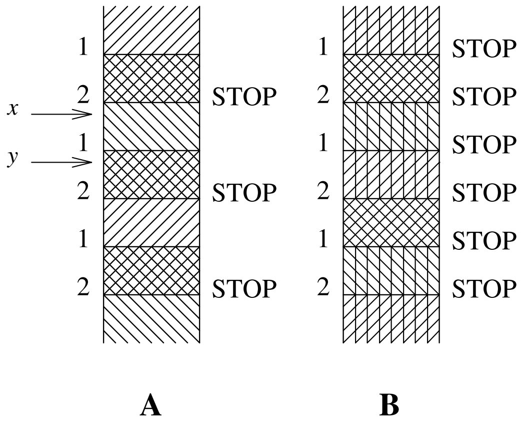


Figure 11: Fuzzy Barriers.

(**A**) depicts the original fuzzy barrier of Gupta [97], and (**B**) depicts our more conservative form. Horizontal lines indicate barriers (either part 1 or 2, as marked on the left). Participants execute code from top to bottom, with possible stopping points indicated on the right side of each diagram. Each diagonal and vertical pattern sweeps out the extent over which other participants may be executing concurrently from any one point. For example, a participant in (**A**) executing at the point indicated by x must expect others to be executing concurrently from the previous 1 through the following 2. At the point indicated by y, the scope expands considerably, from the second previous 1 to the second following 2. In (**B**) the scope is unchanged for x, but much narrower for y. In fact, for (**B**), the scopes of x and y are the same size, each 3 code blocks. This compares to as many as 5 code blocks for (**A**).

All structure fields should be initialized to zero except `c1`, which should be initialized to `BIG`, a number greater than the maximum number of processes.[165] This algorithm implements what we call an *open door* policy for group formation: membership remains open until something happens that requires the group size to be fixed, i.e., an explicit size request (`gsize`), part 1 of a barrier (`gf1barrier`), or a process trying to leave the group (`gunlock`). In contrast, the busy-waiting algorithm in section 3.5.6 implements a *closed door* policy: only those already waiting are granted admission during group formation. Open and closed door policies are depicted in Figure 12, on the next page.

The algorithm is ticket-based, and works in the following general way: To join a group, increment `ticket`, with Fetch&Increment. If the old value was less than `BIG`, entry into the group is granted and no blocking is required. When the group membership is fixed, `ticket` is replaced with a power-of-two constant, `BIG`, larger than the maximum group size (e.g., the maximum number of processes), using Fetch&Store.

```
      // enter group
      // returns index of process within group (0 origin)
int
glock (struct glock *g)
{
      HBUSY (g->ticket < BIG);
      int t = fai (&g->ticket);
      if (t >= BIG)
           ksemp (&g->w0);
      return t % BIG;
}
```

The structure's `size` field contains not only the group size, when fixed, but also provides some synchronization for fixing the size. Specifically, the first process to observe a zero `size` field and to increase `size` by `BIG` is the one selected to fix the group size and close the membership. Since `BIG` is a power of two, we are actually using the high order bits of `size` as an atomic flag. Those processes not selected to fix the group size must wait. The number of such waiters is determined by adding up their `BIG` contributions to `size` (but the `BIG*BIG` value, also a power of two, must not be counted; it is used in `gf1barrier`, on page 309).

[165] To prevent overflow, we also require $2 \times \texttt{BIG}^2 \leq 2^{wordsize}$, as we will show on page 308.

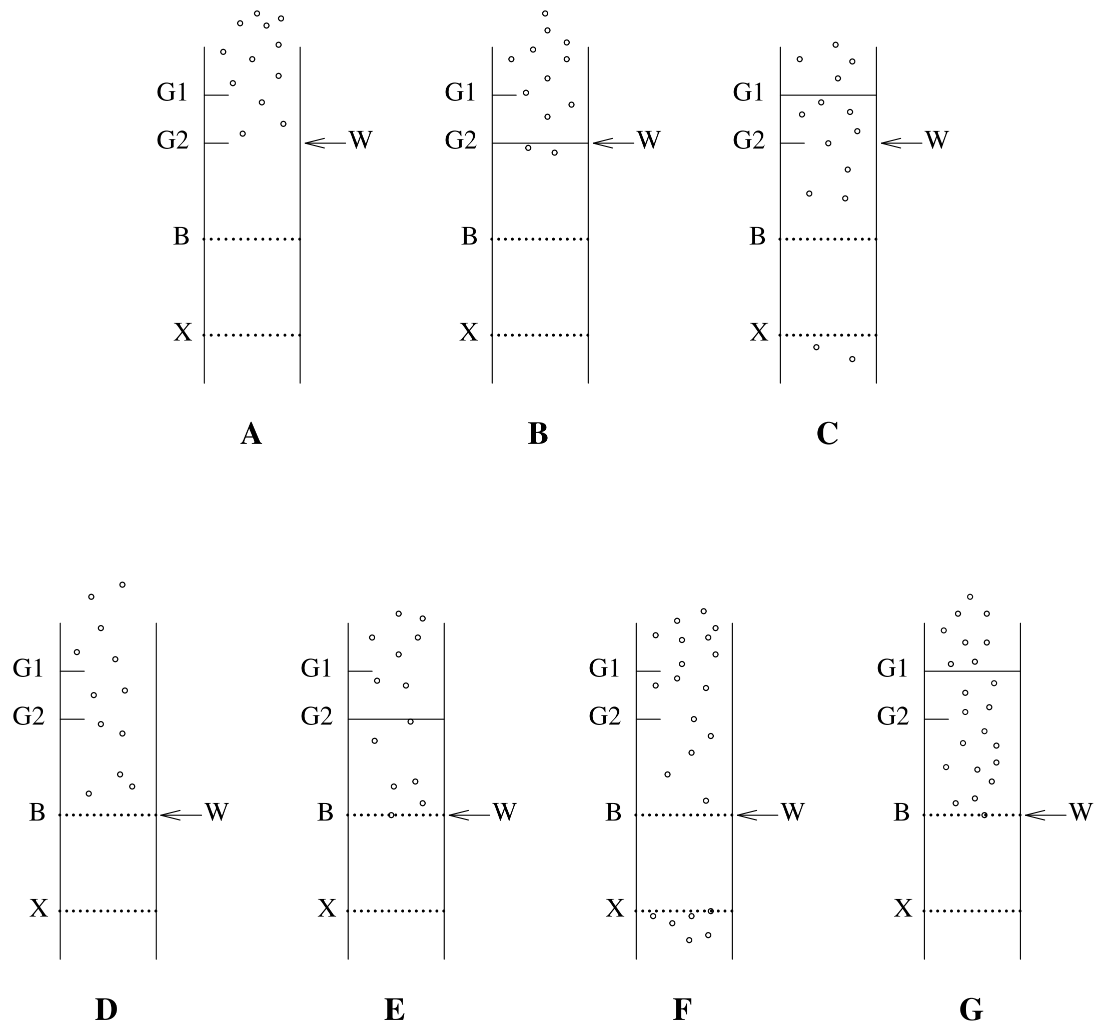


Figure 12: Closed and Open Door Policies for Group Lock.

Group participants, indicated by tiny circles, proceed from top to bottom. **A**, **B**, and **C** depict a closed door policy, while **D**, **E**, **F**, and **G** depict an open door policy. We show three controls for group formation: A two-part gate, marked with G1 and G2, and a watch point marked with W. Admission to the group is sealed when the first participant reaches the watch point. B is a barrier and X is the group lock exit. In **A** and **D**, new participants arrive at a group lock not currently in use. As soon as one passes W, group formation is complete, shown in **B** and **E**. It is very unlikely for the group in **B** to be as large as the one in **E**. When the last group member exits the lock, membership of the next group is sealed instantly under the closed door policy shown in **C**, but the open door policy keeps the group open longer, through **F**, until finally being closed in **G**. (See the algorithms on pages 70 and 306, respectively.)

```
int
gsize (struct glock *g)
{
      int s = g->size;
      if (s % BIG == 0) {
            if (s > 0)
                  HBUSY (g->size % BIG > 0);
            s = faa (&g->size, BIG);
            if (s == 0) {
                  s = fas (&g->ticket, BIG);
                  int w = faa (&g->size, s);
                  w = (w / BIG) % BIG;
                  if (w > 0)
                        ksemv (&g->ws, w);
            }
            else if (s % BIG == 0) {
                  ksemp (&g->ws);
                  s = g->size;
            }
      }
      return s;
}
```

The beginning of barrier part 1 also participates in determining the group size, but doesn't actually have to wait for the value to be known. For this reason, it uses Fetch&Or to set a high bit in `size` instead of using Fetch&Add when vying to be the one to set the size; this is why `%` is used in `gsize` when computing the second argument for `ksemv`. Our use of Fetch&Or requires that it be atomic with respect to Fetch&Add (§3.1/p43), but it also helps avoid overflow because `size` never has a value greater than $2 \times \mathtt{BIG}^2 - 1$ (using Fetch&Add, the maximum value would approach $\mathtt{BIG}^3$).

```
        // fuzzy barrier, part 1
        // returns index of process within pre-barrier group (0 origin)
int
gf1barrier (struct glock *g)
{
        if (g->size == 0) {
                if (faor (&g->size, BIG*BIG) == 0) {
                        int s = fas (&g->ticket, BIG);
                        int w = faa (&g->size, s);
                        w = (w / BIG) % BIG;
                        if (w > 0)
                                ksemv (&g->ws, w);
                }
        }
        HBUSY (g->c1 >= BIG);
        int c = fai (&g->c1);
        if (c < BIG)
                ksemp (&g->w1);
        else if (c % BIG == (g->size % BIG) - 1) {
                int e = fas (&g->exit, 0);
                // adjust group size after members have left
                if (e > 0 && faa (&g->size, -e) % BIG == e) {
                        g->c1 = BIG;
                        int t = faa (&g->ticket, -BIG);
                        t %= BIG;
                        if (t > 0)
                                ksemv (&g->w0, t);
                }
                else {
                        g->c1 = 0;
                        int w = faa (&g->c2, BIG);
                        if (w > 0)
                                ksemv (&g->w2, w);
                }
        }
        return c % BIG;
}
```

The rest of the barrier part 1 and most of the barrier part 2 deal with the fields `c1` and `c2`. One of these fields is incremented with Fetch&Increment upon reaching the corresponding barrier part, and the old value indicates whether blocking is required. When the last process reaches a barrier part, determined by looking at `size` and the old value of `c1` or `c2`, `BIG` is added to the *other* barrier part counter, and the old value of that indicates how many waiting processes to wake up with `ksemv`.

```
        // fuzzy barrier, part 2
        // returns index of process within group (0 origin)
int
gf2barrier (struct glock *g)
{
        HBUSY (g->c2 >= BIG);
        int c = fai (&g->c2);
        if (c < BIG)
                ksemp (&g->w2);
        else if (c % BIG == (g->size % BIG) - 1) {
                g->c2 = 0;
                int w = faa (&g->c1, BIG);
                if (w > 0)
                        ksemv (&g->w1, w);
        }
        return c % BIG;
}
```

In contrast to the barrier algorithm on page 303, this conservative fuzzy barrier approach more naturally matches the hybrid synchronization technique we advocate (§7.3.3).

The major remaining wrinkle is `gunlock`. Since a process may leave the group anywhere between barrier parts 2 and 1, `gf1barrier` is called to check the `exit` field, accounting for all exits and adjusting the group size. We implement `gunlock` mostly in terms of `gf1barrier`, even though it can force logically unnecessary waiting.

```
        // leave group
void
gunlock (struct glock *g)
{
        vfai (&g->exit);
        gf1barrier (g);
}
```

7.4. Lists

Ordinary list algorithms are as applicable to user-mode as to kernel-mode. But although the basic design of highly parallel list algorithms is sensitive primarily to the target machine architecture, some aspects are sensitive to operating system features and characteristics as well (see the issues discussed in section 3.7.1 on page 78).

By far the biggest operating system sensitivity for user-mode lists is internal synchronization. In the kernel environment, where the programmer has greater control, busy-waiting is generally preferred for internal synchronization in list algorithms because it has the lowest overhead. But the user-mode environment is not always so friendly to the programmer, chiefly due to various forms of preemption. Techniques for employing busy-waiting in user-mode were described in section 7.2. Internal synchronization may also be handled by context-switching, if properly supported, as described in section 7.3.

The user-mode environment can also present some advantages, such as a more dynamic memory map, which can make some algorithms more attractive. For example, Wood [207] gets good performance with an algorithm based on extensible hashing; this requires dynamic allocation of large new sections for the hash table.

7.5. Thread Management

Sections 4.2 and 4.3 described the Symunix-2 kernel interface for process management, based upon asynchronous system calls, page faults, and some careful semantics for signals. Flexibility, performance, and allowing the user program to maintain as much control as possible were the driving goals of the design. While the system calls and other facilities described can be used directly, the plan was always to support a thread management layer at the user level. The design of user-mode thread management systems is the subject of this section.

In principle, one can go about designing a user-mode thread system for Symunix-2 in much the same way as for a bare machine. The kernel provides analogues for each hardware component:

processor	process
memory	file
MMU	mapin/mapout/mapctl
interrupts/exceptions	signals
I/O devices	files/devices/sockets
real-time clocks	timers

Many other operating systems provide similar correspondences with, of course, many significant differences in the way the various features interact. Because of the high semantic level provided by the kernel, practical design of user-mode thread management is somewhat different from process management inside the kernel. Some major issues for real operating systems may be handled very differently for user-mode thread management:

- *Machine control.* The kernel provides a much higher level of service than the bare hardware, including substantial device independence. In addition, many of the Symunix-2 kernel services were designed with support for user-mode thread systems specifically in mind.
- *Protection.* Multi-user operating systems are primarily concerned with issues of resource sharing and protection. Even single-user systems that support multiprogramming often provide more independence between jobs or processes than would be appropriate for a user-mode thread system.
- *Fairness.* Conventional multiprogrammed systems are usually concerned with providing some measure of fairness for the processes being scheduled (sometimes subject to external priorities). Complicated heuristic methods are used to balance conflicting goals of interactive response time, throughput, and job turnaround. Resource exhaustion, causing conditions such as thrashing, further contribute to the problem. In contrast, user-mode thread systems need only provide limited resource management for a single application.
- *Generality.* Complete general-purpose operating systems for real machines attempt to support a very wide class of applications. Many tools are provided or supported to aid in programming, debugging, and using many different kinds of applications. User-mode thread systems are more like special purpose operating systems, and each major parallel

programming style or environment can have its own.

In some respects, user-mode thread systems are similar to the kind of simple operating system core that used to be familiarly known as a *monitor*. Aside from miscellaneous feature improvements, such as the availability of sophisticated file systems and networking capabilities from the host operating system, the thing that best distinguishes modern user-mode thread systems from monitors is the explicit focus on using, and supporting the use of, parallelism and shared memory.

7.5.1. Design Space for User-Mode Thread Management

We present a spectrum of user-mode thread system design alternatives, beginning with the simplest forms and advancing to progressively more sophisticated and complex designs, each more capable than the one before. This does not mean, however, that only the most advanced designs are worth serious consideration. In general, the best performance will be achieved by the simplest approach meeting the requirements of a particular application and operating environment.

The simplest form of user-mode thread system is built around the idea that threads are initiated and run until completion. Thus there is no general context-switch, only initiation and termination. The list of threads to be run will be very compact if most such threads are identical, as commonly happens when parallelizing loops. Any synchronization is of the busy-waiting variety. Any traps, exceptions, signals, or system calls are completed without blocking or suffer the loss of the affected process for the duration of such blockage. Run-to-completion is not suitable for threads with extensive I/O and is not well supported by an operating system environment with significant demand paging or timesharing. Nevertheless, there are some situations where this kind of approach is very effective. The big advantages of a run-to-completion design are efficiency and ease of implementation: overhead is cut to the bone, and many difficult features are omitted.

The next big step up from run-to-completion thread systems is non-preemptive scheduling of equal priority threads. This adds the ability of threads to perform context-switching synchronization, and requires a general scheduler with ready and waiting lists for threads. Thread systems in this style suit a very large class of problems, and still offer reasonably good efficiency (certainly much higher than kernel-based thread management).

Thread systems such as we have described so far suffer a loss of performance when they encounter blocking system calls, demand paging, and timesharing; the first of these is addressed by the next step up in thread system design. The idea is to allow a thread context-switch instead of a process context switch when dealing with a system call that must block. Not only are thread context-switches cheaper, but the thread system maintains greater control over its processors compared to letting the kernel choose another process to run.

All the thread system alternatives described so far are *synchronous*: thread context-switches take place only when specifically allowed by the thread system, e.g., when threads must block for synchronization or I/O. The next step up is to support various forms of kernel-initiated preemption, brought about when the kernel delivers a timer interrupt, the process encounters a page fault, or the kernel preempts a process. The signal delivery mechanism provides much of the additional support needed to make these forms of thread preemption work (e.g., saving and restoring temporary registers). Thread systems at this level of complexity address the needs of the vast majority of non-real-time applications, and completely eliminate the need to employ more processes than processors. (Using more processes than processors is a "poor man's" way of maintaining processor efficiency in the face of blocking system calls and page faults. In addition, a thread system that is oblivious to kernel-initiated process preemption will naturally be employing too many processes when the system is under heavy load, a situation that merely slows things down further.)

Really difficult requirements can lead to adding features like priorities, priority-based preemption, and complex interactions between threads (or between threads and the outside world). These things can still be done in user-mode, but their additional complexity adds to overhead and hurts performance compared to the simpler approaches. For the sake of brevity, we refrain from exploring these more demanding requirements in this dissertation.

7.5.2. Examples

This section shows how several different levels of complexity in thread management can be handled. We discuss the following progressively more elaborate solutions, as outlined in section 7.5.1:

- Bare-bones threads that run to completion (below).
- Threads that block for synchronization (p314).
- Threads that block for asynchronous system calls (p315).
- Threads that can be preempted by a timer (p318).
- Threads that can be preempted by the kernel's process scheduler (p318).
- Threads that can be preempted by page faults (p319).

Run to Completion

An example of this kind of system is ParFOR (Berke [24, 25]). The thread state transition diagram, given in Figure 13, below, is trivial. Since these threads don't block, the thread control block degenerates to a template for thread initiation. Generally, threads are created by programming language constructs like the parallel loop, and so are all initiated in essentially the same way. For simplicity, we will assume run-to-completion threads are limited to implementing parallel loops, although other constructs can be handled similarly.

Since, in general, parallel loops can be nested, more than one may be active concurrently. The templates are kept in a list, maintained with a parallel algorithm such as those described in section 7.4. Note, however, that with run-to-completion semantics, there is no need to modify the list structure except when a new loop is encountered or when all the iterates of a loop are exhausted (a multiplicity count is lowered as each iterate begins), so a

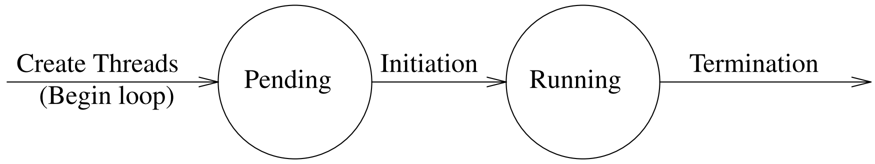


Figure 13: State Transition Diagram for Run-to-Completion Threads.

The *pending* state represents parallel loop iterates that have not yet begun execution.

relatively simple list algorithm may suffice, such as a linked list protected with a readers/writers lock.[166]

When a process is idle, it polls the template list until a template is found. Once a template is found, the process may require some initialization before it can begin executing loop iterates. In ParFOR, typically a few private variables are initialized by copying from the template, and the procedure call stack needs to be set up. Once this process initialization is complete, loop iterates (i.e., new threads) can be executed until they are exhausted. Advancing from one iterate to another within the same parallel loop is the most critical overhead factor, but requires only a handful of instructions, including a single Fetch&Decrement referencing shared memory to decrement the multiplicity.

Blocking for Context-Switching Synchronization

This is the simplest and most conventional form of thread blocking. It is, after all, a fundamental operating system concept taught in common OS texts (in the context of an OS kernel implementing processes, rather than a user process implementing threads). The factors unique to our situation are avoiding serial bottlenecks and creating multiple threads in one operation.

Because we want to create many similar threads in a single operation, we need a list of thread templates similar to the parallel loop templates we used on the previous page for run-to-completion threads. We also need a thread control structure for each thread that has started but not yet completed. The thread control structures are initialized from the templates. Thread control structures contain all thread state that must be saved when the thread isn't running, as well as pointers to any needed per-thread memory, such as the run-time stack and private variables.

Thread state transitions are shown in Figure 14, on the next page. Thread control structures move back and forth between a *ready list* and various *waiting lists*; they are on no list while actually running.

The ready thread list deserves a more highly parallel algorithm than that used for the pending thread template list, simply because of a higher anticipated rate of access. Suitable algorithms are discussed in section 7.4. No special operations are required, such as interior removal, but multi-insertion can be useful as we will see. Fair rather than FIFO ordering is usually adequate for the ready list. Ready list inserts are performed whenever a thread is awakened from the blocked state, deletions are performed by idle processes.

Each form of context-switching synchronization to be supported can use a different waiting list algorithm, if desired. For example, a binary semaphore is well served by a simple linked list protected with a busy-waiting lock, but a barrier is probably better served by a list with more parallelism, and by allowing multi-insert onto the ready list in the manner described in section 4.6.9.

As in a run-to-completion design, it is important to minimize the time from executing the last user instruction of one thread to executing the first user instructions of another as-yet-unexecuted thread from the same template. When a thread terminates, its original

---

[166] This is the solution adopted in ParFOR.

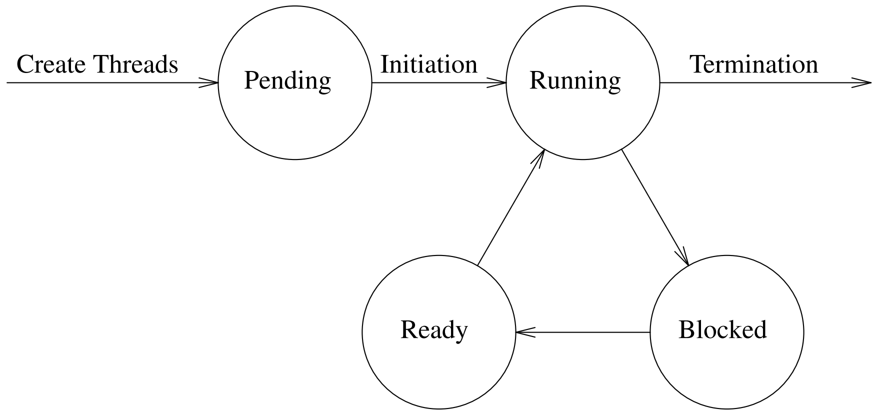


Figure 14: State Transition Diagram for Blocking Threads.

Threads move from *running* to *blocked* when they cannot proceed during synchronization (e.g., semaphore P operation), and they move back to *ready* when awakened (e.g., semaphore V operation).

template can be checked; most of the new thread's initialization overhead can be avoided by re-using the old thread. When the original template is exhausted, the ready list should be consulted before checking the template list to encourage depth-first execution of the logical thread tree. When completely idle, a process busy-waits, polling both the ready and the template lists.

Blocking for Asynchronous System Calls

By using the Symunix-2 asynchronous system call mechanism (§4.3.1), we can avoid blocking the process when a thread makes a system call, e.g., for I/O. For simplicity, we present the traditional synchronous system call model to the threads themselves.

The way it works is best described by example, shown in Figure 15, on pages 316 and 317. A typical system call function, such as `read`, issues a *meta system call*, using a control structure allocated on the thread's run-time stack.[167] In the fortunate case (e.g., the requested data was available in a system buffer without any physical I/O), the system call is fully completed without blocking, and the thread proceeds without delay. Otherwise, the meta call returns without blocking while the system call is executed asynchronously. When this happens, the `read` routine (part of the user-mode thread system) saves state in the thread control structure, and calls a dispatcher routine to consult the ready list for something else to do. When the asynchronous system call finally completes, a `SIGSCALL` signal is generated by the kernel. The thread run-time system includes a `SIGSCALL` handler that locates the thread (from the address of the meta system call control structure, provided by

[167] The meta system calls, `syscall`, `syswait`, and `syscancel`, and the `metasys` control structure are described in section 4.3.1 on page 151.

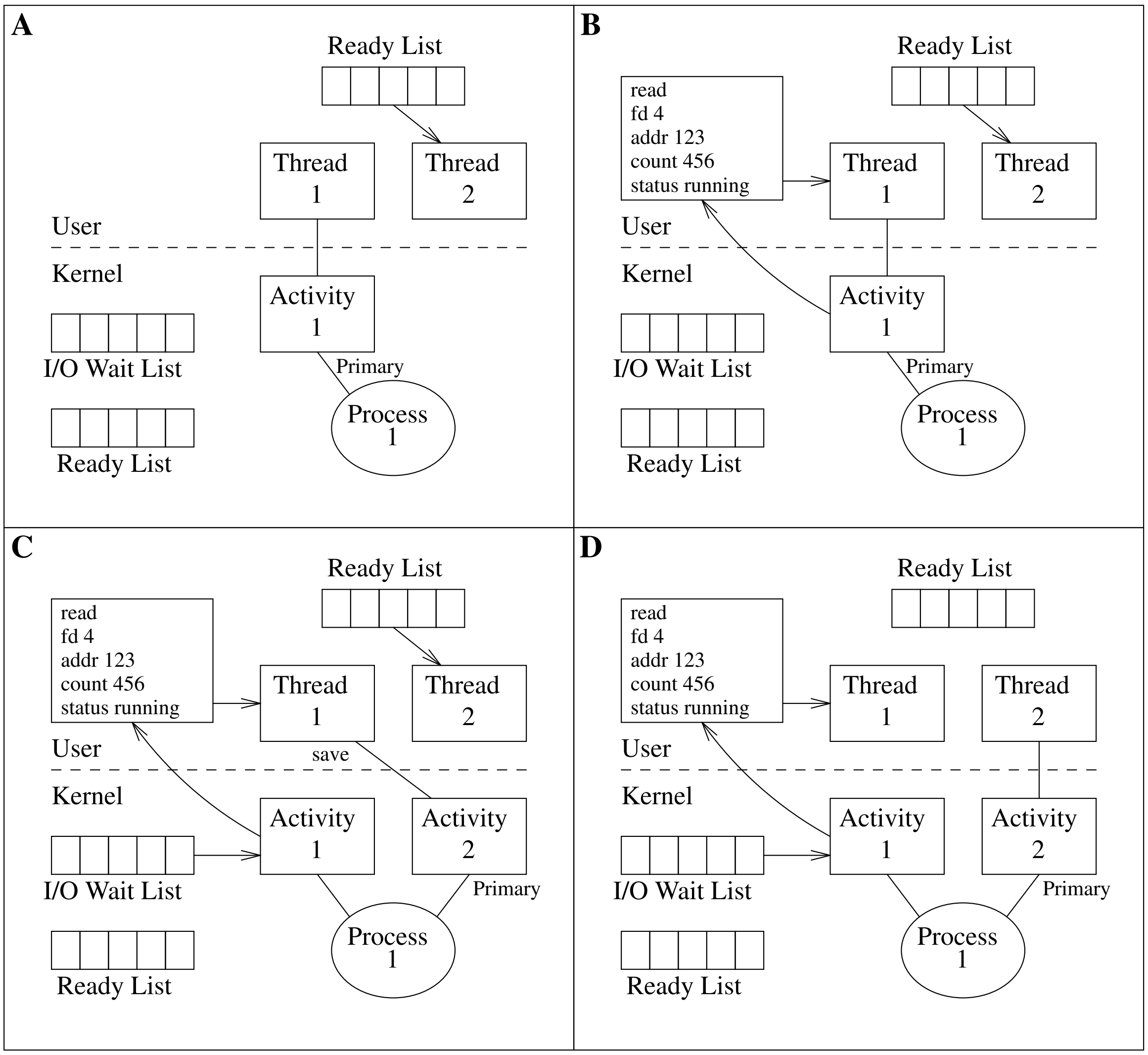


Figure 15: Example of Thread Blocking on Asynchronous System Call.

(A) Thread 1 is running in user-mode on Activity 1, Thread 2 is waiting to run. **(B)** Thread 1 calls `read`, which sets up a control structure and makes a meta system call. Activity 1 switches to kernel-mode. **(C)** In this example, the system call cannot complete without waiting for the disk, so Activity 1 blocks and creates Activity 2, handing off control of the processor to the new Activity. Activity 2 then "returns" to user-mode and saves Thread 1's state. **(D)** Activity 2 finds Thread 2 on the ready list and begins running it.

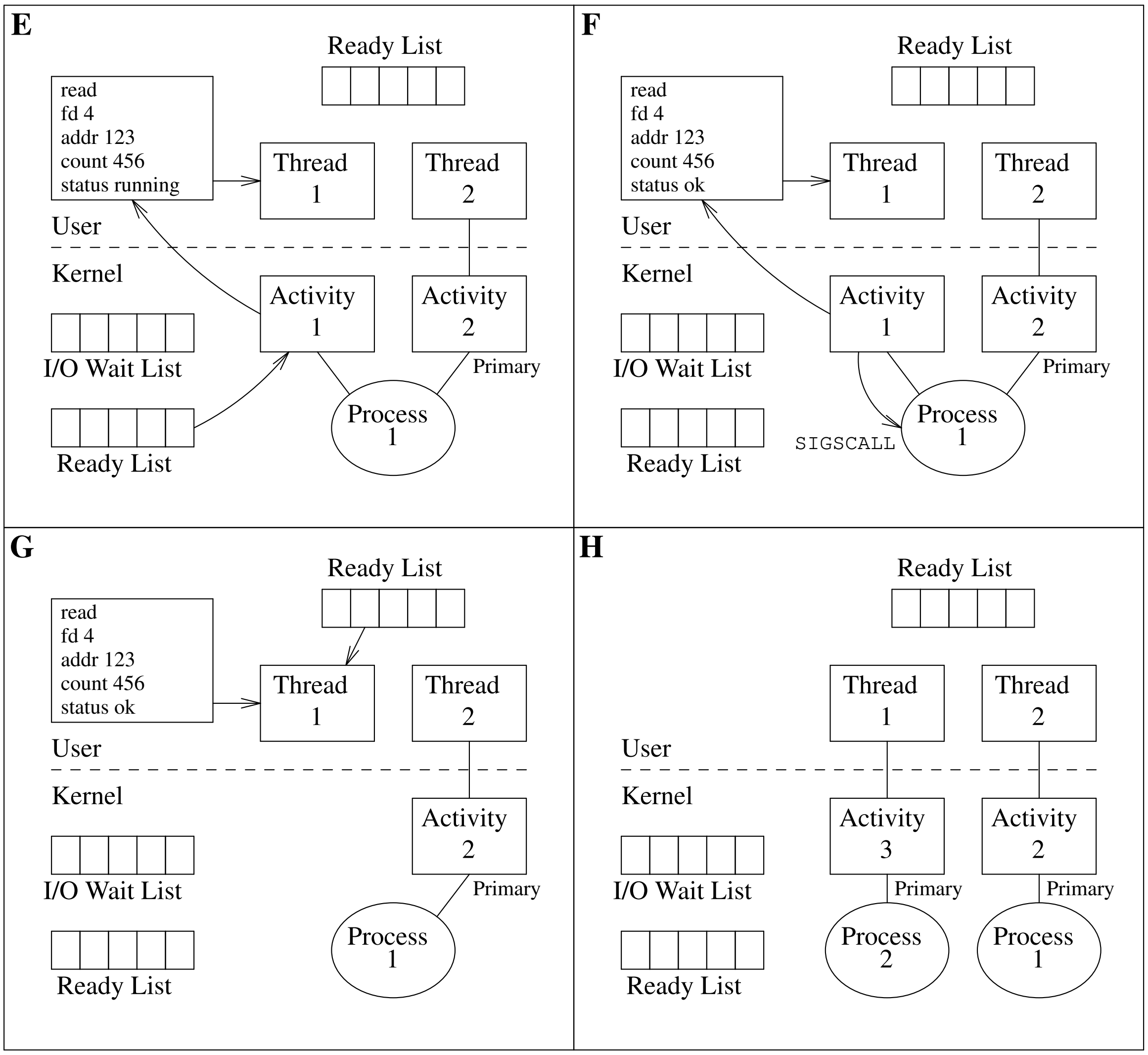


Figure 15: (Continued).

(E) The disk I/O completes and the interrupt handler puts Activity 1 on the system ready list, where it competes with Activity 2 and all other activities in the system (but with higher priority than a user-mode activity, such as Activity 2). **(F)** Activity 1 runs, possibly on a different processor than Activity 2. In this example, no further I/O is required to complete the read, so the control structure's status is updated and `SIGSCALL` is sent to the process. Activity 1 terminates. **(G)** Since Activity 2 is the primary activity for the process, it reclaims the structure for Activity 1 and calls a user-mode signal handler, passing the control structure address. The handler uses this address to find Thread 1, and puts it on the ready list. Thread 2 resumes execution when the signal handler is finished. **(H)** In this example, another process (2) with a different activity (3) running in user-mode finds Thread 1 on the ready list and begins executing it. Thread 1 returns from the original `read` call.

the kernel as an argument to the signal handler) and puts it on the thread ready list. When the thread is finally selected to run again, the `read` routine returns.

Preemption by Timer Expiration

Of course, preemption must be driven by something to be useful. A general priority scheme is a very powerful driving force, but perhaps the simplest possibility is timer-based preemption, such as might be used to implement round-robin execution of threads. Since timer signals are asynchronous, they can be delivered at almost any time. Additional thread state must be preserved compared to a synchronous context-switch; e.g., temporary registers and caller-saved registers must be considered live. The signal delivery mechanism itself is already required to save and restore this additional state, so the additional complexity of adding timer-based preemption is minimal.

The signal handler simply has to save any remaining thread state, put the preempted thread back on the ready list, and call the dispatcher to find another thread to run. Whenever a thread is run, the appropriate timer value must be set. (The overhead of setting the timer, normally a system call, can be mitigated by careful extension of the kernel/user interface, but we will refrain from exploring such solutions here.) Whenever the ready list or thread status are being modified, the timer signal must be masked[168] to avoid deadlock or damage to data structures.

Preemption by the Kernel's Process Scheduler

We have described three ways of coping with kernel scheduling decisions, temporary non-preemption, in section 4.6.4, scheduling groups (with policies such as group-preemption and non-preemption, in section 4.6.3), and the `SIGPREEMPT` signal, in section 4.6.5. Temporary non-preemption is useful around locking situations, e.g., to prevent preemption while a ready list lock is being held, while group- and non-preemptable scheduling policies are more expensive alternatives that attempt to avoid scheduling problems altogether. The `SIGPREEMPT` signal can be used in another way: to be responsive to scheduling in a way that keeps a user-mode thread system always in control of its threads and processors.

When a user-mode thread system runs on multiple preemptable user processes, the kernel's scheduling decisions can have a major impact. When the kernel preempts one of many processes in a parallel application, the thread being run on that process is also preempted, and can't run again until the same process is rescheduled. One way to avoid this is to use the `SIGPREEMPT` signal, described in section 4.6.5, which notifies a process shortly before preemption, giving it time to save thread state before yielding.

Handling of `SIGPREEMPT` is much like handling timer expiration. After saving thread state and inserting the thread onto the ready list, the handler calls `yield` to relinquish control of the processor (`yield` is described in section 4.6.4). An argument of 1 or more should be passed to `yield` to avoid continued competition with cooperating processes that have not been preempted. Eventually, when the process runs again, `yield` will return and the ready list can be examined for a thread to run.

---

[168] See section 4.8.1 for low overhead signal masking.

If the user-mode thread system were oblivious to preemption, we might expect an idle process to busy-wait until the ready list is non-empty. A better solution is to maintain a counting semaphore with the number of threads on the ready list (see section 7.3.3 for an efficient counting semaphore algorithm suitable for user-mode). After inserting a thread, a V operation increments the count, and a P operation insures that a deletion will succeed. The P operation becomes the point at which idle processes block.

Using this method of handling idle processes together with the suggested handling for `SIGPREEMPT` leads to a design where processes are either

- Contributing to the useful work of an application (running user threads),
- Executing user-mode thread overhead code,
- Waiting for work to do (blocked in P), or
- Preempted (ready to run, possibly with lowered external priority).

This assumes page faults either don't occur or are handled as suggested below. There are also some possible implementation details, such as secondary preemption (§4.6.5/p194), that can impose additional small delays, as well as unrelated interrupt activity that can delay a process.

Preemption Due to Page Faults

By using the Symunix-2 asynchronous page fault mechanism (§4.3.2), we can implement an efficient user-mode thread system that overlaps page fault latency with computation by other threads. Properly handling a `SIGPAGE` signal is a cross between timer-based preemption and asynchronous system call handling. As in timer-based preemption, a signal informs the user-mode thread system that the current thread cannot proceed at this time; the signal handler saves state and finds another thread to run. However, as with asynchronous system calls, the page fault handler must not immediately put the preempted thread back on the ready list.

A signal will be sent to notify the thread system when the thread may proceed, but this later notification is keyed to a user virtual address. In the case of system calls, the completion notification signal provides the handler with the virtual address of the completed control structure, which can point directly to the blocked thread. Page faults are more difficult because there is no user-visible control structure. The `SIGPAGE` handler is given two special arguments: the fault address and a boolean value to distinguish page fault initiation and completion. The first `SIGPAGE` inserts the thread into a search structure for later look-up (with the faulting address as key) when the second `SIGPAGE` is delivered. The design of a suitable search structure for fault address look-up depends upon the expected page fault rate. A highly concurrent structure, such as that described in section 3.7.6, is somewhat more complicated than a simple list with a lock, but will not serialize page faults.

As described so far, deadlock is possible. Recall that the user-mode ready list algorithm employs busy-waiting locks at some level (as many highly parallel list algorithms do). Suppose a thread blocks for some reason (e.g., a semaphore P operation), and the process encounters a page fault while holding a ready list lock, i.e., the faulting page is needed to complete a basic ready list operation. If we follow the general outline just given for asynchronous page fault handling, the `SIGPAGE` handler would ultimately need to access the ready list, and could encounter the same (already held) lock. Or it might get another page fault on the same page.

To prevent this unattractive scenario, we simply mask the SIGPAGE signal at certain times. If a page fault occurs when SIGPAGE is masked, it is simply handled in the old-fashioned, transparent, way: the process (i.e., the primary activity) is blocked for the duration. A SIGPAGE sent for page fault completion is simply delayed until the signal is unmasked.

The performance impact of masked page faults can be serious, so it is worth the effort to "wire down" certain relevant pages, to avoid such faults as much as possible. Unfortunately, this can have a significant design impact on the rest of the user-mode thread system. It is important to minimize the total amount of wired down memory, and the frequency of requests to change pageability. For this reason, thread control structures should be allocated from one or a small number of contiguous regions, and should contain all thread state (so that saving and restoring can be done without risk of page fault).

In order to handle signals without risk of page fault, a dedicated, wired down, signal stack should be used.[169] As a consequence, the thread system implementation becomes intimately connected with the signal delivery mechanism, as it must be able to copy the state saved by the signal delivery to the thread control structure. A better alternative would be to put the signal stack inside the thread control structure, and change it on every thread context-switch. (This could be done without system calls, by keeping the address of the signal stack in a user location known to the kernel.)

7.6. Virtual Memory Management

Section 5.2 described the kernel memory model, based on private address spaces supported by the operations `mapin`, `mapout`, `mapctl`, and some careful semantics for signals. Although usefulness was an important criterion in developing this design, it should be clear that higher level facilities are needed to provide a model better suited for most programming. These higher level facilities may be provided as part of an integrated parallel programming language environment, a simple set of library routines, or something in between. In this section we show how some reasonable higher level models may be constructed based upon the kernel model already presented. We will describe solutions for serial programs and for parallel programs with a shared address space model. In addition, because of the flexibility inherent in our approach, it is possible to tailor solutions to meet varying requirements. *Partially shared address spaces* are an example: aspects of the approaches used for serial programs and shared address spaces can be combined in a single parallel programming environment.

7.6.1. Serial Programs

The traditional UNIX process model, as exemplified by 7th Edition UNIX [21], provides a unique address space for each process, consisting of three user-visible segments:

---

[169] Traditionally, UNIX signals are delivered by growing the ordinary C language run-time stack. This works well most of the time, but there are situations where it can cause a problem, such as delivering SIGSEGV caused by stack growth failure. BSD4.2 UNIX introduced a *signal stack*, a dedicated memory area for the C run-time stack when handling certain signals. Although there are weaknesses in the details of the BSD signal stack mechanism (such as lack of ways to specify the size of the signal stack and check for overflow), the basic idea is sound, and is valuable for user-mode thread systems.

- *Text*, containing the machine instructions of the process. It is usually read-only and can also be used on many machines for read-only data, such as string constants.
- *Data*, readable and writable. Usually initialized data comes first, followed by uninitialized (implicitly zero) data. The size of the data segment may be changed dynamically, by extending or contracting it at the high end with the `brk` and `sbrk` system calls; this is generally used to implement a *heap*, e.g., the C function `malloc`.
- *Stack*, used for programming language run-time control and automatic variables. The stack segment is also dynamically grown at one end, as needed.

The vast majority of ordinary utility programs and most user applications are well served by this model, so it behooves any general purpose system to do a reasonable job of implementing it. In our case, there are two general approaches, both straightforward:

(1) The necessary support may be incorporated into the kernel. This approach is especially good for providing binary compatibility with another version of UNIX.

(2) The kernel memory model presented in section 5.2 is sufficient to support a user-mode implementation of the traditional model. The primary advantage of this approach is that it keeps the kernel slightly simpler.

We dismiss the kernel approach from further consideration without prejudice,[170] as it is dominated by machine dependencies and other implementation details. Furthermore, describing the user-mode approach does more to illuminate the use of the underlying kernel memory model.

Text The kernel `exec` system call sets up the text segment in a way that is suitable for just about any program, serial or parallel. The details depend on the file format used for executable programs, but a mapping is established from the user's virtual address space to an image file containing the text.[171]

Data The `exec` system call establishes a mapping for statically allocated data, by setting up an anonymous image file and initializing it (by reading initialized data from the executable file and by clearing uninitialized variables).[172] The ability to expand the data segment at one end is simple to provide: simply reserve enough virtual space for the segment to grow into, and translate `brk` and `sbrk` system calls to `mapin` and `mapout`, as needed, using additional space in the image file as the segment grows, and using less as it shrinks. The code implementing `brk` and `sbrk` must know the virtual starting address of the segment, the open file descriptor of the image file, and the initial size; if necessary, this information is obtained from `mapinfo`.

---

[170] It is a reasonable approach in practice, especially where binary compatibility is of value.

[171] It is reasonable for the executable object file format to specify the virtual address range to be used for each segment; this is done with the COFF file format [11] in the Symunix implementation for the Ultra-3 prototype. It is also possible for `exec` to use a predetermined constant virtual address for the text; this would be required if the executable file format is like the traditional UNIX *a.out* format.

[172] Again, the virtual address to be used for this mapping may be specified by the executable file format, or may be fixed to a predetermined value.

Stack The stack is handled in much the same way as the data segment, but the growth mechanism is different. The details are machine and programming language dependent, but traditional implementations depend either on a special system call, or handling the exception caused when the presently allocated stack area is exceeded; either approach may be used. In the former case, the system call is emulated by appropriate use of `mapin`, in the latter case, the exception generates a signal which is caught, and the signal handler issues a suitable `mapin`. In either case, overhead may be kept arbitrarily low by mapping in successively larger chunks as allocations are required. Traditional implementations never shrink the stack, but doing so is straightforward.

Nothing about this basic emulation scheme introduces serious overhead or implementation complexity, and it also is easily extended in useful ways. Traditional UNIX systems have been extended in several ways, over the years, to provide features such as shared memory and mapping of files to memory. Providing such features by using `mapin` and `mapout` in otherwise serial programs is not difficult.

Another extension, which has become quite common in recent years, is shared libraries. Discussing all the details and nuances of shared library implementations would be well outside the scope of this dissertation. Kernel extensions beyond shared memory are not typically needed to support shared libraries; however, support is generally required from the compilers, assemblers, link-editors, and other tools used in program development.

7.6.2. Shared Address Spaces

The simplest abstract parallel computer models don't include the concept of multiple address spaces (or even the concept of address translation), so it seems reasonable to omit it from a virtual machine designed to support parallel programs. This approach has been adopted by several research and commercial operating systems, and this section will discuss how it can be emulated by the kernel memory management facilities described in section 5.2.

It hardly matters whether the interface to be emulated is based upon mapping files to memory, anonymous memory allocation, or some other mechanism. There are some general problems that must be addressed by any shared address space system (whether it be implemented in the kernel or not):

- *Allocation*. The system must be able to assign the virtual address to use when allocating memory. Sometimes the user wants to provide the address, but generally the system must be able to make the decision.
- *''TLB'' coherence*. The TLB coherence problem arises in any machine that caches address translation information at each processor; there are many possible solutions, involving hardware, software, or both. When the address space is changed (i.e., a virtual→physical address mapping changes), some mechanism must ensure that the change is atomically seen by all processors. When providing a single shared address space by user-mode emulation atop multiple distinct address spaces, the hardware TLB is hidden and the problem could more aptly be called the *multiple address space coherence* problem.
- *Stack growth*. A parallel program generally requires many stacks, each growing to a size that is generally not predictable. The choice of mechanisms for stack growth are essentially the same as described in section 7.6.1 on this page for serial programs, but may be complicated by the problem of allocating space for multiple stacks in the address space (e.g., the address space may be small enough that each stack must have a discontiguous

allocation in order to support many stacks with sufficient growth potential).

In designing a user-mode shared address space emulation, it is useful to look at the kinds of changes that can happen to an address space, and identify them as *safe* or *unsafe* from a user-mode perspective. The safe changes are those that can't cause a serious problem, even if they aren't immediately propagated to all of the involved underlying private address spaces. A problem would be "serious" if it allowed a memory access to succeed even though it should be prohibited by the most recent address space change applicable to the virtual address in question. (The requirements described in section 5.2 on page 212 are stronger than this.) In general, there are only four operations we must consider:

Operation	*Safe?*
Relax Access Permission	yes
Restrict Access Permission	no
Allocate Virtual Memory	yes
Deallocate Virtual Memory	no

Kernel-based shared address space systems must consider other operations, such as page eviction and copy-on-write optimizations, but these are already handled by our underlying private address spaces, as described in section 5.3.

We continue by describing two separate designs, a minimalist approach and a more featureful one.

Minimalist Shared Address Space

We can most simply transform the traditional UNIX process address space into a shared address space by completely avoiding unsafe changes:

- A single access permission (read+write) is suitable for the entire data and stack segments. This avoids any unsafe operations that might restrict permissions.
- We eliminate the little-used ability to return memory to the kernel by using `brk` or `sbrk` to shrink the data segment at the high end. Instead, we only permit growth. Similarly, we never deallocate stack space. These restrictions completely avoid unsafe deallocation operations.

For the data segment, we only need to maintain the current "high water mark" of dynamic allocations in a shared variable. A signal handler for address exceptions (`SIGSEGV`) can `mapin` the part of the image file up to the current high water mark. Locking isn't even required, because the high water mark is monotonically increasing and modified only by Fetch&Add.

There are at least two ways to handle the stack growth:

(1) By adopting discontiguous stack growth, many stacks can be allocated from the same region, which grows at one end. As stacks shrink, some user-mode data structure must keep track of available stack space so it can be reused for future growth, since it isn't deallocated from the kernel's point of view. This approach suffers from fragmentation, and probably requires more overhead for stack growth and retraction.

(2) If the address space is large enough, large address regions may be statically reserved

for the maximum allowable number of cooperating processes.[173] This approach suffers because vacated stack space isn't pooled, and can't be used by another stack, thus wasting physical memory and/or backing storage. Allocating and deallocating whole stacks can be handled as needed by using an unordered list or set (§3.7.5) algorithm.

Regardless of which approach is taken, coherence can be maintained in much the same way as for the data segment. When a `SIGSEGV` is handled, the faulting address is available to the handler, and can be checked against the known bounds of the data segment and the allowable limits of the stack segment(s). The stack segment can be grown implicitly by such a fault, or a high water mark maintained by the code that sets up new activations records can be consulted. The signal handler can `mapin` additional memory to satisfy the faulting reference, or can treat the reference as an error.

It is probably most convenient to use a separate image file for the data segment and each stack segment.

Featureful Shared Address Space

A more featureful design surely includes the ability to deallocate memory, and may also support *sparse* address space management. In a sparse address space, there may be a variable (and possibly large) number of different regions, each with different attributes (image file, offset, access permissions, cacheability, etc.). Of course, some parts of the address space mapping are undefined, i.e., there are virtual *holes*, which are not mapped to anything.

A more general allocation scheme for virtual addresses may be needed than the one used in the minimalist design presented on the previous page. It could be as simple as searching a sorted linked list to implement a first fit or best fit strategy, or something like a buddy system allocator (§5.3.4/p244). It is important to consider the degree of concurrency expected on memory allocation or deallocation operations: is it acceptable to use a serial algorithm protected by a simple lock, or will such a simple approach lead to bottlenecks? The answer depends on the expected workload and other implementation details.

To solve the address space coherence problem, we assume a data structure is maintained in shared user memory so that the correct status of any allocated virtual address can be easily determined. This structure may (or may not) be tightly integrated with the virtual space allocator just described. Some reasonable structures include linear linked lists and trees, but of course there are many possibilities. In any case, the search and update algorithms must support concurrent use and, again, the best algorithm to use depends on the degree of concurrency expected. This structure, which we will call the *user-mode address map*, must produce at least the following information in response to a look-up for a given address:

- image file descriptor[174]

---

[173] This has been argued to be an advantage of 64-bit processors over 32-bit ones (e.g., Dewar and Smosna [55]).

[174] Depending on other implementation choices made, the image file name and/or PID of original mapping process may also be needed. This is because file descriptors are not shared between processes. The `superdup` system call, in section 5.2.7 on page 222, can be used to gain access to another process's file descriptors, or to add descriptors to a common process (e.g., the family progenitor, described in section 4.2 on page 149).

- image file offset
- `mapin`/`mapctl` permission and cacheability flags (§5.2.1/p215).

For good performance, the user-mode address map should keep track of contiguous address ranges with the same attributes, rather than individual pages.

Safe operations are easy to implement: just modify the user-mode address map and make the change to the address space of the current process by calling `mapin` or `mapctl`. When another process makes a reference to the affected address range, it will get a `SIGSEGV` signal which can be caught; the signal handler consults the user-mode address map and performs the appropriate `mapin` or `mapctl`.

Unsafe operations are trickier because of the need to maintain address space coherence. There are generally three strategies that can be used, together or separately, in this kind of situation, whether it be address space coherence, TLB coherence, or even hardware cache coherence:

(1) *Avoid the problem.* In some other circumstances (e.g., maintaining TLB coherence in the face of page eviction), it is possible simply to limit operations (e.g., page evictions) to those virtual addresses that aren't actually shared. This isn't possible for direct user requests, as is the case here, but the difficult part of address space coherence can still be occasionally avoided if the user-mode address map keeps a count of those processes that have mapped in the region. The operation can be done locally if the current process is the only one to have the relevant mapping. (This may be determined by calling `mapinfo` or by keeping track, either in the user-mode address map or in a private user-mode data structure. Care must be taken to avoid races with other cooperating processes, by using readers/writers locks if necessary.)

(2) *Delay the troublesome operation.* In some other circumstances, it may be possible to delay completion of the operation until it is known that all cached copies of the information being changed have been invalidated or otherwise expired, thus changing an unsafe operation into a safe one. This approach is only viable if the frequency of unsafe operations requiring delay is low, or if the expected delay is short. In the case at hand, it doesn't seem like a viable approach since there is no natural periodic invalidation mechanism. Furthermore, the cost of handling `SIGSEGV` and performing `mapin` is high enough that an artificial mechanism to unmap most shared memory periodically would have to be executed extremely rarely.

(3) *Perform a remote invalidation or update.* This approach, often called *shootdown* when invalidation is implemented in software, is the most general, but also the most costly, solution. It should be avoided whenever possible because of its potentially high cost. Messages of some sort must be sent to each site where vital information is cached, to force its invalidation. For address space coherence, some kind of asynchronous interprocess message must be used; in Symunix-2 a signal is appropriate (section 4.8.2 explains how a signal can be efficiently sent to an entire *family* of processes). The signal handler in each process makes the appropriate local address space change, and the process initiating the operation waits until they are all done. Implementation issues are discussed on the next page.

It should be clear that a user-mode shared address space emulation cannot beat the performance of a kernel- or hardware-based implementation, if the workload is dominated by unsafe operations leading to unavoidable shootdown. On the other hand, if such workloads are rarely encountered in practice, there should be little performance difference.

Furthermore, such workloads will perform even better if modified to require fewer unsafe operations or to run in a less featureful environment, such as we described on page 323. The existence of some workloads dominated by unsafe operations is not sufficient justification for a system design skewed in their favor; the case must be made that such workloads are truly “worthy” of such extensive support, i.e., that they cannot readily be altered to depend less on unsafe operations. We note that there are no standard shared address space models to which many important applications have been written.

The remainder of this section discusses implementation issues, primarily for handling shootdown.

When performing a shootdown, the affected address range must be communicated to the signal handlers. If mutual exclusion is used to serialize all shootdowns, a single shared variable dedicated to the purpose suffices.

The signal used for shootdown must not be one that is otherwise used by the system. Semantics of the Symunix-2 signal mechanism ensure that no affected process will miss the signal completely, but waiting until they have all handled it is slightly tricky. As usual, there are several possible implementation approaches. A simple one is to set a counter to the number of processes in the family and have the signal handler decrement it; when it reaches zero the time for waiting is over. This works only if shootdowns are completely serialized, so that only one is pending at a time, and process creation/termination within the family is postponed during a shootdown (a readers/writers lock suffices for this). Waiting for the counter to be decremented to zero can be done by busy-waiting or context-switching.

It is possible to devise more elaborate solutions, such that shootdowns are not serialized unless they affect the same address region. Such solutions can get quite complicated, but not primarily for any reasons unique to user-mode implementation A kernel-based shared address space implementation that avoids serialization will also be complex. In either case, it isn’t clear that such solutions are worthwhile, especially compared to the alternative of simply prohibiting or avoiding unsafe changes.

7.7. Chapter Summary

We talked a lot in earlier chapters about process management, memory management, kernel vs. user threads, private vs. shared address spaces, asynchronous system calls, asynchronous page faults, scheduling issues, signal handling, etc.; this chapter is where we tied a lot of those ideas together. The guiding principles have been avoidance of serial bottlenecks reducing kernel/user interaction, and placing as much control as possible in the hands of the user, rather than the kernel.

In a major thrust against kernel dominance, we presented low-overhead, system call-free, mechanisms for temporary non-preemption and signal masking. We augmented this with a general method for constructing hybrid synchronization algorithms (part busy-waiting, part context-switching) and a new form of kernel-provided semaphore (ksems), designed specifically to support a variety of user-mode context-switching synchronization methods, without system calls in the common case.

Finally, we outlined a spectrum of solutions for user-mode management of threads and address spaces. Not surprisingly, there are many tradeoffs to be made between desired semantics and cost (both run-time overhead and implementation complexity).

Chapter 8: Experience with Symunix-1

Chapters 3 through 7 described the Symunix kernel in some detail; primarily the newer design which has come to be called Symunix-2. Where Symunix-1 and -2 differ, the emphasis has been on the newer version, but in the most significant cases we have tried to present the evolutionary path taken from one to the other. Whereas the earlier chapters were concerned with qualitative description and justification, this chapter focuses on quantitative results.

To provide an understanding of these results, we begin in section 8.1 by briefly describing the hardware system under study: the Ultra-2 prototype. After describing our instrumentation methods in section 8.2, we present the results of several performance measurements in sections 8.3−8.5, and conclude with a summary in section 8.6.

8.1. Ultra-2

The first generation of NYU Ultracomputer prototype hardware, later dubbed Ultra-1, was built around two Motorola 68000 evaluation boards, each with a 68000 microprocessor, some RAM, ROM, serial and parallel ports, and a small wire wrap expansion area. The expansion area was utilized to build an interface to external hardware containing a Motorola 68451 MMU and a custom-designed "Ultra Bus" for connecting the two processors to the memory and I/O system (developed earlier as part of the PUMA project; the PUMA itself was not used for the Ultracomputer). Our master/slave UNIX system, mentioned in section 1.2 on page 3, was developed for this platform, and became operational in 1982. This was also the platform to run the very first version of Symunix-1, in 1984.

Ultra-1 was intended as a quick stepping stone to a more ambitious machine, which came to be known as Ultra-2. Ultra-2 includes 8 processors with new, but largely compatible, custom board designs. Each processor contains the pin-compatible 68010 instead of the 68000, circuitry to initiate Fetch&Add operations, "assembly/disassembly" hardware to provide atomic 32 bit memory access despite the 68000's 16 bit interface, 32 kilobytes of RAM memory, and a 32 kilobyte 2-way, set-associative, write-through cache with a 32 bit line size and virtual address tags. Fetch&Add operations are performed atomically, at the memory module. Certain upgrades were performed over time: the memory was replaced with a compatible board design independent of the PUMA, floating point hardware was provided in the form of an add-on board for each processor, and the PDP-11 acting as I/O controller was replaced with an Intel 386sx-based personal computer, dubbed the IOP (I/O Processor). Figure 16, on the next page, gives a block diagram of Ultra-2 at the height of its use. A total of five Ultra-2 systems were constructed.

The Motorola 68451 MMU perfectly matched our software plans, since it supports power-of-2 sized and aligned segments, just right for our buddy system memory allocation

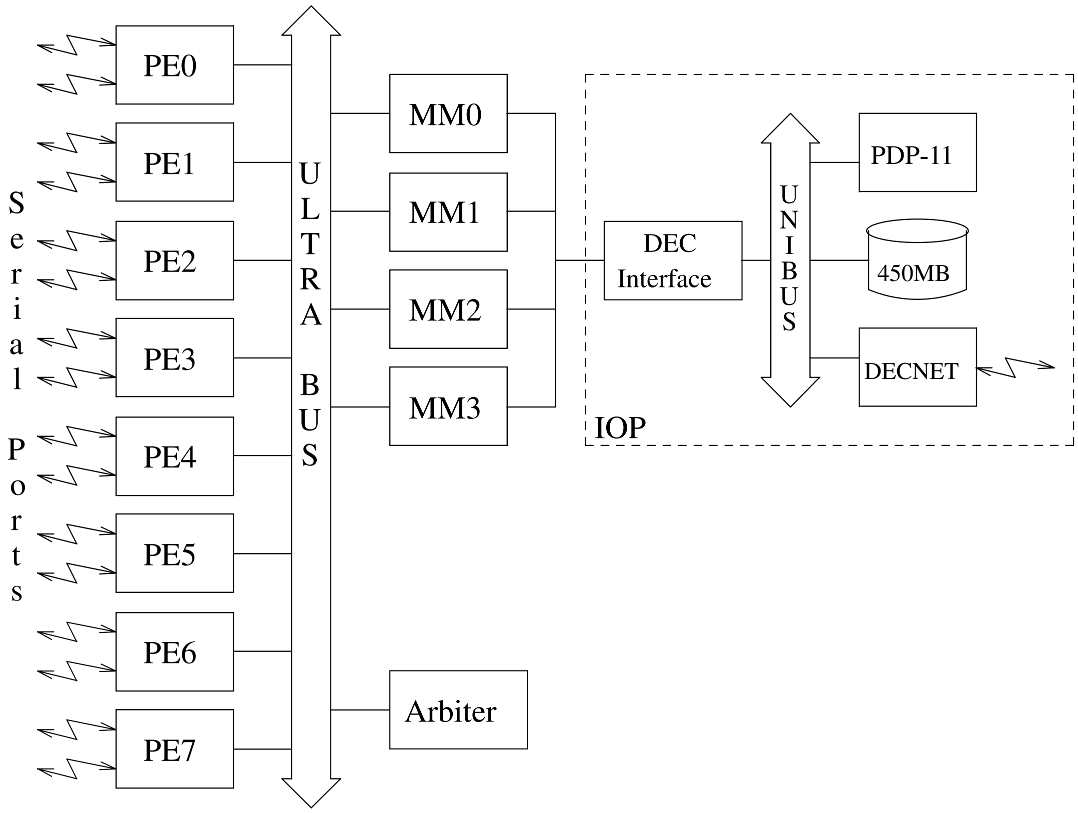


Figure 16: Ultra-2 Hardware Block Diagram.

Each MM contains 4MB of memory. (This Figure is derived from Bianchini [28]).

algorithm (§5.3.4/p244). Cacheability and access to processor-private memory or devices is controlled for each memory access by three bits from the MMU.[175] One specifies "on board" or "off board" access, the second cacheability for data references, and the third cacheability for instruction fetches. The use of two bits for cacheability was a concession to Symunix-1, since it uses the "text" segment to hold writable shared variables as well (as discussed in section 1.2 on page 3).

Basic Performance of Ultra-2

In order to understand some of the timing data presented later in this chapter, we must know something about the basic performance of the hardware. Ultra-2 runs at 10MHz. The microprocessor uses at least four clock cycles (400ns) to access memory; this is achieved in

[175] We simply used the three high-order physical address bits, passing more address bits "around" the MMU on the low end, and raising the minimum segment size from 256 bytes to 2 kilobytes.

Ultra-2 for cache read hits, with access times for writes and read misses being longer but variable due to bus contention.

The dominant factor limiting performance is bus and memory contention. As a worst-case example, Figure 17, below, shows the maximum aggregate memory copy performance with varying numbers of processors.

Figure 18, on the next page, shows the maximum performance for disk reads of various sizes, made through the UNIX "raw I/O" interface.

The Dhrystone 2.1 benchmark results are given in Figure 19, on page 331, which shows the average performance of multiple processes running the same benchmark.

It is important to remember that Ultra-2 was not built with performance as a primary goal. The number of processors is modest, the Ultra Bus is not nearly so optimized as commercial buses (e.g., Sequent [19]), we did not consider even simple "snoopy" cache coherence schemes, and the clock speed (10MHz), although not bad at the time, quickly lost ground to commercial machines. What Ultra-2 did well was provide a base for highly parallel software development, testing, and evaluation.

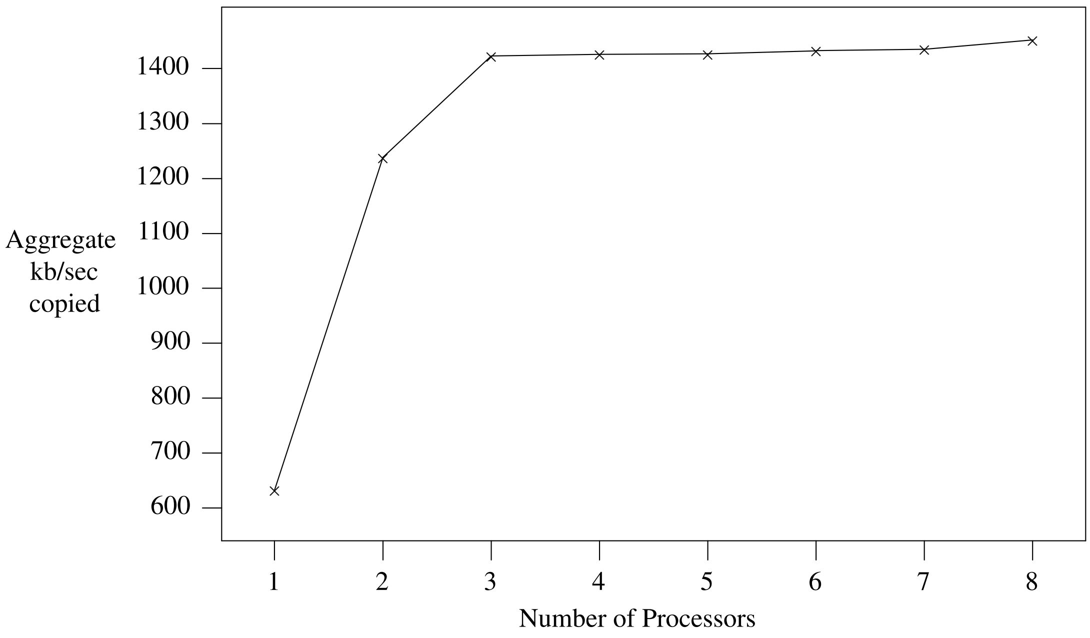


Figure 17: Ultra-2 Aggregate Memory Copy Performance.

Two processors can get just about all the memory bandwidth they ever want, but three cannot. The best copy rate as seen by a single processor can vary from about 182 to 632 kilobytes/second, depending on what other processors are doing. I/O transfers through the PDP-11 or IOP have an additional affect.

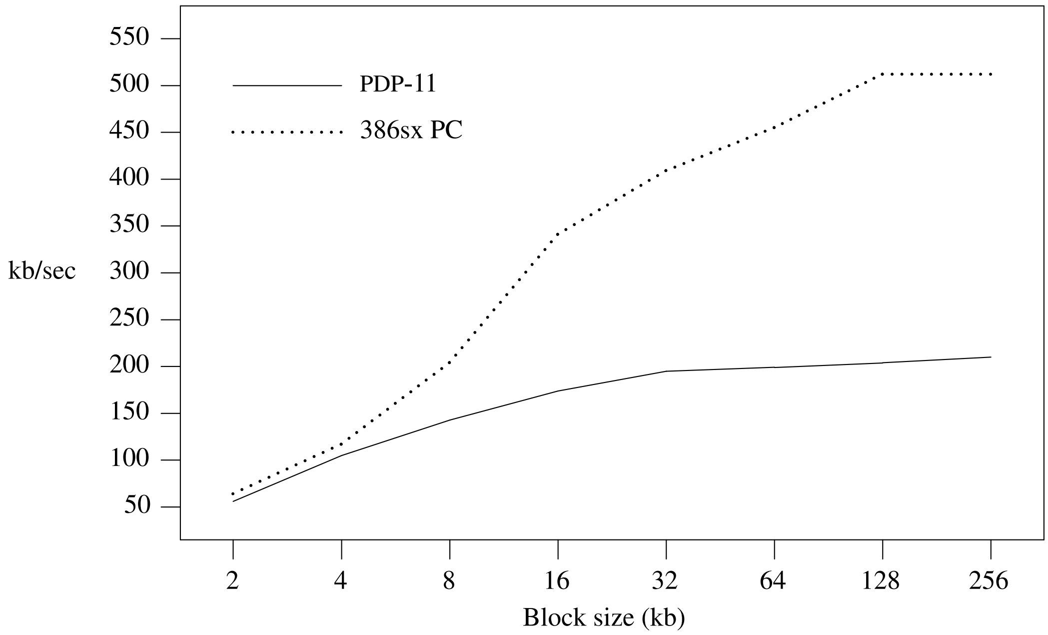


Figure 18: Ultra-2 Disk Read Transfer Rate.

Random reads of the “raw” disk for various block sizes. Performance is dependent on hardware configuration details and IOP software; data is presented from the two configurations actually used. Normal file I/O in Symunix-1 is performed exclusively on 2 kilobyte blocks; larger blocks are used only for swapping and raw I/O.

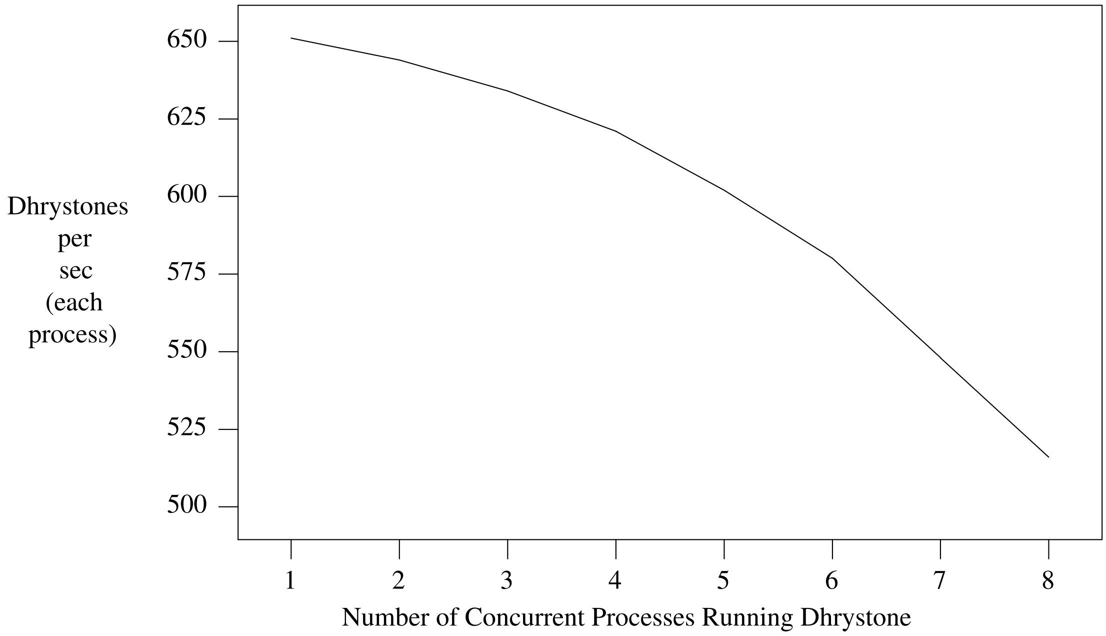


Figure 19: Ultra-2 Dhrystone 2.1 Performance.

8.2. Instrumentation

The kernel has been augmented with several forms of instrumentation that aid performance analysis and debugging. Most of them are only conditionally compiled, but have almost never been omitted because of the research value of the data collected.

Informational System Call

Some useful information is available to user programs via a system call. It returns a structure containing static information about the system (kernel name and version, etc.), the number of processors, time of last boot, "load average" data, and the number of process swaps to secondary storage since boot time. (Recall that Symunix-1, like 7th Edition UNIX from which it is descended, is a swapping, not a paging system.) The swapping data has been important in ensuring the validity of conclusions drawn from performance measurements on Ultra-2. By checking the number of swaps before and after an experiment, it is easy to verify that there were none during the experiment. Recall that Ultra-2 has fairly poor I/O performance (§8.1/p330), and because of the way buddy system memory allocation works, it is often difficult to predict whether or not a given parallel application will require swapping (§5.3.4/p247).

System Call Counts

The number of times each system call is executed is accumulated (by Fetch&Increment) in the appropriate element of an array. The array may be extracted from the kernel by reading `/dev/kmem`.

Significant Events

The number of times each of several interesting events happened within the kernel is recorded by Fetch&Increment on an element of a counter array. The increment is performed by a macro:

```
#define didhappen(thing) vfai(&happenings[thing])
```

Each event (`thing`) is given a symbol beginning with `DID`; these symbols are defined in a single include file to reference distinct indices of the `happenings` array. It is a simple matter to examine the array by reading `/dev/kmem`. There are many such events defined in the code; such as performing a queue deletion on a non-empty queue, encountering a "hole" in a queue (§3.7.1/p79), or swapping for any of several possible reasons.

Histograms

Both types of histograms described in section 3.3 are used in the kernel. They are examined by reading `/dev/kmem`. The exponential histograms are used to measure waiting time for various locks and other events. Every source of busy-waiting or blocking is measured in this way, generally with one histogram for each similar kind of use, including:

- Queue and pool sublist locks (busy-waiting).
- Other linked list locks (busy-waiting).
- Per-process readers/writers lock (busy-waiting). Readers' and writers' waiting times are accumulated together.
- Disk buffer readers/writers lock (context-switching). The readers' and writers' wait times are separated in this case.
- TTY input and output waiting times (context-switching). Waiting occurs when writing if the output buffer fills, and when reading if the input buffer is empty.

The context-switching times are measured in clock ticks, which occur every 1/16 second, and the busy-waiting times are measured in iterations of the busy-waiting loop. Experimentally determined data showing the correlation between waiting iterations and real time is presented in Figure 20, on the next page.

Ordinary histograms are used for other things, including:

- Distribution of clock interrupt work per processor.
- Distribution of disk request queue lengths.
- Distribution of TTY character buffer lengths.
- Distribution of load averages.
- Distribution of scheduling priorities.
- Swap time for each memory block size. This is actually two histograms, one for the total time for each size, and one for the number of swaps of each size.

8.3. Scalability

An important question is, "at what size machine do Fetch&Add-based algorithms such as readers/writers locks and highly-parallel queues begin to pay off?" Several experiments were performed to investigate this. The method used is to replace certain highly parallel structures with less parallel ones, and look at the change in performance. Consider three versions of the operating system kernel:

Normal Symunix-1, as already described.

Reduced A version with reduced support for parallelism. All reader/writer and reader/reader locks are systematically replaced by simple binary locks (busy-

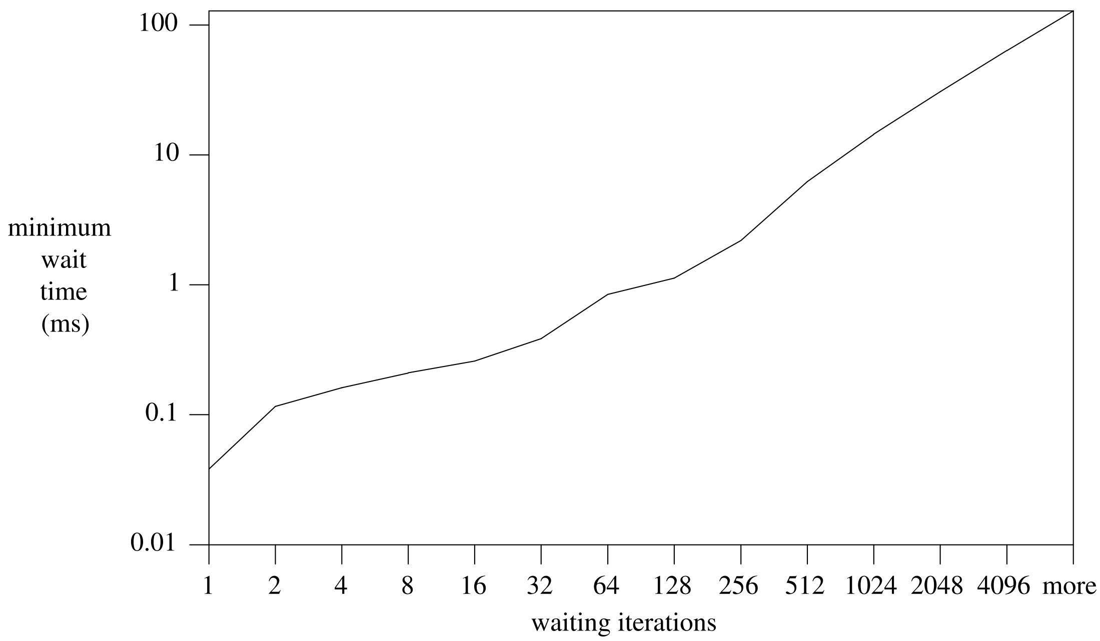


Figure 20: Minimum Waiting Times for Histogram Values.

The minimum real waiting time implied by each possible histogram value is plotted. Memory contention and interrupt processing can add further delays.

waiting or context-switching, as appropriate), and all queues and pools are replaced by simple linked lists protected by busy-waiting locks.

Uniprocessor A version for a uniprocessor, with all the substitutions of the reduced version, but with busy-waiting locks eliminated completely. It runs only on a single processor of the Ultra-2 prototype, with the other 7 processors effectively halted.

The substitutions to produce the reduced and uniprocessor versions are done with a fairly small number of `#ifdefs` and `typedefs`, and no other tailoring. The simplicity of the substitutions supports the notion that the lock and list abstractions we use are of benefit even to systems running on uniprocessors or small multiprocessors, but we cannot quantify the impact of fine-grained locking and other techniques that are not modified by the substitutions.

8.3.1. The `fork` and `spawn` System Calls

In the normal version of the kernel, the `spawn` system call is supported in the ready list by including a multiqueue for each priority level along with an ordinary queue (of "singleton" items). A spawning process inserts itself, with appropriate multiplicity, onto the ready list. In contrast, `fork` is implemented in the traditional UNIX way: a parent performs all initialization before placing its child on the ready list. Because `spawn` does not follow the "parent does all" approach of `fork`, it incurs no serial work proportional to the multiplicity, thus avoiding a potential bottleneck.

When a processor looks on the ready list for a process to run, it tries all the priority levels from highest to lowest, looking for a queue or multiqueue from which a deletion can be made. When a process is found on a multiqueue, a proc structure and u-block are allocated, and minimal initialization is performed. After the minimal initialization, the child gains control and completes the initialization. The parent is able to run only after the last child has completed all initialization, and is ready to run in user-mode.

The reduced and uniprocessor versions of the kernel don't have multiqueues in the ready list, so the "parent does all" approach is used for `spawn` as well as `fork`. The parent must perform all initialization and do an ordinary queue insertion for each child. (There are other alternatives to this simple approach. As we will see on page 338, the reduced kernel could profit handsomely by performing the u-block copies in parallel.)

A program was written to measure the performance of `fork` and `spawn`, and run on Ultra-2 with the normal, reduced, and uniprocessor kernels. The program creates some number of *worker* processes, that each go into a loop creating *drones* and `wait`ing or `mwait`ing for them.[176] The drones do nothing but exit. Several parameters may be varied:

- How many workers are created.
- Whether `fork` or `spawn` should be used.
- The multiplicity to be used when workers `spawn` drones.
- The total number of drones to be created. This must be a multiple of the multiplicity.

The primary result of running this program is the elapsed real time, user time, and system time; the graphs report throughput in real time. In addition, we gather quite a lot of other data about each run, such as the distribution of work among workers (how many children did each create), used to verify that we experience the concurrency we expect, and most of the kernel instrumentation described in section 8.2. The experiments were run on a machine with no other users or daemons running. Each result plotted is the mean of several (at least 6) runs (standard deviations were too small to plot).

Single `spawn` Performance

Figures 21, 22, and 23 show the rate at which a single process can create new processes with `spawn` and `mwait`, for various multiplicities. Each new process gets a copy of the 16 kilobyte data segment, 2 kilobyte user stack segment, and 4 kilobyte u-block. These copies are all done in parallel for the normal kernel, but the reduced kernel copies the u-block serially. The uniprocessor kernel is entirely serial, since it runs on only one processor.

Figure 21, on the next page, shows the original results obtained by running this test. The erratic results were investigated, with the initial conclusion that the program was suffering from significant swapping. For example, when the normal kernel data for Figure 21 was collected, 7% of new processes with multiplicity = 8 were unable to allocate memory, so their images were created on the disk and swapped in later. Intuitively, there should be no swapping, because the machine had plenty (16 megabytes) of memory. The swapping was

---

[176] `mwait` waits for all spawned children in a way that doesn't serialize in the absence of signals or errors (§4.2/p149).

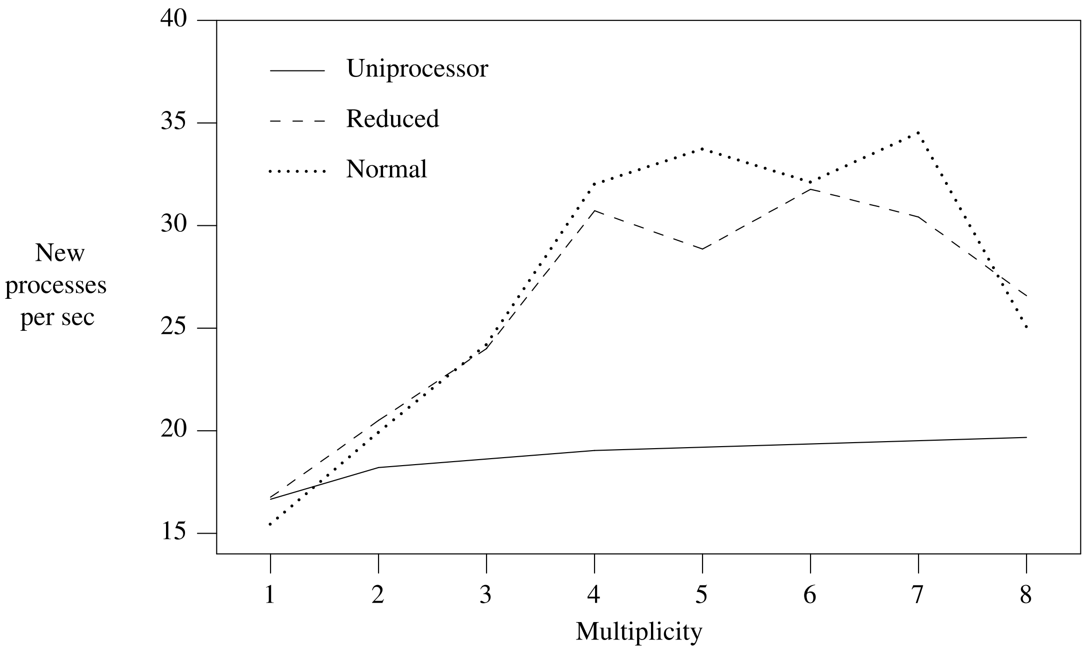


Figure 21: Throughput of `spawn/mwait` (Original Kernels).

caused by a race condition in the buddy memory allocation algorithm in the kernel: when a request for a block of a certain size couldn't be immediately satisfied, a larger block would be split and the unneeded half made available. When multiple processors tried to allocate memory at about the same time, some of them were likely to fail to get any block at all.

The buddy allocation race was corrected by adopting the algorithm described in section 5.3.4 on page 247, and the data presented in Figure 22, on the next page, was collected. Clearly there was another force at work, because the timings were still erratic. Furthermore, they were very sensitive: small changes in the test program that should have been insignificant sometimes caused big changes in the performance numbers. This time the scheduler was to blame: when a processor had nothing to do, it would go into an idle loop where it would do nothing until the next interrupt arrived. After handling the interrupt, the processor would try to get work from the ready list, and continue idling if none was found. The Ultra-2 real time clock broadcasts an interrupt to all processors at a 16 Hz rate. Because of this idle strategy, all but one spawned child will generally wait for the next clock interrupt before starting. Since each `spawn` follows an `mwait`, the program would become synchronized with this 16 Hz clock and exhibit continuously unpredictable performance.

This idle problem was corrected by making an idle processor check the ready list more frequently. The frequency of checking was made long enough not to cause significant memory contention, but very short compared to the time to initialize a spawned child process. The implementation was simply a tight loop counting down from 256 to 0, executing out of cache and making no data references. The results of this change are shown in Figure 23, on page 337, where the improvement is dramatic.

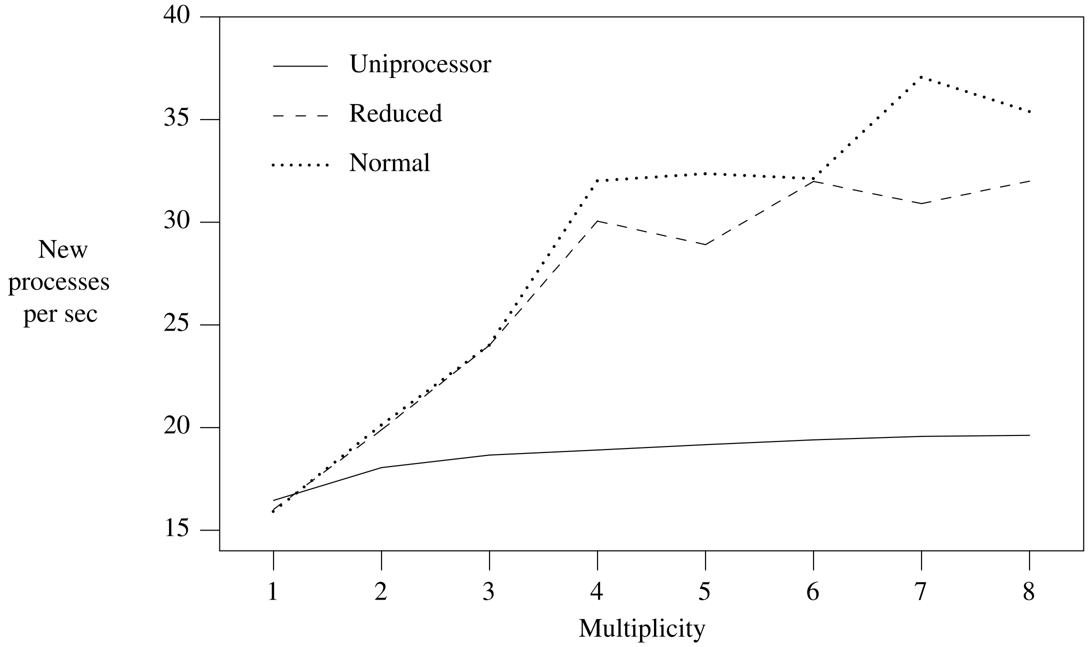


Figure 22: Throughput of `spawn/mwait` (Improved Buddy Alg.).

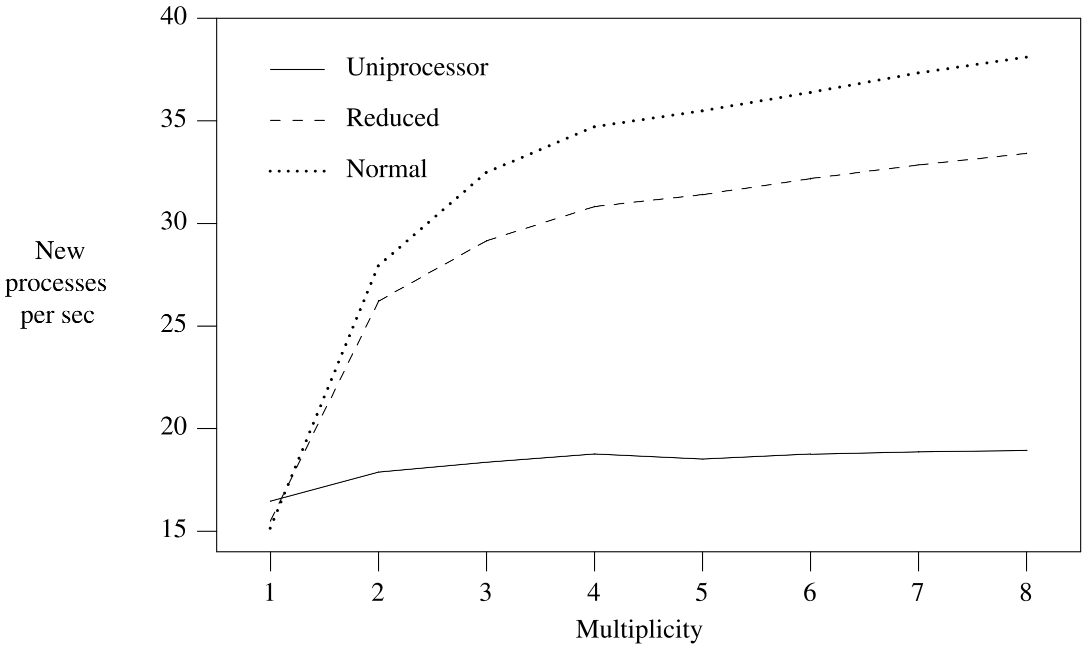


Figure 23: Throughput of `spawn/mwait` (Improved Buddy and Idle).

Despite the improvements shown in the progression from Figure 21 to 23, the performance is still limited by memory copying. From Figure 17 we can see that copying 22 kilobytes may take from 35 to 121 milliseconds on Ultra-2, depending on bus and memory contention. This generally accounts for about half of the time indicated by Figure 23. The average performance gain for the normal and reduced kernels in Figure 23 is about a factor of 2.3, almost identical to the gain shown in Figure 17. In fact, the difference in performance between the normal and reduced kernels in Figure 23 is almost entirely due to the latter's serial copying of the u-block. It is possible to estimate the throughput of `spawn` with a hypothetical memory copying cost of 0 by using the data of Figures 17 and 23; the result is shown in Figure 24, on the next page.

This similarity of performance between normal and reduced kernels makes it harder to evaluate the effect of the bottleneck-free synchronization and list algorithms. Do the better values for the normal kernel with multiplicity ≥ 4 indicate a scalability trend, or are they too small to be significant? We can gain some additional insight by looking at synchronization waiting times within the kernel. There are several busy-waiting locks that appear to behave differently under the normal and reduced kernels:

Callouts.

A *callout* in the unix kernel is a structure describing some (small amount of) work to be done at a specific time in the future (Bach [15]). On each clock interrupt, a list of pending callouts is checked, and the work is done for those whose time has arrived. Symunix-1 has a central pool of available callouts, and a separate list of pending callouts for each processor (protected by a lock so that any processor can schedule

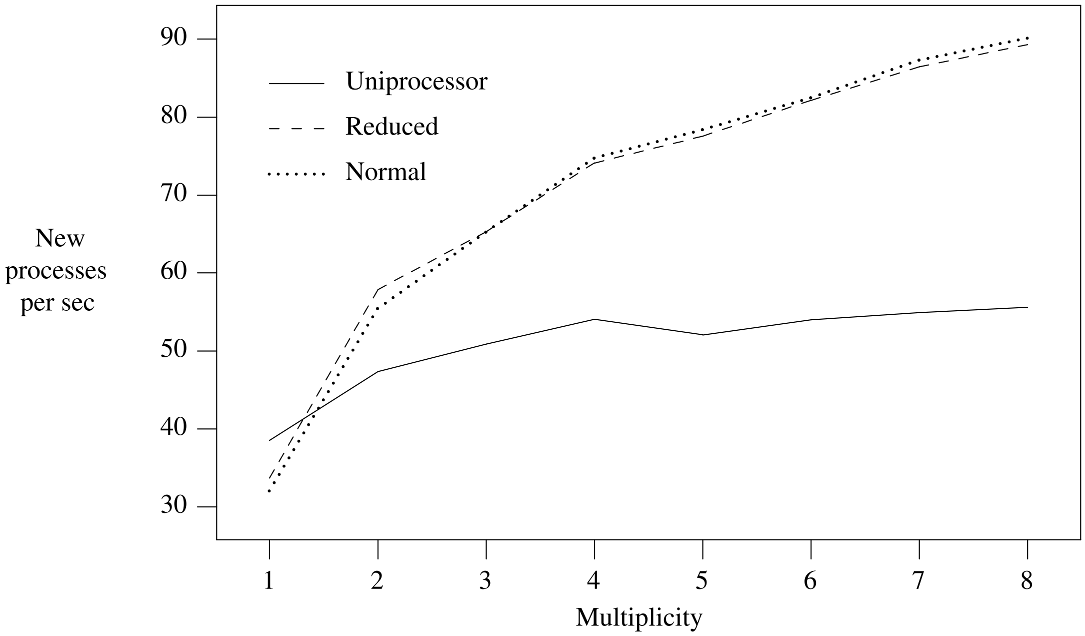


Figure 24: Throughput of `spawn/mwait` (Estimated, no Memory Copy Cost).

Copy overhead is estimated by assuming parallel copies are perfectly synchronized and using the data of Figures 17 and 23.

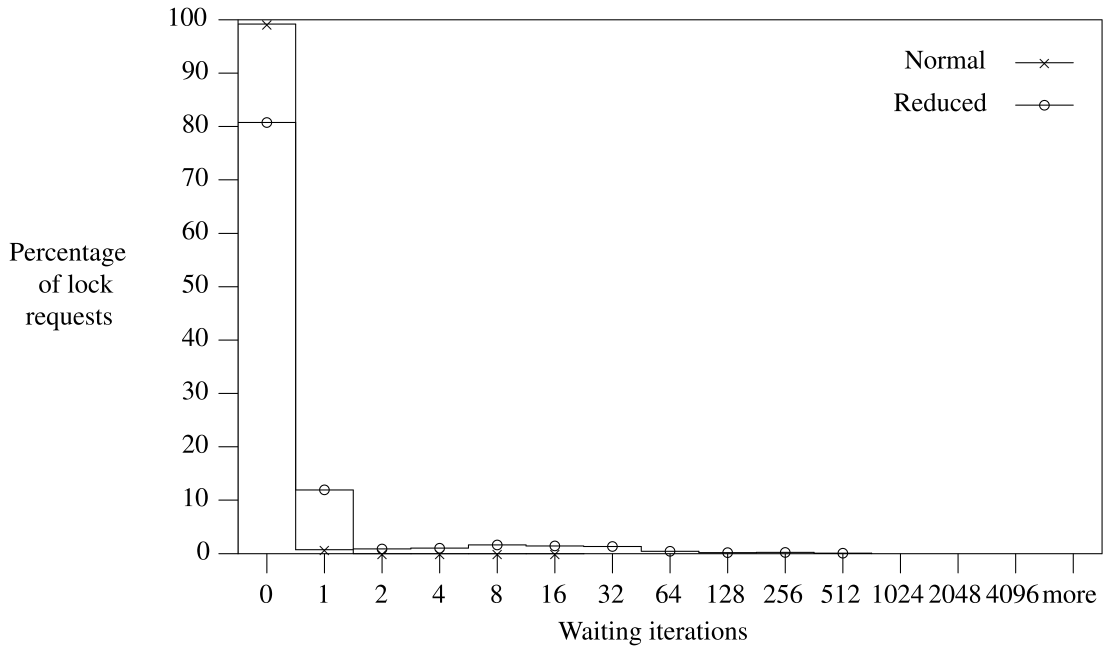


Figure 25: Available Callout List Lock Waiting Times.

This data is from the same experiments that produced the multiplicity 8 values in Figure 23. The number of lock requests per second is the same for both kernels. Waiting times corresponding to iterations are shown in Figure 20. The single available callout list is implemented as a highly-parallel pool (using 8 locks) in the normal kernel, and as a simple linked list (using 1 lock) in the reduced kernel. In both cases, there were about 30 list operations per second, evenly divided among the processors.

work for itself or any other).[177] During the experiments that produced Figure 23, the only significant callout activity was some periodic device support executed by each processor about once each second. Instrumentation results show that there is essentially never any waiting for the locks protecting the per-processor callout lists, but there is some contention for the pool of available callouts. Figure 25, above, shows the histogram of waiting times for the lock(s) protecting that pool in the normal and reduced kernels.

Process Structures.

Each proc structure has a busy-waiting readers/writers lock used to control state changes and coordinate the parent/child relationship during process termination. The lock is always obtained for exclusive access, except for terminating children, which obtain a shared lock on their parent. Instrumentation shows very little

[177] Note the per-processor pending callout list cannot be any of the types described in section 3.7.1 because it must be sorted (as described by Bach [15]).

contention for this lock (for the multiplicity = 8 experiment shown in Figure 23, waiting was required for less than 2% of normal kernel requests and 3% of reduced kernel requests).

Like the callout structures, unused proc structures are kept in a pool. Very little contention is experienced for this pool (for the multiplicity=8 experiment shown in Figure 23, waiting was never required for this pool in the normal kernel, and only needed for .2% of the pool operations in the reduced kernel).

Ready list.

Processes that are waiting to run are on placed on the *ready list*, which actually consists of many lists, one for each priority. The normal kernel uses highly-parallel list algorithms, and also has additional lists supporting multiplicity for the sake of `spawn`, but the reduced kernel uses simple linked lists protected with locks. The highly-parallel list algorithms are based on algorithms of Wilson [205], and are built of several linked lists, each with its own lock. Figure 26, below, shows that the more

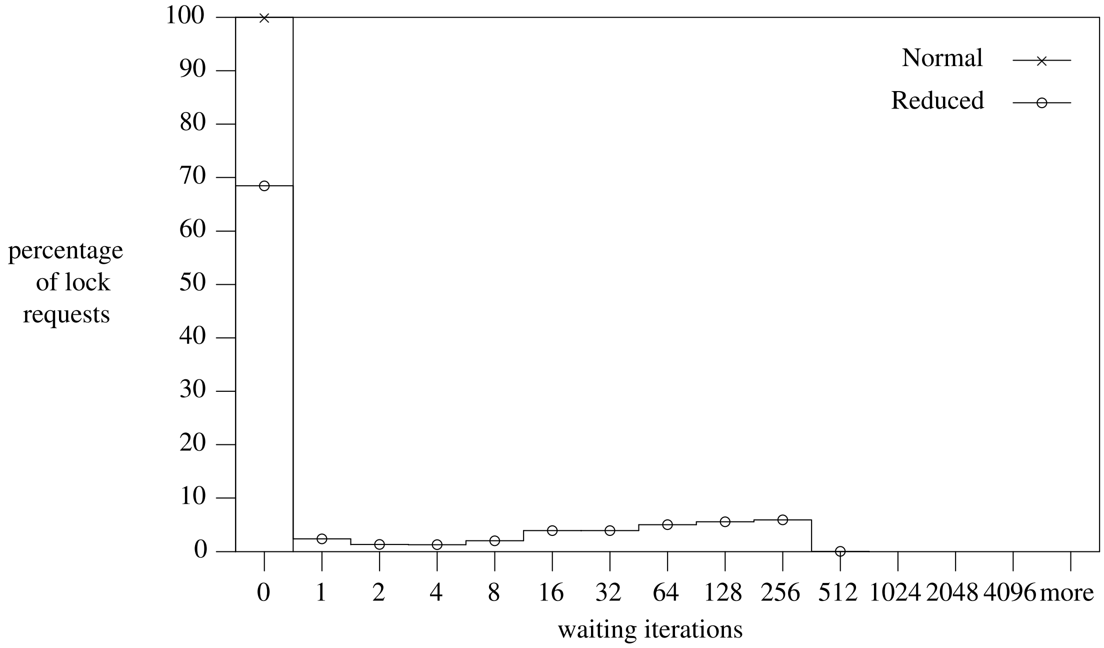


Figure 26: Ready List Lock Waiting Times.

This data is from the same experiments that produced the multiplicity 8 values in Figure 23. The normal kernel has a both a singleton and multi-queues (to support `spawn`). The reduced kernel makes more lock requests than the normal kernel, because one multi-insert is replaced with eight singleton inserts, and because the multi-queue algorithm allows deletion of eight items from one multi-item with only two lock acquisitions, instead of eight.

parallel algorithms seem to have paid off in this case, as they never suffer from any lock contention at all, while more than 30% of the simpler algorithm's lock requests suffer some delay.

Buddy memory allocator.

We saw in Figures 21 and 22 that the buddy memory allocation algorithm has a big impact on performance. The improved buddy memory allocation algorithm, which is described in section 5.3.4 on page 247, has five kinds of locks or lists that employ busy-waiting. Using the terminology of pages 246 and 247, these are:

- `bfree(`j`)`. This is the list of available blocks of size 2^j. Instrumentation shows that neither kernel seems to wait significantly on the locks protecting these lists.
- `bfrr(`j`)`. This is a readers/writers lock that prevents simultaneous freeing of even and odd buddies of size 2^j. Figure 27, below, shows the reduced kernel waits for 17% of these lock requests compared to the normal kernel's 11%.
- `barr(`j`)`. This is a readers/writers lock that prevents simultaneous starting and finishing of merged allocation requests of size 2^j. Figure 28, on the next page, shows the reduced kernel waits for 19% of these lock requests compared to the normal kernel's 5%.

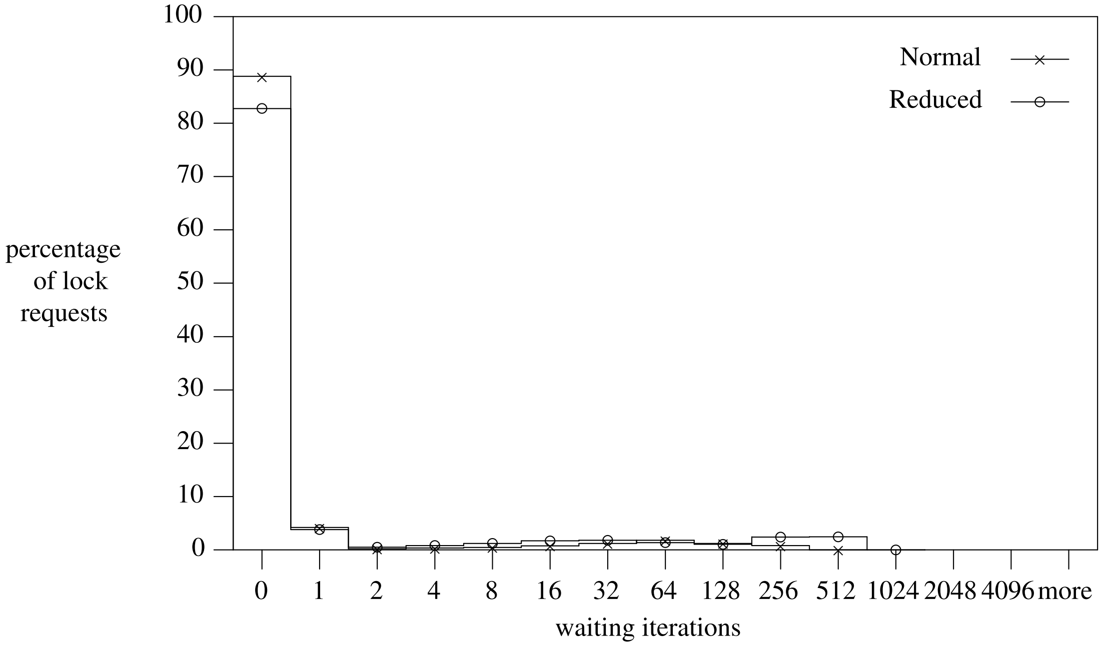


Figure 27: Buddy Free Readers/Readers Lock Waiting Times.

This data is from the same experiments that produced the multiplicity 8 values in Figure 23. These locks prevent simultaneous freeing of even and odd buddies.

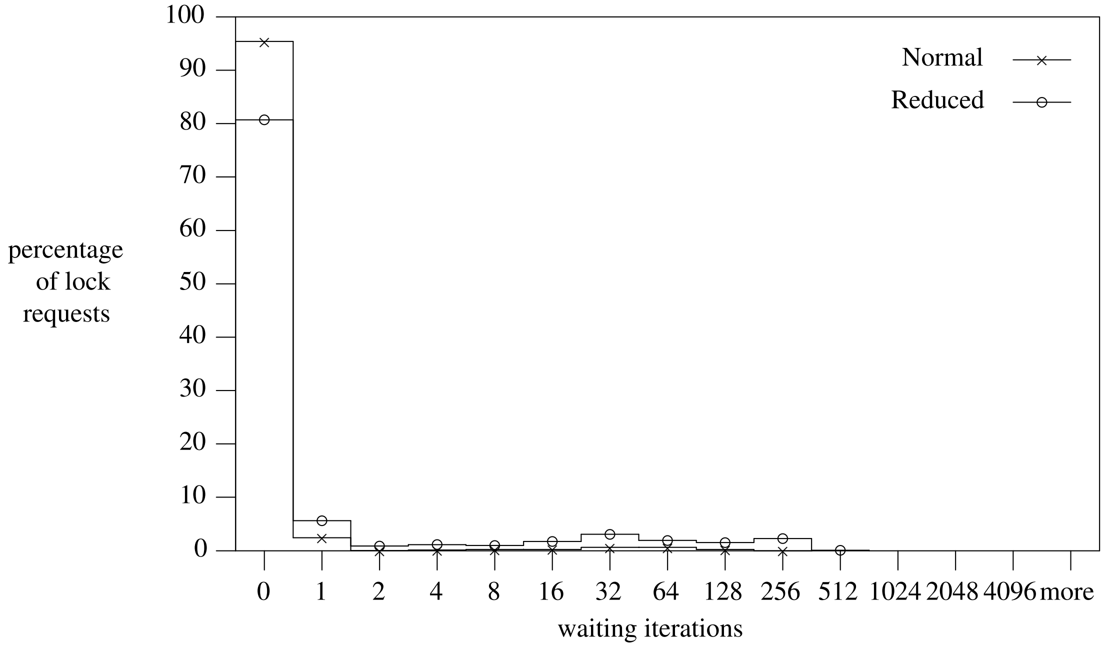


Figure 28: Buddy Allocation Readers/Readers Lock Waiting Times.

This data is from the same experiments that produced the multiplicity 8 values in Figure 23. These locks prevent requests from merging together into larger requests at the same time as previously merged requests are splitting their results.

- `bspare(`*j*`)`. This is a pool to hold blocks allocated by even-numbered requests on behalf of odd-numbered requests of size 2^j. The normal kernel never seems to require waiting on `bspare(`*j*`)`, but the reduced kernel waits for 4% of the operations on it.
- `bmerge(`*j*`)`. This is a binary semaphore used by odd-numbered requests to await availability of a block being split from a larger block by an even-numbered request. As such, it measures the effect of all the other synchronization required to allocate and split a larger block. Figure 29, on the next page, shows 21% more waiting in the reduced kernel than the normal one, on the average.

Clearly, Figure 24 shows that the extra parallelism provided by the normal kernel's bottleneck-free synchronization and list algorithms isn't enough to provide a big performance boost on the Ultra-2 prototype. However, it is important to note that the performance difference is not large, and the bottleneck-free kernel never incurs a significant penalty, even for a multiplicity of one, compared with the kernel using more conventional locking algorithms. With 8 processors, we can detect the effects of lock contention, and the benefits of bottleneck-free algorithms, although they are small. This suggests that a machine with enough concurrency to benefit from these algorithms won't be impractically large.

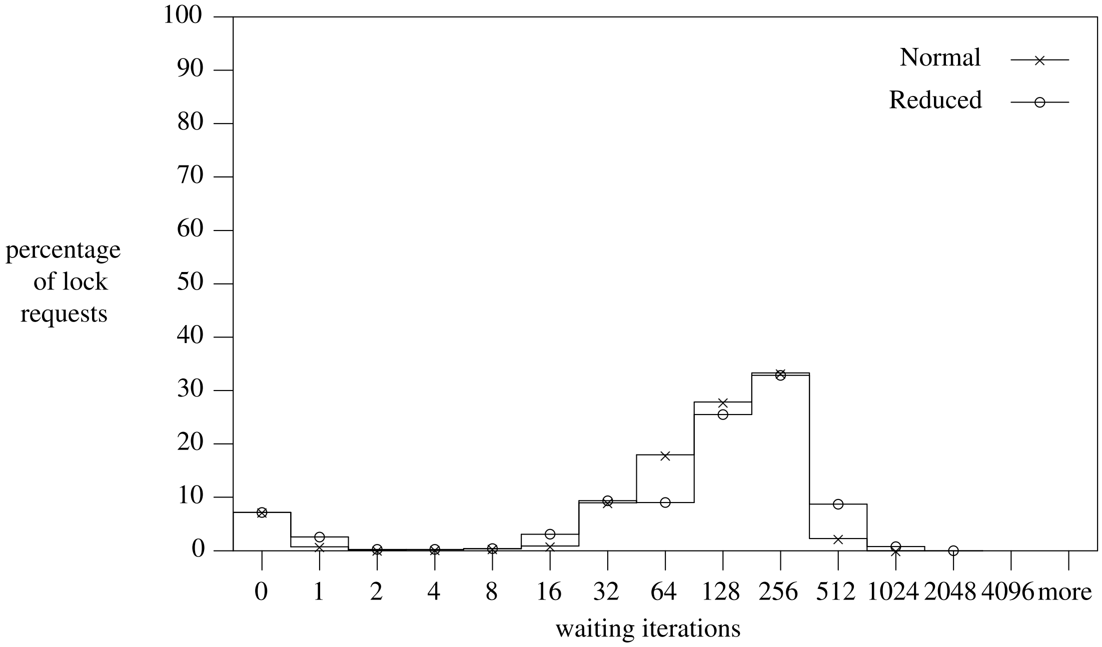


Figure 29: Buddy Allocation Merge Lock Waiting Times.

This data is from the same experiments that produced the multiplicity 8 values in Figure 23. This semaphore measures the effect of all the other synchronization required to allocate and split a larger block to satisfy two requests. According to the data shown in Figure 20, the mean waiting time is 1.4ms for the normal kernel and 1.8ms for the reduced kernel.

Multiple `spawn` *Performance*

We see from Figure 23 that the highly parallel kernel can `spawn` new processes 14% faster than the reduced kernel and 95% faster than the uniprocessor kernel. But how do multiple concurrent `spawns` interact with one another? Figure 30, on the next page, shows similar measurements when more than one `spawn` is executed concurrently. This graph gives another idea of the cost of highly parallel synchronization, as the reduced kernel's result is better than the normal kernel's by 4% for multiplicity = 1 but worse by 12% for multiplicity = 8. The multiplicity = 1 difference reflects two important factors:

(1) The reduced kernel suffers no disadvantage of serial u-block copying, as it does with higher multiplicities.

(2) The 8 workers are operating independently, so there is naturally less lock contention and therefore less advantage for the highly parallel algorithms employed in the normal kernel.

Most instrumented values behave similarly for the experiments of Figures 23 and 30, although some exhibit reduced contention in Figure 30, especially for the reduced kernel at low multiplicities. The clearest example is the ready list, for which the normal kernel uses

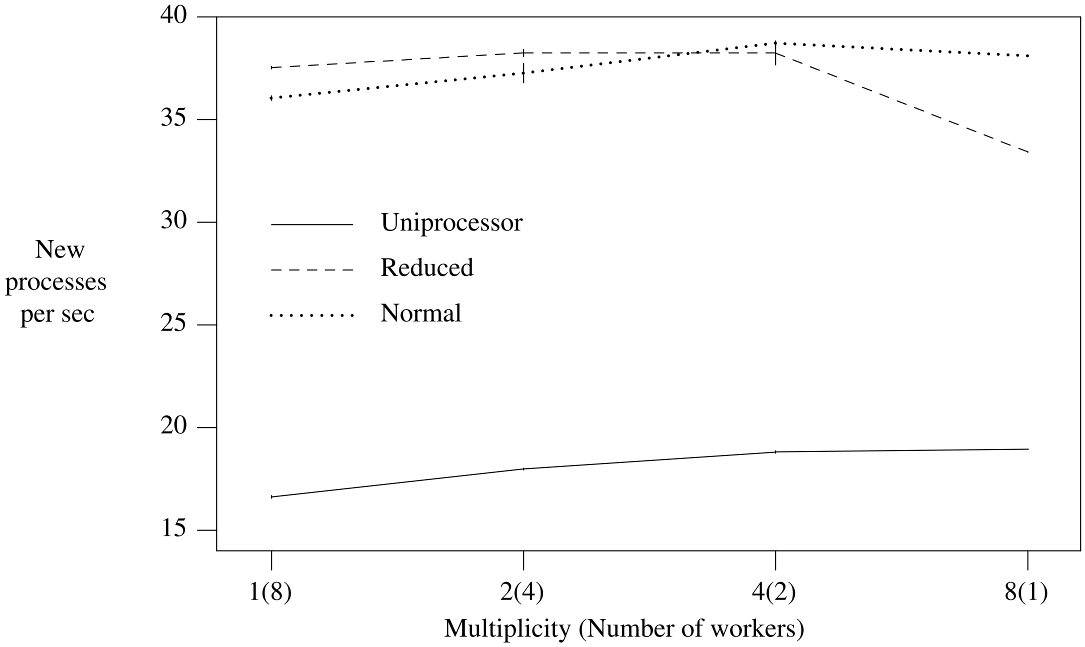


Figure 30: Cost of Concurrent `spawn` on Ultra-2.

Performance is dominated by copying the 16 kilobyte data segment, 2 kilobyte user stack segment, and 4 kilobyte u-block. Vertical lines show a small but noticeable standard deviation in some cases.

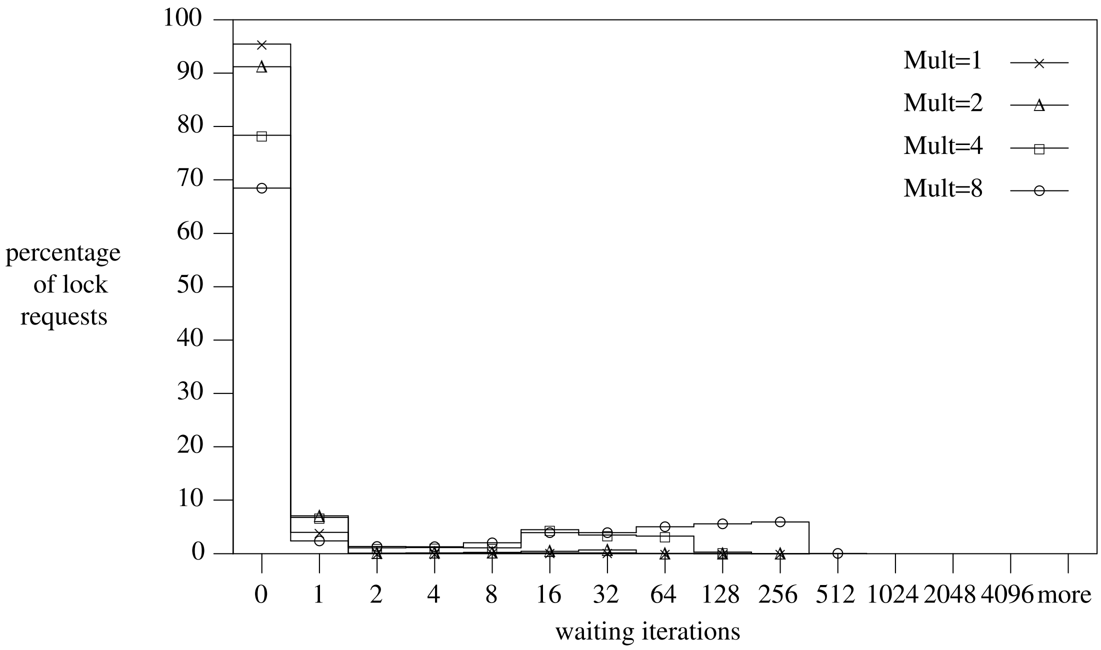


Figure 31: Ready List Lock Waiting Times, Reduced Kernel.

both singleton queues and multi-queues, while the reduced kernel uses only singleton queues. On page 340 we showed how the multi-queues were good for performance, especially for multiplicity = 8, because they reduce the number of queue operations and also the amount of locking internal to the queue. But when the `spawn` multiplicity drops, so do the advantages. Because independent `spawns` aren't executed in close synchronization, they are less likely to access the ready list at the same time. The singleton queues used in the reduced kernel are almost "perfect" for many concurrent multiplicity = 1 `spawns`: they impose low overhead and exhibit low contention. The multi-queue algorithm used in Symunix-1 requires a constant amount of locking with a break-even point at multiplicity = 2, and allows parallelism on deletions only to the extent of the head multi-item's multiplicity. (Improved multi-queue algorithms should be helpful in this respect.)

Figure 31, above, shows reduced kernel lock contention for the ready list with different multiplicities. The normal kernel never waits for ready list accesses when multiplicity = 4 or 8, but waits for almost .5% when multiplicity = 2, and about 1.25% for multiplicity = 1.

8.3.2. The `open` and `close` System Calls

In any UNIX system, the basic job to be done when opening a file is to obtain an in-memory copy of the file's key information, called the *i-node*. Naturally, there are several steps to this:

- The i-node of each directory in the path must be found (i.e., this is a recursive description). The starting point is the root or current working directory; the others are found by searching in sequence. The end result of this procedure is a pair of numbers, the device number and the *i-number*, which uniquely identify the final file and the location of its i-node on the disk.

- If a needed i-node is already in memory, no I/O is necessary, but this cannot be determined without a search.
- If the search for the i-node in memory fails, an i-node structure must be allocated and initialized (by finding the on-disk i-node structure in the buffer cache, or by reading the disk).
- Directory searches are like reading the contents of an ordinary file, and must be protected against concurrent updates (e.g., creation of a new file in the directory).
- A *file* structure must be allocated and initialized. The file structure is an object whose primary responsibility is to maintain the seek pointer generated by open. It is required by traditional UNIX semantics.

There are also several steps to reading or writing a file or directory:

- In the case of an ordinary file read or write, a file descriptor is first translated into a file structure pointer. The file structure points to the i-node in memory. (Directory look-ups don't use file descriptors or file structures, they begin with i-node pointers.)
- The i-node contains disk block addresses allowing the file or directory contents to be located.
- File writes and directory updates can cause the disk block addresses in the i-node to change; reads and other updates must be protected from any temporary inconsistencies.
- Before starting I/O on the device, the buffer cache is checked. The cache is fully associative with LRU replacement. On a read hit, data is copied to the user's memory; directory searches operate directly on the kernel's buffer. Writes are not normally written to the device until the buffer is about to be reassigned, but certain file-system-critical data (e.g. i-nodes, directories, etc.) are written through immediately.

Even the seemingly simple operation of close requires several steps:

- The file descriptor is again translated into a file structure.
- The file structure's reference count is decremented; unless it reaches 0, the close operation is finished.
- When the file structure's reference count reaches 0, it must be deallocated, and the i-node's reference count must also be decremented. Unless the i-node's reference count reaches 0, the close operation is finished.
- When the i-node's reference count reaches 0, it must be deallocated in memory. If there are no more links to it in the file system, the file is deallocated.

Symunix-1 adopts the following approach to bottleneck avoidance:

- Each i-node has a context-switching readers/writers lock, to enforce exclusive access only when the file or the i-node itself is updated.
- Searching for an i-node (using the device number and i-number pair) is done by the method described in section 3.7.6. This uses a hash table with busy-waiting readers/writers locks on each bucket.
- Pools are used to keep unused i-node structures and file structures available for allocation.
- Each buffer has a context-switching readers/writers lock to provide mutual exclusion for updates, and a context-switching event to support waiting for I/O completion.

To evaluate the effectiveness of these solutions, a program was written to generate `opens` and `closes`, and run on Ultra-2 with the normal, reduced, and uniprocessor kernels. The program creates some number of worker processes that each make a fixed number of paired `open` and `close` calls. Several parameters can be varied:

- The number of workers.
- The number of times each worker calls open and close.
- Whether all workers open the same file or different files.
- The path name of the directory containing the file(s).
- Whether the file should be read before closing.
- Whether the file should be kept open by the parent, thus ensuring that all closes are as brief as possible (the i-node reference count will never reach 0 during the test).

The primary result of running this program is the elapsed real time, user time, and system time; the graphs report throughput in real time. In addition, we gather most of the kernel instrumentation described in section 8.2.

The experiments were run on a machine with no other users or daemons running. Each experiment was run several (generally 6) times, to produce the mean and standard deviation (shown as a vertical line at each data point in the graphs). Unless otherwise stated, all results in this section were obtained with the same kernel as that used in Figure 23, in

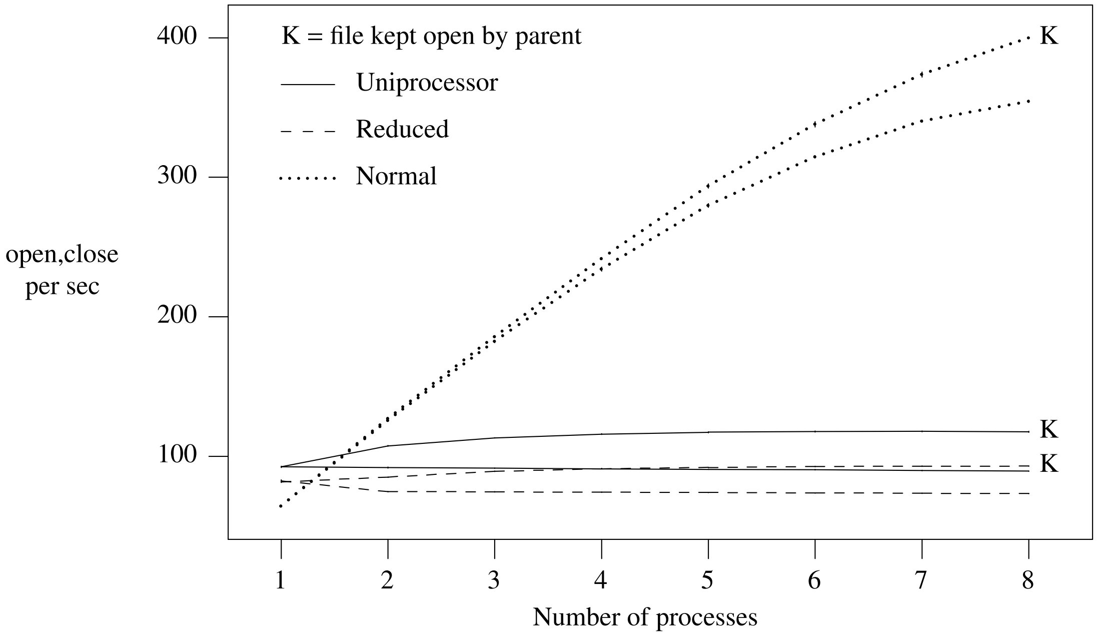


Figure 32: Open/Close Throughput.

Different files, current directory. For the curves marked K, the parent process keeps the files open so the i-nodes never get deallocated from memory. Otherwise, the i-nodes are deallocated on every `close` and reallocated on every `open`. (The current directory i-node is always held open.) No I/O is performed, but the disk buffer cache is searched when allocating i-nodes.

section 8.3.1 on page 337.

Figure 32, on the previous page, shows the case of `open` followed immediately by `close`, as a function of the number of processes doing so. In this case, the processes are all opening different files in the (same) current directory. If the parent process keeps all the files open, the i-node reference counts never reach zero during the test. Otherwise, the data file i-node copies in memory are deallocated on every `close`, so the next `open` has to allocate another one and initialize it by copying from the disk buffer cache. Either way, the normal kernel performs dramatically better than the less parallel kernels. The difference is attributable to the readers/writers locks in the i-node hash table buckets and in the i-nodes themselves.

Figure 33, below, shows the throughput under similar conditions, except the full path name, starting at the root, is used for `open`. Although the throughput is lower, because of the directory searching, the normal kernel performance is still far better than the others.

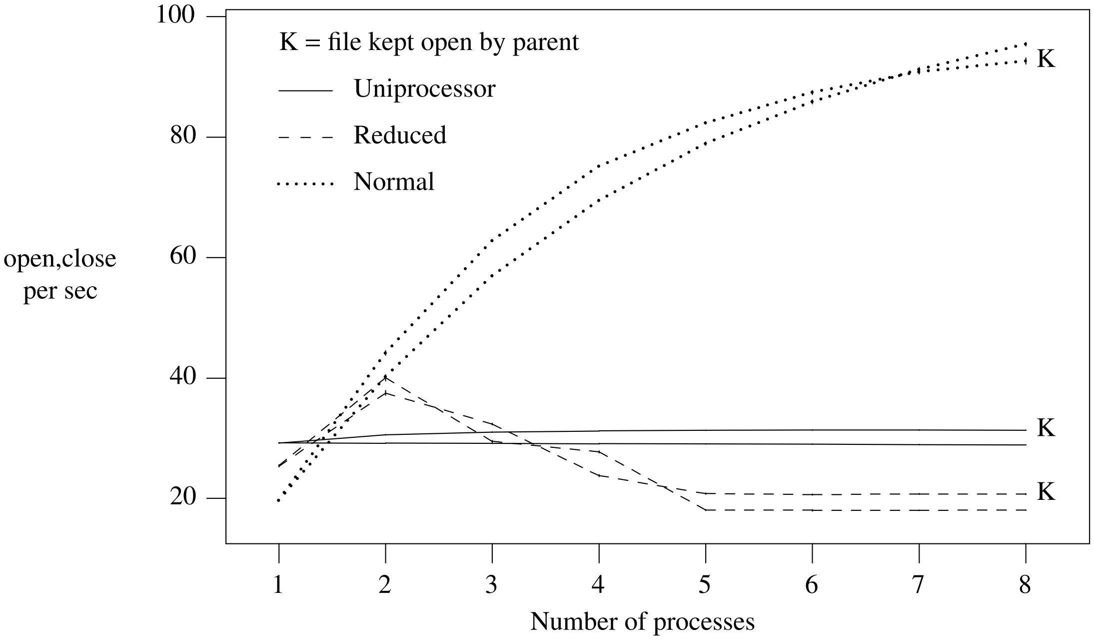


Figure 33: Open/Close Throughput.

Different files, same directory, full path names. Conditions are similar to Figure 32, except for full path names. The path names used, `"/a/sys/edler/mytest/#X`*n*`"`, where *n* is unique for each process, include a file system mount crossing, so six directory i-nodes must be examined for each `open`. Four of the six are kept open by the kernel because they are the root or current directory, or associated with the mount point; the other two are subject to allocation and deallocation in memory according to a reference count.

The experiments run for Figures 32 and 33 are certainly not representative of real workloads, as the only system calls they use are `open` and `close`. However, they do illustrate the limits of performance for these important system calls when processes are nearly independent. What happens to performance when there is dependence? We found greater sensitivity than we expected; the results are shown in Figures 34–37.

Figure 34, below, shows what happened when we first ran the experiment with all processes accessing the same file in the current directory (using the same kernel as in Figure 22, in section 8.3.1 on page 336). The performance advantage of the normal kernel was devastated, even when the file was kept open by the parent. What happened? In looking at the results of kernel instrumentation, we discovered serious contention for context-switching readers/writers locks associated with each i-node. It didn't take long to recall that the implementation of `close` always acquired an exclusive i-node lock when the file structure reference count reached zero (i.e., on every `close`, since this experiment doesn't include `dup`, or `fork`, or `spawn` calls).

We changed `close` to be more optimistic: no i-node lock is required if, when the i-node reference count is decremented, it doesn't go to zero. (Some additional cost must be paid when this optimism turns out wrong.) The algorithm is very similar to that presented in section 3.7.6 on page 134. The results of this experiment are given in Figure 35, on the next

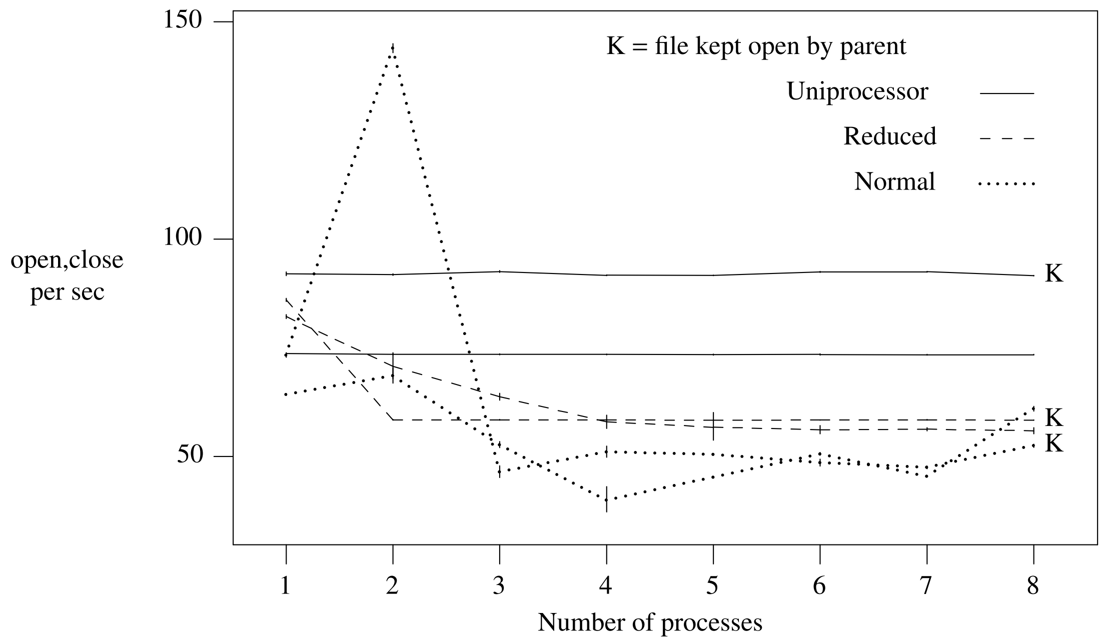


Figure 34: Open/Close Throughput.

Same file, current directory. The kernels are the same as reported in Figure 22, in section 8.3.1 on page 336. Conditions are similar to Figure 32, except that all `opens` are directed at a single file.

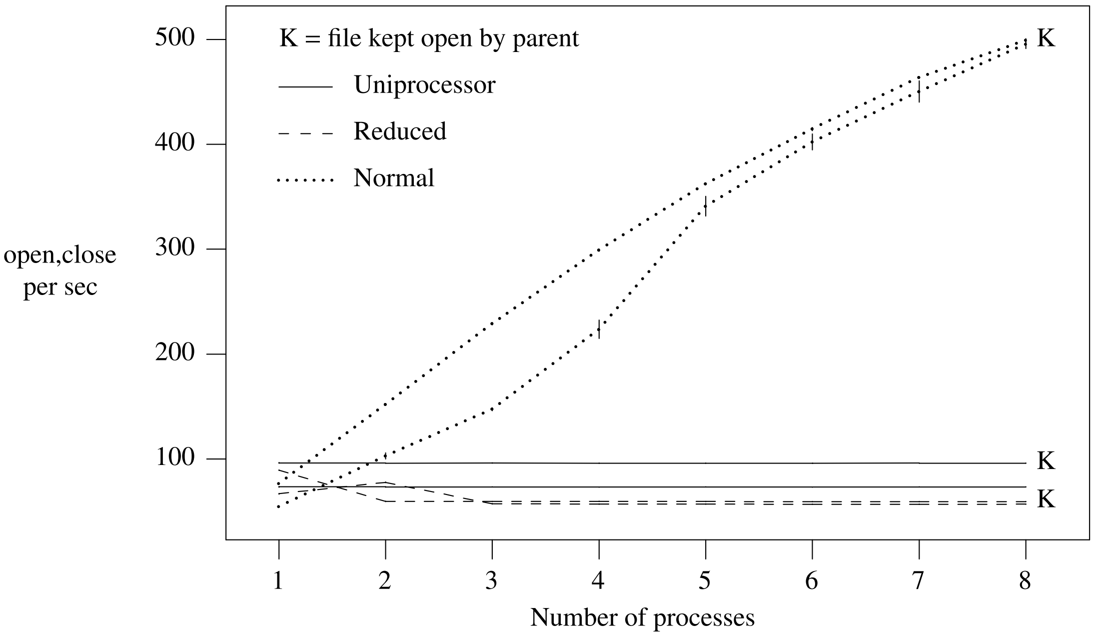


Figure 35: Open/Close Throughput.

Same file, current directory. This is the same experiment as shown in Figure 34, but the kernel's `close` routine has been altered to be more optimistic; it locks the i-node less often.

page. This time the results looked pretty good, but when we added the change to the kernel's idle loop described in section 8.3.1 on page 335 for Figure 23, we saw significant variations in the results for the normal kernel, with a somewhat worsening trend for larger numbers of processes, shown in Figure 36, on the next page. These results are caused by a fundamental race in the management of reference-counted objects identified by an external key (the i-node in this experiment). Substantial effort must be expended to deallocate such an object whenever its reference count reaches zero. If another attempt to reference the same object occurs quickly, it must first wait for the deallocation to complete, then go through the cost of allocation. This is a race condition because the extra cost would be avoided if the order of operations happened to be reversed. Unlike many race conditions encountered when developing parallel algorithms, this one cannot be algorithmically eliminated; it is fundamental to the problem of managing such objects. However, the cacheing technique described in section 3.7.6 on page 136 should be able to eliminate the problem in practice and provide a performance boost by avoiding some I/O as well.

When the reference count is being heavily manipulated, as it is in the open/close test program, the extra overhead can be substantial, mostly due to waiting for deallocation and reallocation. As the number of processes involved increases, the probability of any particular close decrementing the reference count to zero decreases, but the number of processors needing to wait in such a case goes up. In the cyclic test program, the waiting that results when a

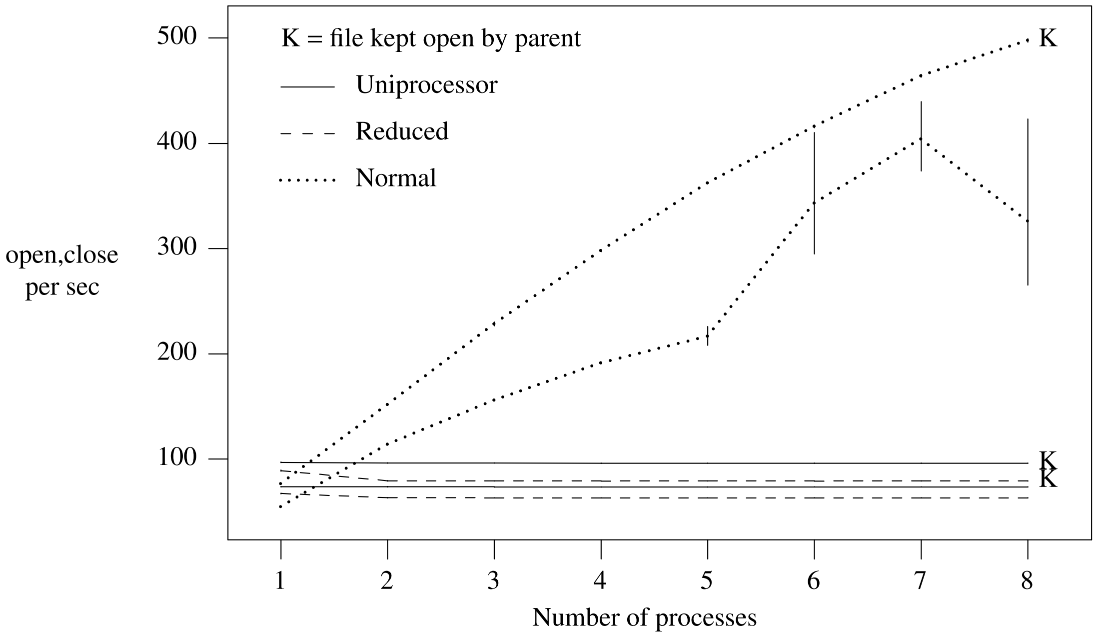
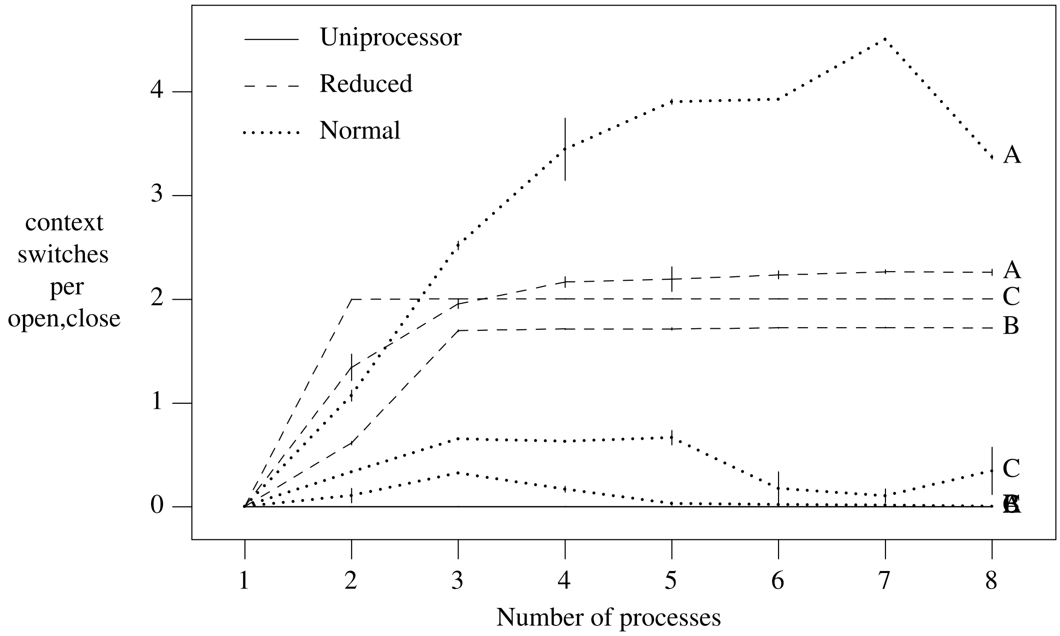


Figure 36: Open/Close Throughput.

Same file, current directory. This is the same experiment as shown in Figure 35, but the kernel's idle loop has been improved in the same manner as in Figure 23. Poorer performance than in Figure 35 for the normal kernel is caused by the cyclic test program's tendency to get into "lock step", incurring much more i-node deallocation/reallocation overhead and context-switching. This condition is more stable now that idle processors are more responsive.



A − experiment in Figure 34 (base system).
B − experiment in Figure 35 (A+improved close).
C − experiment in Figure 36 (B+improved idle).

Figure 37: Context-Switches per Open/Close.

Same file, current directory. There is no preemption in these experiments; all context-switching is caused by blocking. (The fact that the uniprocessor cases do no context-switching at all indicates that the scheduler is poorly tuned for uniprocessors.)

reference count goes to zero has a tendency to make the processes run in near “lock step” fashion. That is, between the beginning of the deallocation and the beginning of the next reallocation, there is sufficient time for nearly all other processes to “catch up”, and synchronize. Once most processes have synchronized after a reallocation, they are likely to remain nearly synchronized by the time they decrement the reference count, and it is likely the reference count will go to zero again. This synchronous behavior will continue until something disturbs it, such as a longer-than-usual random delay in some process; this prevents the reference count from dropping all the way to zero for awhile, and the “lock step” synchronization is lost. Thus, the impact of decrementing the reference count to zero is more prolonged than it would be in a less cyclic test.

The large variations exhibited in Figure 36 for some cases are due to the combination of the race and the magnification caused by cyclic waiting. Figure 37, above, shows the average number of context-switches per open and close pair when operating on the same file in the current directory.

Figure 38, on the next page, shows the throughput under the same circumstances as Figure 32, but including the cost of reading the files (7993 bytes each) before closing them.

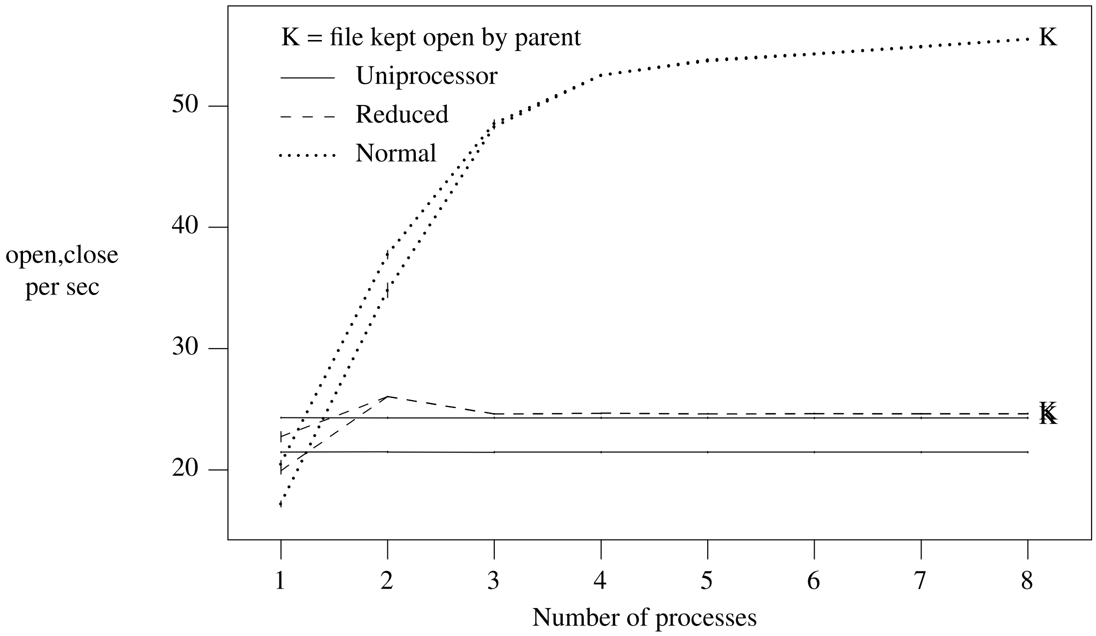


Figure 38: Open/Read/Close Throughput.

Different files, each 7993 bytes, current directory.

8.4. Temporary Non-Preemption

The results described so far have all pertained to the kernel rather than user programs. In the absence of system calls, user program performance depends on the basic machine execution speed, together with certain kernel costs: page faults, swapping, TLB misses, interrupt processing, scheduling, etc. In the case of Symunix-1 on Ultra-2, there are no page faults or TLB misses. One of the ways in which interrupt processing and scheduling can severely impact a user program is by interfering with busy-waiting synchronization. In section 4.6.4, we described the temporary non-preemption mechanism and how it avoids the worst effect of such interference, inopportune preemption.

David Wood wrote a program called *fifo* to test FIFO behavior of parallel list algorithms [207]. Because this program is very queue-operation intensive, it turns out to be a good illustration of the effects of inopportune preemption. We easily observe the difference between running *fifo* with the temporary non-preemption mechanism enabled and disabled.

Fifo exercises a list by using an even number of processes, half for insertions and half for deletions. The inserters insert (*pid*,*seq*) pairs, where *pid* is a process identifier and *seq* is a sequence number for that process. If a deleter gets two pairs with matching *pid*s and out-of-order *seq*s, a violation of the serialization principle has been detected.

The particular queue algorithm used for this experiment directs each operation (insert or delete) to one of P ordered sublists (P = number of processors), and enforces serialization on the sublists.[178] Directing each operation to a sublist is done in round-robin fashion by using Fetch&Add to increment a *head* or *tail* counter (deletes increment *head*, inserts increment *tail*). To ensure FIFO ordering, the algorithm makes sure that inserts and deletes directed to a sublist are accomplished in the order indicated by the old value of the head or tail pointer just incremented. $Insert_j$ cannot proceed until $insert_{j-P}$ has completed, and similarly for deletes. It is the enforcement of this ordering that can cause performance anomalies.

Once a process increments the head or tail counter, all subsequent like operations directed to the same sublist become dependent on its completion. Two things limit the negative effect of this serialization:

(1) the request rate for like operations on the queue, and

(2) the number of other sublists, $P-1$, to which additional operations will be directed with no dependency on this one.

If unrestricted preemption is allowed, a process may experience extreme delay after incrementing the counter but before completing the insert or delete. (Even assuming there is no swapping to secondary storage, the delay may last several seconds.) Concurrent operations directed to other sublists can proceed, but 1 of every P like operations will be assigned to the same sublist and "get stuck" waiting for the preempted process to finish. (Even dissimilar operations may be "stuck", since the sublist itself is a linked list protected by a binary semaphore.) Because the "stuck" processes are busy-waiting, they continue to consume the very resource needed by the preempted process. Temporary non-preemption helps by postponing preemption until after any inter-process dependencies are satisfied.

Clearly another approach to solving this problem would be to use a context-switching synchronization method, or a hybrid of busy-waiting and context-switching (called "two-phase blocking" by Ousterhout [160]). Temporary non-preemption allows the use of busy-waiting, which is simpler and typically has lower overhead than other methods. In addition, many context-switching methods are difficult or impossible to implement without themselves making some use of busy-waiting. Even a little busy-waiting can be a problem if arbitrary preemption is allowed.

Experimental results are given in Figure 39, on the next page. The performance anomalies caused by inopportune preemption can be dramatic, as shown in this extreme example. Temporary non-preemption essentially eliminates the problem completely.

8.5. Extended Workload Measurements

So far, we have only looked at synthetic workloads, designed to illuminate the cost of certain primitive operations (`spawn`, `open`) or to show the significance of a particular optimization (temporary non-preemption). In this section, we present some data collected from "live" system operation with real users. We collected data on two Ultra-2 systems for several months in 1989 and 1990, along the general lines described in section 8.2. During this time, the systems were in regular use by several users developing parallel programs.[179] This was charac-

[178] This queue (without temporary non-preemption) is called "queue" by Wood [207].

[179] The users were mostly independent, and no high-level log of their activities was maintained.

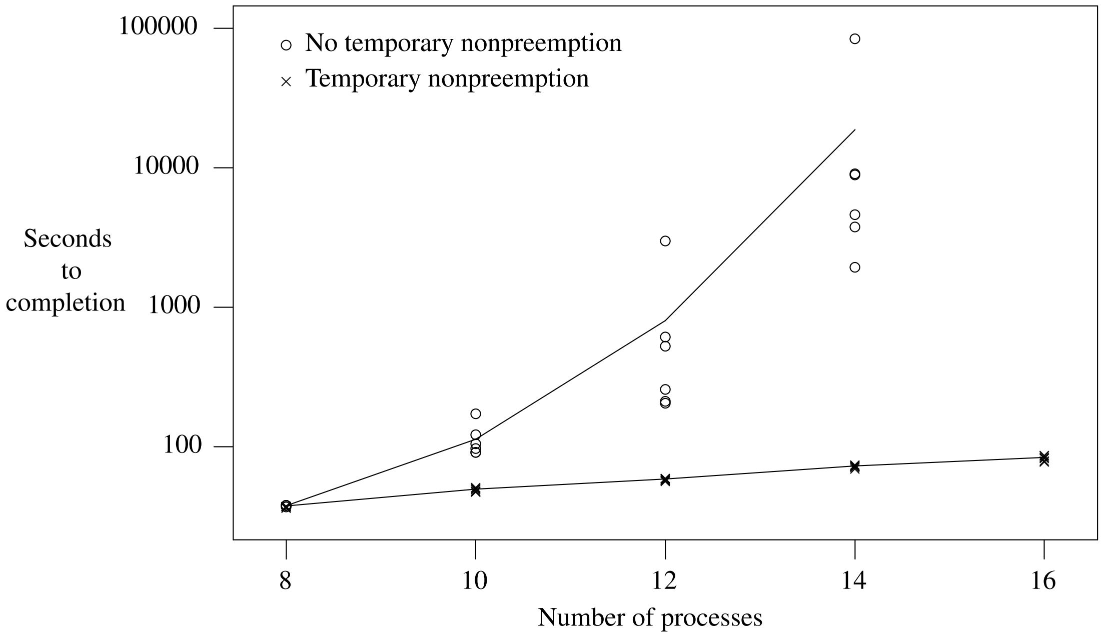


Figure 39: Effect of Temporary Non-Preemption.

The test program is very queue-operation intensive. Equal numbers of processes are inserting to and deleting from a single queue. The machine has eight processors, so serious anomalies only begin with more than eight processes when there are no other users. Measurements with more than 14 processes were impractical without temporary non-preemption. The results of six test runs are plotted with marks, and the lines show the averages (the standard deviation is so large without temporary non-preemption that showing it as a vertical bar would be distracting).

Sample	*Machine Name*	*Date*	*Time*
A	ultdev	6/14/89	8:00–12:00
B	ultdev	3/30/90	8:00–12:00
C	ultdev	6/7/89	12:00–16:00
D	ultdev	8/10/89	12:00–16:00
E	ultra	10/17/89	20:00–24:00
F	ultra	4/12/90	16:38–20:00
G	ultra	4/27/89	8:00–12:00
H	ultra	4/22/89	12:00–16:00

Table 39: Live Data Samples.

These intervals were arbitrarily selected from among the busiest ones recorded in 1989 and 1990.

terized by interactive use of editors, compilers, debuggers, short testing or timing runs of the programs being developed, and use of *uucp* for file transfers. Generally the systems were lightly loaded during this period, with no more than 2 or 3 users, and often 1 or less. Occasionally, longer parallel program runs were made on an otherwise idle machine; these were generally compute-intensive, with little OS intervention.

Sample	*idle %*	*system call*	*context switch*	*disk I/O*	`iget`	`iget` *hit %*	`iget` *retry*
A	87.2	5741371	10672	4660	8614	47.1	0
B	13.2	3414158	446089	191020	497774	81.0	18
C	38.9	755608	97024	38351	29513	47.9	0
D	0	218106	20912	10916	11333	49.1	0
E	39.7	2201344	873647	4640	8531	45.2	0
F	96.4	626512	40291	7851	13770	49.7	0
G	?	7797840	18271	5386	10090	47.8	0
H	?	5190968	18230	8118	9495	45.8	0

• *idle %* is the percentage of total processor time spent in the idle loop. A kernel bug prevented accurate idle time measurement for samples *G* and *H*.
• *system call* is the total number of system calls.
• *context switch* is the total number of context-switches.
• *disk I/O* is total number of disk reads and writes.
• `iget` is the total number of i-node look-ups (similar to §3.7.6/p129).
• `iget` *hit %* is the percentage of `iget` calls to succeed without allocation.
• `iget` *retry* is the number of `iget` calls that tried and failed to upgrade the hash bucket lock from reader to writer.

Table 40: Sample Measurements of General Activity.

Several measurements of general system activity are shown for each sample.

Data was collected at approximately 4-hour intervals. We selected four such samples from each system to present here; the selections were made from among those that were busiest, measured in either high system call usage or low idle time. The samples, which we label *A* through *H* for convenience, were taken as shown in Table 39, on the previous page.

Table 40, on the previous page, shows several measurements of general system activity for each sample. Table 41, on the next page, shows the ten most heavily used system calls for each sample. The results are not very surprising, with `read` being the most popular system call in most samples. Sample *B* stands out for having a much higher proportion of `writes` than the others, and sample E shows an unusually large number of `spawns`. The popularity of `getpid`, especially in sample *E*, is mildly surprising, since the system was enhanced earlier in 1989 to eliminate the need for this traditional system call to trap into the kernel except in the most obscure cases.[180] Presumably most of the exhibited `getpid` calls are from executables that predate the change. Sample *E* is also unusual in the high frequency of `spawn` calls.

The remaining data we present is in the form of histograms, as described in section 8.2. We choose to present this data in tabular form rather than graphically, as in other sections; composite graphs of all eight samples would be too cluttered for easy study, and separate graphs would take more space and make the samples harder to compare.

The tty lock histogram records waiting time for a busy-waiting lock (similar to `bwlock`, in section 3.5.3) associated with each serial communications device ("tty" being an abbreviation for teletype). The histogram data is shown in Table 42, on page 359; the bottom row shows the total number of times this lock was granted without waiting, the next row up shows the number of times the lock was granted after one iteration of the busy-waiting loop (i.e., the second attempt succeeded), and each higher row gives counts for twice as many iterations as the previous. Thus we can see that on one occasion, in each of *C*, *E*, and *F*, some processor was delayed 2^{13} iterations. As shown in Figure 20, this corresponds to at least 128 milliseconds, a rather considerable delay! These eight samples give us some idea of the kind of delays actually experienced during average use of the Ultra-2 prototypes; such extreme values are not typical. The tty lock is generally used to coordinate access to data shared between the "top" and "bottom" halves of the driver, since they may execute on different processors. The major concern with such long lock delays in a driver would be excessive interrupt masking; this problem is minimized in Symunix-1 by the use of separate hard and soft interrupt handlers. The hard interrupt handler does hardly more than simulate DMA for this "character at a time" device, and it does its work without touching the tty lock, so character interrupt response is very good. There were only 14 character overruns in all of the 8 sample periods, despite the large number of lock delays in excess of the 1 millisecond intercharacter arrival time (assuming 9600 bits per second).

It appears that the excessive waiting in some of these samples was caused primarily by hard interrupts interrupting the soft interrupt handler. These hard interrupts could be for the very same device, a different one, or the 16Hz clock interrupt. Both hard and soft tty interrupts can execute for variable amounts of time, handling more than one character.

[180] The obscure cases occur when a signal is delivered during a 2 or 3 instruction window surrounding a trap instruction for `fork` or `spawn` and the signal handler makes a `getpid` call.

A		*B*		*C*		*D*	
total	5741371	*total*	3414158	*total*	755608	*total*	218106
`read`	5720321	`read`	1292047	`mwait`	559508	`alarm`	76807
`alarm`	4219	`write`	1240763	`read`	47087	`read`	62370
`signal`	3632	`signal`	165945	`close`	32851	`signal`	41935
`write`	1972	`close`	150425	`lseek`	19243	`write`	15439
`close`	1768	`exec`	99158	`write`	18441	`close`	5298
`ioctl`	1048	`brk`	85190	`alarm`	16957	`lseek`	3279
`pause`	969	`getpid`	42947	`signal`	16765	`ioctl`	1196
`open`	857	`dup`	42800	`getpid`	8946	`brk`	1177
`time`	632	`fstat`	42332	`exit`	8302	`dup`	1169
`brk`	554	`ioctl`	36478	`brk`	3304	`open`	1091

E		*F*		*G*		*H*	
total	2201344	*total*	626512	*total*	7797840	*total*	5190968
`close`	1172466	`alarm`	239960	`read`	7765723	`ioctl`	4922672
`getpid`	390640	`read`	201426	`alarm`	7226	`read`	241196
`exit`	390490	`signal`	122179	`signal`	6479	`signal`	4757
`spawn`	55742	`write`	44791	`write`	3836	`alarm`	4051
`mwait`	55742	`lseek`	4088	`close`	2646	`close`	3499
`alarm`	46940	`close`	2509	`ioctl`	2348	`write`	3304
`read`	33345	`time`	1021	`pause`	1333	`lseek`	2165
`signal`	32101	`stat`	954	`open`	1011	`brk`	1052
`pause`	15345	`open`	919	`brk`	658	`open`	938
`write`	1372	`ioctl`	757	`stat`	626	`pause`	855

Table 41: Most Used System Calls.

The ten most heavily used system calls for 8 sample time periods. `userblock` is a system call to block a process; an experiment that helped lead to ksems (§7.3.4). The numbers given are the actual number of times each system call was executed during the interval; *total* gives the sum for all system calls (not just the ten most popular). (These are the same as the *system call* values shown in Table 40.)

	A	B	C	D	E	F	G	H
tty			1		1	12		
lock		5	1		5	1053		
	6	25	4	13	13	6283		3
	87	97	60	129	50	15684	3	17
	152	179	145	354	183	20005	9	34
	119	301	164	725	134	16958	14	34
	101	210	147	760	80	20394	12	19
	237	244	195	1128	123	26314	29	26
	599	486	442	2080	475	72892	64	143
	241	238	186	879	151	33041	40	50
	209	139	155	479	149	14851	23	54
	148	60	88	288	85	8632	8	17
	54	38	59	185	55	5060	8	16
	164	224	181	679	163	20936	34	41
	81089	1989853	635032	543836	137906	1474063	85465	5063125

Table 42: Histograms of TTY Lock Waiting Time.

Several possible solutions are apparent and should be explored, including:

- Hard interrupts could be masked when the tty lock is to be held for a time so brief that hard interrupt latency would not be adversely affected. This would prevent hard interrupts from lengthening tty lock holding times in those cases.
- Code that holds the tty lock could be restructured to more tightly bound the amount of work done when another processor is waiting for the lock.
- A different form of synchronization could be used between the main-line code and the soft interrupt handlers. Interactions could be based on highly parallel list structures or on non-blocking algorithms.

The pdp11 q lock is another busy-waiting lock, used to serialize access to the queue of outstanding disk I/O requests; Table 43, below, shows the waiting time data collected during the sample intervals. This is not a bottleneck, since the PDP-11 I/O is serialized anyway in

	A	B	C	D	E	F	G	H
pdp11		27	2					
q		197	4		6			
lock		436	8		2			
		324	14	1	6	1		2
		170	7	1	6	1	1	0
	1	123	10	0	3	0	1	1
	2	122	10	1	2	0	0	0
	17	3008	64	19	37	15	11	9
	16022	642799	151464	40763	43102	39755	18954	31395

Table 43: Histogram of PDP-11 Queue Lock Waiting Time.

the Ultra-2 prototype. On other systems, where physical I/O is (hopefully) not serialized, there should be a separately locked queue for each independent I/O device. The greater delays experienced by sample *B* simply reflect a larger amount of PDP-11 I/O.

Table 44, below, shows the waiting time experienced for the malloc lock, which serializes accesses to the `swapmap`, the same data structure used for the first-fit swap space allocator in Seventh Edition UNIX. Again, this is not a bottleneck, since actual swapping is serialized at the PDP-11 on this machine. Superior alternatives are required for use in systems with greater I/O capability, such as discussed in section 6.2.2 and in section 5.3.4 on page 244.

We have already seen callouts, in section 8.3.1 on page 339, but that was the lock for the pool of available callout structures. During the sample periods $A-H$, the histogram data for the available callout pool lock was combined with that of several other pools (p363). Table 45, below, reports on waiting times for the locks protecting the per-processor lists of pending callouts. As the data shows, these locks have fairly low contention and do not constitute a bottleneck. The primary use of callouts in Symunix-1 is provided by the APC mechanism in Symunix-2 (§3.6), which can be implemented in roughly the same way.

The time brwlock is a busy-waiting readers/writers lock, in the style of `bwrwlock`, in section 3.5.4. Table 46, on the next page, shows the waiting times experienced during the sample intervals. This lock protects the current absolute time, recorded as a two-word

	A	*B*	*C*	*D*	*E*	*F*	*G*	*H*
malloc		2						
lock		10						
		6						
		8					1	
		114					0	
		99					0	
	1	75	1				0	
	1	97	2	1			0	
	1	62	2	0			0	
	0	1325	23	0	12		0	
	1483	165706	4100	1699	3600	1868	1439	1584

Table 44: Histogram of Malloc Lock Waiting Time.

	A	*B*	*C*	*D*	*E*	*F*	*G*	*H*
callout		2						
lock		1		1		5		
		0		0		18		
		23		2	1	240		1
	4129820	4141710	4128864	4130795	3614739	3521331	4128712	4128227

Table 45: Histogram of Callout Lock Waiting Time.

	A	*B*	*C*	*D*	*E*	*F*	*G*	*H*
time		1						
brwlock		13			7	2		
		7			0	0		
		4			0	0		
		4			0	1		
		28	3	1	19	1		
		59	4	0	17	1		
	23560	441925	96325	33237	116110	53091	24871	31088

Table 46: Histogram of Time Readers/Writers Lock Waiting Time.

structure, which is not atomically accessible. Only updating the time, a job done by the clock interrupt handler, requires the exclusive lock; all other accesses are performed while holding a read lock. We note, however, that Lamport [132] produced a much better solution, which requires no locks and only 3 reads or writes to access the time; there is no excuse any longer for doing it the way we did in Symunix-1.

The ihash brwlock corresponds to the `rwhash` array in section 3.7.6 on page 128. Table 47, below, shows the waiting times experienced for this lock in Symunix-1. We see that a small but worrisome percentage of lock attempts experienced substantial delays, especially in sample *B*. This is almost certainly due to the fact that this busy-waiting readers/writers lock is used without interrupt masking (it isn't necessary for correctness−interrupt handlers don't touch i-nodes directly). This works fine most of the time, probably to the benefit of interrupt latency, but causes problems under high interrupt loads. The obvious solution is to mask soft interrupts when accessing this lock, as we indicate in section 3.7.6.

	A	*B*	*C*	*D*	*E*	*F*	*G*	*H*
ihash		1						
brwlock		0						
		13						
		296						
		220		1				
	1	145	1	0		1		
	0	95	1	0		0	1	
	0	63	0	0		0	0	
	0	54	0	0		1	0	
	0	30	0	0		0	0	
	0	289	0	0		1	0	
	0	89	1	0	1	1	0	
	29648	717975	75902	34883	29732	38416	32760	32101

Table 47: Histogram of Inode Hash Readers/Writers Lock Waiting Time.

	A	*B*	*C*	*D*	*E*	*F*	*G*	*H*
i-node			3					
rwlock			2					
			1					
		1	1					
		3	2	2		8		
	2	4	14	0	2	5		1
	1	34	12	0	1	4		0
	1	526	1	1	1	4	1	3
	5	5215	8	5	0	3	0	3
	389	36768	57	39	160	20	45	32
	5720271	1981570	78825	33152	37623	41525	27492	24200

Table 48: Histogram of Inode Readers/Writers Lock Waiting Time.

The i-node rwlock is a context-switching readers/writers lock (using an inferior algorithm to that presented in section 4.5.2); we show waiting times in Table 48, above. Delays are counted in terms of clock ticks (1/16 seconds) rather than busy-waiting iterations. An i-node is exclusively locked when writing the data in the file or modifying the file attributes (this results in more serialization than the improved scheme described in section 6.2.4 on page 261 for Symunix-2).

Buffer management in Symunix-1 is not as described in section 6.1.2. The algorithm used is similar to that presented in section 3.7.6 on page 136, but with `NRW_SEARCHHASH` set to 1. Table 49, below, shows the waiting times experienced for this 1 global lock; certainly there is room for improvement, as illustrated by the heavy-I/O sample, *B*.

	A	*B*	*C*	*D*	*E*	*F*	*G*	*H*
buf	1	70	1		1	2		
brwlock	2	513	4	1	3	4	1	
	5	3610	8	14	4	11	2	2
	0	50240	19	4	5	9	3	2
	6	73432	21	13	21	19	1	3
	38	107902	83	60	49	38	7	7
	45	120645	125	67	53	45	13	19
	64	102385	128	81	49	70	18	16
	75	77246	145	59	54	63	29	24
	48	53643	129	56	36	49	18	10
	131	183449	235	100	111	123	36	40
	205	270714	315	172	183	151	48	48
	59963	5578192	362060	120389	62990	114231	74350	82202

Table 49: Histogram of Buffer Readers/Writers Global Lock Waiting Time.

	A	*B*	*C*	*D*	*E*	*F*	*G*	*H*
qlist		4						
lock		8						
		24						
		51				1		
		21			1	0		
		16			0	0		
		4			1	0		
		724	9		39	3	1	
	24601	1564601	298067	65093	5311412	89657	49678	56879

Table 50: Histogram of Queue Sublist Lock Waiting Time.

The qlist lock histogram records busy-waiting times for sublists of the parallel-access list in Symunix-1; the data is presented in Table 50, above. A single algorithm is used for all such lists, with the exception of unordered lists (pools) and multi-queues. The algorithm used is not FIFO, but is starvation free; it is essentially the `dafifo` algorithm in section 3.7.1 on page 82. The data structure consists of some number of serially-accessible sublists, each protected by a busy-waiting lock. Insert and delete operations on the overall list are directed to the various sublists by using Fetch&Increment on tail and head counters, respectively. We can see from the histogram data that this is an effective mechanism for avoiding lock contention most of the time.

Table 51, below, reports waiting times for the poollist lock, which is similar to the qlist lock because the two list algorithms use essentially the same structure. In the pool, each sublist is singly-linked and treated as a LIFO, instead of being doubly linked and organized as a FIFO. When these samples were taken, waiting times for all such lists were recorded together. During subsequent studies, the level of detail was increased, leading to separate waiting times for each pool, as shown in Figure 25.

Each process in Symunix-1 has a busy-waiting readers/writers lock, in contrast to the simple lock used for Symunix-2 activities (§4.4.4/p165). We report waiting times for this lock

	A	*B*	*C*	*D*	*E*	*F*	*G*	*H*
poollist						1		
lock	1		1	7		8		
	0	5	4	18		19		3
	1	5	14	9	2	14		2
	13	8	28	24	2	23	14	7
	19	3	40	62	7	23	47	22
	578	3083	2147	1337	10458	1499	4119	1195
	454876	825189	500619	458562	1282847	434657	452999	455506

Table 51: Histogram of Pool Sublist Waiting Time.

	A	*B*	*C*	*D*	*E*	*F*	*G*	*H*
proc		3	1		983			
brwlock	1	475	16	2	2917	1		
	1	992	18	1	4686	62		
	12	656	16	12	5182	80	3	
	4	369	21	7	3601	74	5	2
	22	298	20	4	3159	16	13	1
	14	213	18	4	2735	7	9	1
	15	161	37	2	8588	2	13	1
	73	1836	73	22	12833	262	151	8
	11806989	14126803	1151815	5319321	1031148	3008908	8348634	5663663

Table 52: Histogram of Process Readers/Writers Lock Waiting Time.

in Table 52, above. The shared lock is used by children when referencing their parent during termination; since many children could terminate at once, this was an anti-bottleneck measure. The use of mechanisms in Symunix-2 such as visit lists (§3.7.3) helps to eliminate contention for this lock, which is why it has been replaced with a simpler lock in Symunix-2. The higher incidence of lock delays in samples *B* and *E* is probably due to an overall higher rate of interrupts causing delays in code holding the proc lock with only soft interrupts masked. Many such sections could be changed to block hard interrupts also, without significantly affecting hard interrupt latency.

As described in section 3.1 on page 43, some useful Fetch&Φ operations may not be supported directly by hardware; such is the case for the Ultra-2 prototype, which supports only Fetch&Add. A few others are provided by software, using an array of busy-waiting locks to serialize access for the synthesized operations; the address of the target memory location

	A	*B*	*C*	*D*	*E*	*F*	*G*	*H*
emul		3						
faop		3						
lock		89						
		313						
		182						
		310						
		452						
		607				1		
		709				0		
		121				0		
		81				1		
		7174				3		
	5725559	2891615	95084	86756	37468	254897	7771584	248612

Table 53: Histogram of Fetch&Φ Emulation Lock Waiting Time.

is hashed to locate a lock. Table 53, on the previous page, shows the histogram data for this lock. As with the i-node hash bucket lock discussed on page 361, these Fetch&Φ emulation locks are used without interrupt masking. The critical sections are so short that interrupt latency would not be adversely affected by masking all interrupts for their duration; this is probably the easiest way to avoid the excessive waiting times experienced in sample *B*.

8.6. Chapter Summary

This chapter provides some details about the executable Symunix-1 system on the Ultra-2 prototype, as distinct from the more sophisticated but still incomplete implementation of Symunix-2, which is the focus of most other chapters. We hope to have illuminated not only the general quality of the system, but also the costs and benefits of some of the specific techniques we advocate, especially highly parallel algorithms and data structures and temporary non-preemption.

We began with the raw hardware performance characteristics, since it is important to establish the proper context for evaluating the remaining performance measurements. The system in question is a mid-1980s vintage research prototype and must not be confused with more recent commercial products.

The meat of this chapter consists of measurements of `spawn`, `open`, temporary non-preemption, and "live" workloads. We faced two major limitations in our evaluations:

(1) *The lack of another operating system to compare with for the same hardware.* The major experimental method we used to attack this problem is reconfiguration of the kernel to replace some of our highly parallel algorithms with "lowly parallel" or serial versions. This technique is incomplete, because it doesn't address higher-level structural issues, but it does provide useful information.

(2) *The small number of processors available (8).* This makes it impossible to draw any firm conclusions about the ultimate scalability of our techniques, except for one thing: very little, if any, performance degradation results from using our highly parallel algorithms, even when we might expect lowly-parallel techniques to be justified.

It is much easier to draw a positive conclusion about temporary non-preemption: it's simple to implement and the benefits can be dramatic. The major shortcoming of this part of our evaluation is that we haven't explored the limits of its applicability. We would like to know, based on real applications, when we can expect temporary non-preemption to be effective, and when we'd be better off using context-switching synchronization.

The live workload measurements point up both strengths and weaknesses. It is gratifying that many of the locks, especially readers/writers locks, exhibit very low delays. The most significant problem area is interrupts that lengthen otherwise insignificant critical sections. Most of these appear to be solvable either with somewhat more aggressive masking of soft interrupts, or adoption of the more advanced algorithms and data structures designed for Symunix-2.

Chapter 9: Conclusion

In the course of designing, developing, using, and documenting highly parallel operating systems for over a decade, we've acquired a certain style and a bag of tricks; we hope this dissertation has expressed them in a clear and useful manner. Most of the chapters have concluded with summaries of their own, but in this final, brief, chapter, we summarize our key contributions and give our view of the lessons to be learned.

9.1. Contributions

Symunix itself, in both its forms, is a proof of concept: we wanted to show that highly parallel Fetch&Φ-based techniques are adequate to implement a complete general-purpose operating system, with additional support for highly parallel applications with demanding performance requirements. In this we have clearly succeeded (although final proof of the "highly" part still awaits the opportunity to build and run a very large system). Aside from being a minor landmark in the history of computing, however, the best hope for *lasting* contributions lies with some of the specific algorithms, data structures, and techniques we have developed. At the risk of being presumptuous, we identify the most significant ones in the remainder of this section.

Synchronization

We introduced the readers/readers locking mechanism as a generalization of the readers/writers lock, and gave a very low overhead bottleneck-free algorithm for it. We also showed a new but simple approach to implementing two-phase blocking, a hybrid between busy-waiting and context-switching synchronization.

We presented the design and implementation of ksems, a simple yet powerful new mechanism for inter-process context-switching synchronization. Although based on counting semaphores and implemented in the kernel, ksems are designed specifically to support higher-level coordination algorithms with a high-degree of concurrency and extremely low overhead. We gave several examples of synchronization algorithms using ksems, including a new algorithm for group lock, featuring a new form of fuzzy barrier.

We demonstrated a number of techniques for reducing overhead and generally improving performance, such as an aggressive two-stage implementation strategy, check functions, and specialized variants of standard operations designed to reduce overhead (e.g., "quick" waiting functions).

Data Structures

We showed how to take advantage of varying requirements in the design of efficient parallel-access lists, e.g., in our new set data structures, in new variations on existing highly parallel lists, and in the waiting lists of our ksem implementation. We showed how important it is to consider the effect of memory usage on overall scalability of parallel list algorithms.

We also introduced highly-parallel lists with new and different goals, such as visit lists, LRU lists, broadcast trees, and searching and cacheing structures. These new kinds of lists greatly expand the amount of operating system functionality that we can provide in a bottleneck-free manner.

Process and Memory Models

We showed how a combination of well-established concepts, asynchronous I/O and interrupts, can enhance the basic UNIX process model enough to support efficient user-mode thread systems and other highly-parallel applications, as well as lowly-parallel concurrent applications. In particular, we extended the notion of asynchronous I/O to include essentially any system call, and provided the new abstraction of asynchronous page faults. The resulting programming model is simpler, more general, and more familiar to systems programmers than a design based on kernel threads.

We showed how to generalize a private address space process model with features enabling it to emulate shared address spaces or even partially shared address spaces. We also showed how to emulate sharing of other resources, such as file descriptors, etc., among cooperating processes. The resulting system is much more flexible and powerful than a fixed model of kernel threads within shared resource domains.

We presented an implementation design based on the concept of *activities*, which are completely transparent to users, except as the force behind asynchronous system calls and page faults. Although superficially similar in some ways to kernel threads, their transparency allows us to make certain optimizations that would not be possible with a more conventional thread model.

Scheduling and Resource Management

We showed how unprivileged users may be given the ability to prevent preemption for a limited period of time, and we presented quantitative evidence for its effectiveness. Using a similar implementation strategy (a small set of communication variables shared between the user and kernel), we showed how to provide signal masking without system calls.

We introduced the asynchronous procedure call (APC) abstraction, a simple model for interprocessor interrupts. We also presented a generalized form of APC, called random broadcasts, which is a very lightweight way to schedule and execute short but high priority tasks.

We also presented an improved parallel buddy system allocator, designed to reduce fragmentation by merging requests.

9.2. Lessons

In this section, we go further out on a limb in an effort to pull together some deeper conclusions:

- The unix process model is well suited for parallel programming; many of the basic elements are there.
- The importance of careful implementation in synchronization code, in order to achieve scaleability and low overhead, cannot be overstressed. The modular approach provides a way to improve portability without sacrificing performance.
- The self-service paradigm, and the use of centralized yet highly parallel data structures and algorithms, has been a success.

9.3. Future Work

Most of the chapters included sections giving a few of the ideas we haven't been able to follow up on yet. Overall, there are some obvious areas to work on, such as the following:

- We hope to produce quantitative results, like those in chapter 8, for Symunix-1 running on the Ultra-3 prototype. This will be very important because of Ultra-3's combining network and somewhat larger number of processors.
- We would like to complete and demonstrate Symunix-2.
- Investigate hardware/software tradeoffs by running algorithms such as we have presented here, or even the whole operating system and some small applications, on several different simulated machines within our target class. We are currently developing the Molasses simulator [73] for this and other purposes.

We also believe that many of the ideas embodied in Symunix-2 can be translated and applied to other systems.

References

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young, "MACH: A New Kernel Foundation for UNIX Development", *Proc. USENIX Conference*, pp. 93−112 (June, 1986). Cited on pages 18 and 19.

[2] Advanced Micro Devices, Inc., *Am29050 Microprocessor User's Manual*, AMD, Sunnyvale, California (1991). Cited on pages 160 and 274.

[3] Eric P. Allman, "−me Reference Manual", in *UNIX User's Supplementary Documents, 4.3BSD* (April, 1986). Cited on page vi.

[4] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith, "The Tera Computer System", *Proc. International Conference on Supercomputing*, pp. 1−6 (June, 1990). Cited on page 6.

[5] American National Standards Institute, *ANSI X3.159-1989: American National Standard for Information Systems − Programming Language − C*, ANSI, New York (December 14, 1989). Cited on pages 41, 50, 70, 71, 119, and 271.

[6] Thomas E. Anderson, "The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors", *IEEE Transactions on Parallel and Distributed Systems* **1** (1), pp. 6−16 (January, 1990). Cited on pages 21, 22, and 53.

[7] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy, "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism", *Proc. 13th ACM Symposium on Operating Systems Principles (SOSP)* (October, 1991). Cited on pages 20, 25, 34, and 273.

[8] Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy, "The Performance Implications of Thread Management Alternatives for Shared Memory Multiprocessors", *IEEE Transactions on Computers* **38** (12), pp. 1631−1644 (December, 1989). Cited on pages 20, 24, and 34.

[9] Ziya Aral, James Bloom, Thomas Doeppner, Ilya Gertner, Alan Langerman, and Greg Schaffer, "Variable Weight Processes with Flexible Shared Resources", *Proc. USENIX Conference*, pp. 405−412 (Winter, 1989). Cited on page 19.

[10] J. S. Arnold, D. P. Casey, and R. H. McKinstry, "Design of Tightly-Coupled Multiprocessing Programming", *IBM Sys. J.* **13** (1), pp. 60−87 (1974). Cited on page 20.

[11] AT&T, *UNIX System V − Release 2.0 Support Tools Guide*, AT&T (April, 1984). Cited on page 321.

[12] AT&T, *System V Interface Definition, Issue 2*, AT&T (1986). Cited on page 276.

[13] AT&T, *UNIX System Documenter's Workbench Software (Release 3.1)* (1990). Cited on page vi.

[14] M. J. Bach and S. J. Buroff, "The UNIX System: Multiprocessor UNIX Systems", *AT&T Bell Laboratories Tech. J.* **63** (8), pp. 1733−1750 (October, 1984). Cited on pages 4 and 31.

[15] Maurice J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall (1986). Cited on pages 94, 157, 165, 251, 252, 257, 258, 264, 337, and 339.

[16] Robert Baron, Richard Rashid, Ellen Siegel, Avadis Tevanian, and Michael Young, "MACH-1: A Multiprocessor-Oriented Operating System and Environment", pp. 80−99 in *New Computing Environments: Parallel, Vector and Systolic*, ed. Arthur Wouk, SIAM (1986). Cited on page 19.

[17] J. M. Barton and J. C. Wagner, "Beyond Threads: Resource Sharing in UNIX", *Proc. USENIX Conference*, pp. 259−266 (February, 1988). Cited on page 19.

[18] BBN Advanced Computers, *Inside the Butterfly Plus* (October, 1987). Cited on pages 6 and 35.

[19] Bob Beck and Bob Kasten, "VLSI Assist in Building a Multiprocessor UNIX System", *Proc. USENIX Conference*, pp. 255−275 (Summer, 1985). Cited on pages 32, 45, 187, 274, and 329.

[20] Bob Beck and Dave Olien, "A Parallel Programming Process Model", *Proc. USENIX Conference*, pp. 83−102 (Winter, 1987). Cited on page 32.

[21] Bell Telephone Laboratories, *UNIX Time Sharing System: UNIX Programmer's Manual (Revised and Expanded Edition (Seventh Edition), Volumes 1 and 2)*, Holt, Rinehart and Winston (1983). Cited on pages 17 and 320.

[22] Bell Telephone Laboratories, *UNIX Research System Programmer's Manual, Tenth Edition (Volumes I and II)* (1990). Cited on page 222.

[23] Jon L. Bentley and Brian W. Kernighan, "GRAP − A Language for Typesetting Graphs", in *UNIX Research System Papers, Tenth Edition, Volume II*, Saunders College Publishing (1990). Cited on page vi.

[24] Wayne Berke, “ParFOR – A Parallel Extension to FORTRAN 77 User’s Guide”, NYU Ultracomputer Documentation Note #8, New York University (March, 1988). Cited on pages 146 and 313.

[25] Wayne Berke, “ParFOR – A Structured Environment for Parallel FORTRAN”, NYU Ultracomputer Note #137, New York University (April, 1988). Cited on pages 146 and 313.

[26] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy, “Lightweight Remote Procedure Call”, *Proc. 12th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 102–113 (December, 1989). Cited on page 19.

[27] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon, “The Midway Distributed Shared Memory System”, *Proc. IEEE CompCon Conference* (1993). Cited on page 15.

[28] Ronald Bianchini, “Packaging Ultracomputers and Implementing Ultracomputer Prototypes”, NYU Ultracomputer Note #177, New York University (May, 1992). Cited on page 328.

[29] Ronald Bianchini, Susan Dickey, Jan Edler, Gabriel Goodman, Allan Gottlieb, Richard Kenner, and Jiarui Wang, “The Ultra III Prototype”, *Proceedings of the Parallel Systems Fair, 7th International Parallel Processing Symposium*, pp. 2–9 (April 14, 1993). Extended version available as NYU Ultracomputer Note #185. Cited on pages 5, 142, and 206.

[30] Kenneth Birman, André Schiper, and Pat Stephenson, “Lightweight Causal and Atomic Group Multicast”, *ACM Transactions on Computer Systems (TOCS)* **9** (3), pp. 272–314 (August, 1991). Cited on page 10.

[31] Kenneth P. Birman and Thomas A. Joseph, “Exploiting Virtual Synchrony in Distributed Systems”, *Proc. 11th ACM Symposium on Operating System Principles (SOSP)*, pp. 123–138 (November, 1987). Cited on page 10.

[32] A. D. Birrell, J. V. Guttag, J. J. Horning, and R. Levin, “Synchronization Primitives for a Multiprocessor: A Formal Specification”, *Proc. 11th ACM Symposium on Operating System Principles (SOSP)*, pp. 94–102 (November, 1987). Cited on pages 20 and 34.

[33] G. A. Blaauw and F. P. Brooks, Jr., “The Structure of System/360 – Part 1: Outline of Logical Structure”, pp. 695–710 in *Computer Structures: Principles and Examples*, ed. Siewiorek, Bell, and Newell, McGraw-Hill, New York (1982). Cited on page 17.

[34] David L. Black, “Scheduling and Resource Management Techniques for Multiprocessors”, Ph.D. Thesis, Carnegie Mellon University (July, 1990). Cited on page 25.

[35] David L. Black, Richard F. Rashid, David B. Golub, Charles R. Hill, and Robert V. Baron, "Translation Lookaside Buffer Consistency: A Software Approach", *Proc. 3rd International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 113−122 (April, 1989). Cited on page 211.

[36] D. G. Bobrow, J. D. Burchfiel, D. L. Murphy, and R. S. Tomlinson, "TENEX, a Paged Time Sharing System for the PDP-10", *Communications of the ACM* **15** (3), pp. 135−143 (March, 1972). Cited on page 266.

[37] D. E. Bodenstab, T. F. Houghton, K. A. Kelleman, G. Ronkin, and E. P. Schan, "The UNIX System: UNIX Operating System Porting Experiences", *AT&T Bell Laboratories Tech. J.* **63** (8), pp. 1769−1790 (October, 1984). Cited on page 18.

[38] Alan Borodin and John E. Hopcroft, "Routing, Merging, and Sorting on Parallel Models of Computation", *Proc. 14th ACM Symposium on Theory of Computing*, pp. 338−344 (1982). Cited on page 23.

[39] Per Brinch Hansen, "The Nucleus of a Multiprogramming System", *Commun. ACM* **13** (4), pp. 238−241 and 250 (April, 1970). Cited on pages 17 and 19.

[40] Per Brinch Hansen, *Operating System Principles*, Prentice-Hall (1973). Cited on page 17.

[41] Ray Bryant, Hung-Yang Chang, and Bryan Rosenburg, "Experience Developing the RP3 Operating System", *Computing Systems* **4** (3), USENIX Association (Summer, 1991). Cited on page 33.

[42] Douglas C. Burger, Rahmat S. Hyder, Barton P. Miller, and David A. Wood, "Paging Tradeoffs in Distributed-Shared-Memory Multiprocessors", *Proc. Supercomputing 94*, pp. 590−599 (1994). Cited on page 25.

[43] John B. Carter, John K. Bennett, and Willy Zwaenepoel, "Implementation and Performance of Munin", *Proc. 13th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 152−164 (October, 1991). Cited on page 15.

[44] David R. Cheriton, "The V Distributed System", *Communications of the ACM* **31** (3), pp. 314−333 (March, 1988). Cited on page 18.

[45] Jerome Chiabaut, *Private Communication* (1987). Cited on page 122.

[46] James Clark, "GNU Groff Document Formatting System", Version 1.09 (February, 1994). Cited on page vi.

[47] Frederick W. Clegg, "Hewlett-Packard's Entry into the UNIX Community", *Proc. USENIX Conference*, pp. 119−131 (January, 1983). Cited on page 18.

[48] Daniel S. Conde, Felix S. Hsu, and Ursula Sinkewicz, “ULTRIX Threads”, *Proc. USENIX Conference*, pp. 257–268 (Summer, 1989). Cited on page 20.

[49] Convex Computer Corporation, *Convex UNIX Programmer’s Manual, Version 6.0* (November 1, 1987). Cited on page 151.

[50] P. J. Courtois, F. Heymans, and D. L. Parnas, “Concurrent Control with Readers and Writers”, *Commun. ACM* **14** (10), pp. 667–668 (October, 1971). Cited on page 59.

[51] Cray Research, Inc., *UNICOS System Calls Reference Manual (SR-2012)* (March, 1986). Cited on page 151.

[52] Mark Crovella, Prakash Das, Czarek Dubnicki, Thomas LeBlanc, and Evangelos Markatos, “Multiprogramming on Multiprocessors”, Technical Report 385, University of Rochester (May, 1991). Cited on page 25.

[53] Peter J. Denning, “The Working Set Model for Program Behavior”, *Communications of the ACM* **11** (5), pp. 323–333 (May, 1968). Cited on page 231.

[54] L. Peter Deutsch and others, *Ghostscript Version 2.4* (1992). Cited on page vi.

[55] Robert B. K. Dewar and Matthew Smosna, “Taking Stock of Power Processing”, *Open Systems Today*, p. 49 (February 1, 1993). Cited on page 324.

[56] Susan R. Dickey and Richard Kenner, “Design of Components for a Low-Cost Combining Network”, *VLSI Design* **2** (4), pp. 287–303 (1994). Cited on page 144.

[57] Digital Equipment Corporation, *Alpha Architecture Handbook* (1992). Cited on page 21.

[58] E. W. Dijkstra, “Solution of a Problem in Concurrent Programming Control”, *Commun. ACM* **8** (9) (September, 1965). Cited on page 20.

[59] E. W. Dijkstra, “Cooperating Sequential Processes”, in *Programming Languages*, ed. F. Genuys, Academic Press, New York (1968). Cited on pages 17 and 55.

[60] E. W. Dijkstra, “The Structure of THE Multiprogramming System”, *Commun. ACM* **11** (5), pp. 341–346 (May, 1968). Cited on pages 17 and 55.

[61] E. W. Dijkstra, “Hierarchical Orderings of Sequential Processes”, *Acta Informatica* **1** (2), pp. 115–138 (October, 1971). Cited on page 58.

[62] Isaac Dimitrovsky, “A Short Note on Barrier Synchronization”, Ultracomputer System Software Note #59, New York University (January, 1985). Cited on pages 274, 302, and 303.

[63] Isaac Dimitrovsky, “Two New Parallel Queue Algorithms (Revised)”, Ultracomputer System Software Note #58, New York University (February, 1985). Cited on pages 82, 121, 283, and 289.

[64] Isaac Dimitrovsky, “A Group Lock Algorithm, With Applications”, NYU Ultracomputer Note #112, New York University (November, 1986). Cited on page 68.

[65] Isaac A. Dimitrovsky, “ZLISP − A Portable Lisp System”, Ph.D Thesis, Courant Institute, NYU (June, 1988). Cited on pages 23, 68, 274, 283, 289, and 302.

[66] Isaac A. Dimitrovsky, “The Group Lock and Its Applications”, *Journal of Parallel and Distributed Computing* **11**, pp. 291−302 (April, 1991). Cited on pages 23 and 68.

[67] Thomas Doeppner, Jr., “Threads: A System for the Support of Concurrent Programming”, Technical Report CS-87-11, Brown University Computer Science Department (June, 1987). Cited on pages 20 and 25.

[68] R. Draves and E. Cooper, “C Threads”, Technical Report CMU-CS-88-154, School of Computer Science, Carnegie-Mellon University (June, 1988). A revised version is also available, dated September 11, 1990. Cited on page 20.

[69] Peter Druschel, Larry L. Peterson, and Norman C. Hutchinson, “Beyond Micro-Kernel Design: Decoupling Modularity and Protection in Lipto”, *Proc. 12th International Conference on Distributed Computing Systems*, pp. 512−520 (June, 1992). Cited on page 19.

[70] Jan Edler, “Readers/Readers Synchronization”, Ultracomputer System Software Note #46, New York University (March, 1984). Cited on pages 4 and 64.

[71] Jan Edler, “An Event Mechanism for the Ultracomputer”, Ultracomputer System Software Note #48, New York University (September, 1984). Cited on page 4.

[72] Jan Edler, “Restricted Parallel Sets”, Unpublished paper (March, 1988). Cited on page 123.

[73] Jan Edler, “Molasses: An Ultracomputer Simulator (Preliminary Version)”, NYU Ultracomputer Note #189, New York University (March 8, 1994). Cited on page 369.

[74] Jan Edler, Allan Gottlieb, and Jim Lipkis, “Considerations for Massively Parallel UNIX Systems on the NYU Ultracomputer and IBM RP3”, *Proc. USENIX Association winter conf.* (1986). Cited on page 33.

[75] Jan Edler, Jim Lipkis, and Edith Schonberg, “Memory Management in Symunix II: A Design for Large-Scale Shared Memory Multiprocessors”, *Proc. USENIX Workshop on UNIX and Supercomputers*, pp. 151−168 (September, 1988). Also available as NYU Ultracomputer Note #135. Cited on page 20.

[76] Jan Edler, Jim Lipkis, and Edith Schonberg, "Process Management for Highly Parallel UNIX Systems", *Proc. USENIX Workshop on UNIX and Supercomputers*, pp. 1−18 (September, 1988). Also available as NYU Ultracomputer Note #136. Cited on pages 20 and 25.

[77] Philip H. Enslow, Jr., "Multiprocessor Organization − A Survey", *ACM Computing Surveys* **9** (1) (March, 1977). Cited on pages 15 and 16.

[78] R. S. Fabry, "Capability-Based Addressing", *Commun. ACM* **17** (7), pp. 403−412 (July, 1974). Cited on pages 8 and 28.

[79] W. A. Felton, G. L. Miller, and J. M. Milner, "The UNIX System: A UNIX System Implementation for System/370", *AT&T Bell Laboratories Tech. J.* **63** (8), pp. 1751−1768 (October, 1984). Cited on page 18.

[80] Gary Fielland and Dave Rogers, "32-Bit Computer System Shares Load Equally Among Up to 12 Processors", *Electronic Design* (September 6, 1984). Cited on pages 6, 45, 187, and 274.

[81] Eric J. Finger, Michael M. Krueger, and Al Nugent, "A Multiple CPU Version of the UNIX Kernel", *Proc. USENIX Conference*, pp. 11−22 (Winter, 1985). Cited on page 18.

[82] Michael. J. Flynn, "Very High Speed Computing Systems", *Proc. IEEE* **54**, pp. 1901−1909 (December, 1966). Cited on page 5.

[83] Bryan Ford and Jay Lepreau, "Evolving Mach 3.0 to a Migrating Thread Model", *Proc. USENIX Conference*, pp. 97−114 (January, 1994). Cited on page 20.

[84] Steven Fortune and James Wyllie, "Parallelism in Random Access Machines", *Proc. 10th ACM Symposium on Theory of Computing*, pp. 114−118 (1978). Cited on page 23.

[85] Eric Freudenthal, *Private Communication* (1990). Cited on pages 60 and 246.

[86] Eric Freudenthal and Allan Gottlieb, "Process Coordination with Fetch-and-Increment", *Proc. 4th International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 260−268 (April, 1991). Cited on pages 24, 40, 66, and 69.

[87] Eric Freudenthal and Olivier Peze, "Efficient Synchronization Algorithms Using Fetch&Add on Multiple Bitfield Integers", NYU Ultracomputer Note #148, New York University (February, 1988). Cited on page 64.

[88] George H. Goble and Michael H. Marsh, "A Dual Processor VAX 11/780", Tech. Report TR-EE 81-31, Purdue University (September, 1981). Cited on page 31.

[89] David Golub, Randall Dean, Alessandro Forin, and Richard Rashid, "UNIX as an Application Program", *Proc. USENIX Conference*, pp. 87–96 (Summer, 1990). Cited on page 19.

[90] James R. Goodman, Mary K. Vernon, and Philip J. Woest, "Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors", *Proc. 3rd International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 64–75 (April, 1989). Cited on pages 21 and 24.

[91] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir, "The NYU Ultracomputer – Designing a MIMD, Shared-Memory Parallel Machine", *IEEE Transactions on Computers* **C-32** (2), pp. 175–189 (February, 1983). Cited on page 23.

[92] Allan Gottlieb and Clyde Kruskal, "Coordinating Parallel Processors: A Partial Unification", *ACM Computer Architecture News* (October, 1981). Cited on page 22.

[93] Allan Gottlieb, B. D. Lubachevsky, and Larry Rudolph, "Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors", NYU Ultracomputer Note #16, New York University (December, 1981). Includes additional material beyond version printed in ACM TOPLAS, April, 1983. Cited on pages 22, 23, and 196.

[94] Allan Gottlieb, B. D. Lubachevsky, and Larry Rudolph, "Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors", *ACM Transactions on Programming Languages and Systems (TOPLAS)* **5** (2), pp. 164–189 (April, 1983). Cited on pages 4, 22, 23, 42, 51, 55, 58, 60, 61, 77, and 123.

[95] Gary Graunke and Shreekant Thakkar, "Synchronization Algorithms for Shared Memory Multiprocessors", *IEEE Computer* **23** (6), pp. 60–69 (June, 1990). Cited on pages 21 and 22.

[96] Albert G. Greenberg, Boris D. Lubachevsky, and Andrew M. Odlyzko, "Simple, Efficient Asynchronous Parallel Algorithms for Maximization", *ACM Transactions on Programming Languages and Systems (TOPLAS)* **10** (2), pp. 313–337 (April, 1988). Cited on page 235.

[97] Rajiv Gupta, "The Fuzzy Barrier: A Mechanism for High Speed Synchronization of Processors", *Proc. 3rd International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 54–63 (April, 1989). Cited on pages 69, 275, 304, and 305.

[98] Rajiv Gupta and Charles R. Hill, "A Scalable Implementation of Barrier Synchronization Using and Adaptive Combining Tree", *International Journal of Parallel Programming* **18** (3), pp. 161–180 (June, 1989). Cited on page 24.

[99] Erik Hagersten, Anders Landin, and Seif Haridi, “DDM−A Cache-Only Memory Architecture”, *IEEE Computer* **25** (9), pp. 44−56 (September, 1992). Cited on page 15.

[100] Jung Hamel, “The Operating System for the Uniprocessor and Dual Processor NYU Ultracomputer”, M.S. Thesis, Courant Institute, NYU (May, 1983). Cited on page 3.

[101] Malcolm C. Harrison, “Add-and-Lambda II: Eliminating Busy Waits”, NYU Ultracomputer Note #139, New York University (March, 1988). Cited on page 22.

[102] E. A. Hauck and B. A. Dent, “Burroughs’ B6500/B7500 Stack Mechanism”, pp. 244−250 in *Computer Structures: Principles and Examples*, ed. Siewiorek, Bell, and Newell, McGraw-Hill, New York (1982). Cited on page 26.

[103] John A. Hawley, III and Walter B. Meyer, “MUNIX, A Multiprocessing Version of UNIX”, M.S. Thesis, Naval Postgraduate School, Monterey, California (June, 1975). Cited on page 30.

[104] Maurice Herlihy, “A Methodology for Implementing Highly Concurrent Data Structures”, *Proc. 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pp. 197−206 (March, 1990). Cited on page 21.

[105] Maurice Herlihy, “Wait-Free Synchronization”, *ACM Transactions on Programming Languages and Systems (TOPLAS)* **13** (1), pp. 124−149 (January, 1991). Cited on page 272.

[106] Maurice Herlihy, “A Methodology for Implementing Highly Concurrent Data Objects”, Technical Report CRL 91/10, Digital Equipment Corporation (October 2, 1991). Cited on page 21.

[107] Maurice P. Herlihy and Jeannette M. Wing, “Axioms for Concurrent Objects”, *Proc. 14th ACM Symposium on Principles of Programming Languages (POPL)*, pp. 13−26 (1987). Cited on page 123.

[108] Dan Hildebrand, “An Architectural Overview of QNX”, *Proc. USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pp. 113−126 (April, 1992). Cited on page 18.

[109] Tom Hoag, “The HEP/UPX Operating System, Denelcor”, *IEEE Software* **2** (4), pp. 77−78 (July, 1985). Cited on page 18.

[110] C. A. R. Hoare, “Monitors: An Operating System Structuring Concept”, *Commun. ACM* **17** (10), pp. 549−557 (October, 1974). Corrigendum CACM **18**, 2. Cited on page 17.

[111] C. A. R. Hoare, “Communicating Sequential Processes”, *Commun. ACM* **21** (8), pp. 666−677 (August, 1978). Cited on page 17.

[112] IEEE, *Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Standard 754-1985)*, IEEE (August, 1985). Cited on page 274.

[113] IEEE, *ISO/IEC 9945-1 (ANSI/IEEE 1003.1): POSIX Part 1: Portable Operating System Interface (API) [C Language]*, IEEE, New York (1990). Cited on pages 18, 203, and 212.

[114] IEEE, *Draft Standard for Information Technology: POSIX 1003.4a/D7 (PTHREADS)*, IEEE, New York (April, 1993). The standard, now at draft 10 or beyond, is near final approval and will eventually be known as P1003.1c. Cited on page 20.

[115] Intel Corporation, *Pentium Family User's Manual, Volume 3: Architecture and Programming Manual* (1994). Cited on page 23.

[116] International Business Machines Corporation, "An Application-Oriented Multiprocessing System", *IBM Sys. J.* **6** (2), pp. 78–132 (1967). Cited on page 26.

[117] International Business Machines Corporation, *The IBM Power PC Architecture – A New Family of RISC Processors*, Morgan Kaufmann, San Mateo, CA (1994). Cited on page 21.

[118] E. H. Jensen, G. W. Hagensen, and J. M. Broughton, "A New Approach to Exclusive Data Access in Shared Memory Multiprocessors", Technical Report UCRL-97663, Lawrence Livermore National Laboratory (November, 1987). Cited on page 20.

[119] A. K. Jones, R. J. Chansler, Jr., I. Durham, K. Schwans, and S. R. Vegdahl, "StarOS: A Multiprocessor Operating System for the Support of Task Forces", *Proc. 7th ACM Symposium on Operating Systems Principles (SOSP)* (December, 1979). Cited on page 29.

[120] Anita K. Jones and Peter Schwarz, "Experience Using Multiprocessor Systems – A Status Report", *ACM Computing Surveys* **12**, pp. 121–165 (June, 1980). Cited on pages 17 and 29.

[121] Michael B. Jones, "Bringing the C Libraries With Us into a Multi-Threaded Future", *Proc. USENIX Conference* (Winter, 1991). Cited on page 3.

[122] Harry F. Jordan, "Special Purpose Architecture for Finite Element Analysis", *Proc. 1978 International Conference on Parallel Processing*, pp. 263–266 (August, 1978). Cited on pages 23, 49, and 188.

[123] William Joy and Mark Horton, "Ex Reference Manual, Version 3.7", in *UNIX User's Supplementary Documents, 4.3BSD* (April, 1986). Cited on page vi.

[124] Daniel P. Julin, Jonathan J. Chew, J. Mark Stevenson, Paulo Guedes, Paul Neves, and Paul Roy, "Generalized Emulation Services for Mach 3.0−Overview, Experiences and Current Status", *Proc. Second USENIX Mach Symposium*, pp. 13−26 (November, 1991). Cited on page 19.

[125] Anna R. Karlin, Kai Li, Mark S. Manasse, and Susan Owicki, "Empirical Studies of Competitive Spinning for a Shared-Memory Multiprocessor", *Proc. 13th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 41−55 (October, 1991). Cited on page 40.

[126] David Katsuki, Eric S. Elsam, William F. Mann, Eric S. Roberts, John G. Robinson, F. Stanley Skowronski, and Eric W. Wolf, "Pluribus − An Operational Fault-Tolerant Multiprocessor", pp. 371−386 in *Computer Structures: Principles and Examples*, ed. Siewiorek, Bell, and Newell, McGraw-Hill, New York (1982). Cited on page 17.

[127] David Keppel, "Tools and Techniques for Building Fast Portable Threads Packages", University of Washington Technical Report UWCSE 93-05-06 (1993). Cited on page 20.

[128] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice-Hall (1978). Cited on page 40.

[129] Steve Kirkendall, "Elvis, ex, vi, view, input − The Editor", UNIX man page for version 1.8.3 (June, 1994). Cited on page vi.

[130] Donald E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms (2nd Edition)*, Addison-Wesley, Reading, MA (1973). Cited on pages 84 and 125.

[131] Leslie Lamport, "A Fast Mutual Exclusion Algorithm", *ACM Transactions on Computer Systems (TOCS)* **5** (1), pp. 1−11 (February, 1987). Cited on page 20.

[132] Leslie Lamport, "Concurrent Reading and Writing of Clocks", *ACM Transactions on Computer Systems (TOCS)* **8** (4), pp. 305−310 (November, 1990). Cited on page 361.

[133] B. W. Lampson, "Dynamic Protection Structures", *Proc. AFIPS Fall Joint Computer Conference (FJCC)*, pp. 27−38, AFIPS Press (1969). Cited on pages 17 and 28.

[134] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley, Reading, Massachusetts (1989). Cited on pages 157, 165, and 252.

[135] David C. Lennert, "Decreasing Realtime Process Dispatch Latency Through Kernel Preemption", *Proc. USENIX Conference*, pp. 405−414 (Summer, 1986). Cited on page 9.

[136] S. Leutenegger, “Issues in Multiprogrammed Multiprocessor Scheduling”, Ph.D. Thesis, University of Wisconsin/Madison (August, 1990). Cited on page 25.

[137] Kai Li, “Shared Virtual Memory on Loosely Coupled Multiprocessors”, Ph.D. Thesis, Yale University (September, 1986). Cited on page 15.

[138] William Lonergan and Paul King, “Design of the B5000 System”, pp. 129−134 in *Computer Structures: Principles and Examples*, ed. Siewiorek, Bell, and Newell, McGraw-Hill, New York (1982). Cited on page 26.

[139] Tom Lovett and Shreekant Thakkar, “The Symmetry Multiprocessor System”, *Proc. 1988 International Conference on Parallel Processing*, pp. 303−310 (1988). Cited on pages 6, 45, 187, and 274.

[140] H. Lycklama and D. L. Bayer, “The MERT Operating System”, *Bell System Technical Journal* **57** (6), pp. 2049−2086 (July-August, 1978). Cited on page 18.

[141] R. A. MacKinnon, “Advanced Function Extended with Tightly-Coupled Multiprocessing”, *IBM Sys. J.* **13** (1), pp. 32−59 (1974). Cited on pages 16, 19, 20, 26, and 50.

[142] P. B. Mark, “The Sequoia Computer: A Fault-Tolerant Tightly-Coupled Multiprocessor Architecture”, *Proc. 12th International Symposium on Computer Architecture (ISCA)*, p. 232 (June, 1985). Cited on page 18.

[143] Evangelos Markatos, Mark Crovella, Prakash Das, Czarek Dubnicki, and Thomas LeBlanc, “The Effects of Multiprogramming on Barrier Synchronization”, Technical Report 380, University of Rochester (May, 1991). Cited on page 25.

[144] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos, “First-Class User-Level Threads”, *Proc. 13th ACM Symposium on Operating Systems Principles (SOSP)* (October, 1991). Cited on pages 20, 25, and 35.

[145] N. Matelan, “The FLEX/32 Multicomputer”, *Proc. 12th International Symposium on Computer Architecture (ISCA)*, pp. 209−213 (June, 1985). Cited on page 18.

[146] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry, “A Fast File System for UNIX”, *ACM Transactions on Computer Systems (TOCS)* **2** (3), pp. 181−197 (August, 1984). Cited on page 263.

[147] John M. Mellor-Crummey and Michael L. Scott, “Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors”, *ACM Transactions on Computer Systems (TOCS)* **9** (1), pp. 21−65 (February, 1991). Cited on pages 22 and 24.

[148] John M. Mellor-Crummey and Michael L. Scott, “Scalable Reader-Writer Synchronization for Shared-Memory Multiprocessors”, *Proc. 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pp. 106−113 (April, 1991). Cited on page 24.

[149] John M. Mellor-Crummey and Michael L. Scott, “Synchronization Without Contention”, *Proc. 4th International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 269−278 (April, 1991). Cited on pages 22 and 24.

[150] Robert M. Metcalfe and David R. Boggs, “Ethernet: Distributed Packet Switching for Local Computer Networks”, *CACM* **19** (7), pp. 395–403 (July, 1976). Cited on page 53.

[151] MIPS Computer Systems, Inc., *MIPS R4000 User’s Manual* (1991). Cited on page 21.

[152] Bram Moolenaar, “VIM − Vi Improved, A Programmers Text Editor”, UNIX man page for version 3.0 (August, 1994). Cited on page vi.

[153] Mortice Kern Systems Inc., *MKS Toolkit Reference Manual* (1989). Cited on page vi.

[154] David Mosberger, “Memory Consistency Models”, *Operating Systems Review* **27** (1), pp. 18−26 (January, 1993). Relevant correspondence appears in **27** (3); a revised version is available as Technical Report 93/11, University of Arizona, 1993. Cited on pages 11 and 212.

[155] Sape J. Mullender, Guido van Rossum, Andrew S. Tanenbaum, Robbert van Renesse, and Hans van Staveren, “Amoeba: A Distributed Operating System for the 1990s”, *IEEE Computer* **23** (5), pp. 44−53 (May, 1990). Cited on page 18.

[156] New York University, “NYU Ultracomputer UNIX Programmer’s Manual”, Ultracomputer Documentation Note #1, Ultracomputer Research Laboratory (March, 1988). Cited on page 122.

[157] Robert Olson, “Parallel Processing in a Message-Based Operating System”, *IEEE Software* **2** (4) (July, 1985). Cited on page 18.

[158] Elliot I. Organick, *The Multics System: An Examination of its Structure*, MIT Press, Cambridge, Massachusetts (1972). Cited on page 18.

[159] Elliott I. Organick, *Computer System Organization: the B5700/B6700 Series*, Academic Press, New York (1973). Cited on pages 16 and 26.

[160] John K. Ousterhout, “Scheduling Techniques for Concurrent Systems”, *Proc. 3rd International Conference on Distributed Computing Systems*, pp. 22–30 (1982). Cited on pages 20, 25, 40, 189, 273, 278, and 354.

[161] John K. Ousterhout, Donald A. Scelza, and Pradeep S. Sindhu, “Medusa: An Experiment in Distributed Operating System Structure”, *Commun. ACM* **23** (2), pp. 92–104 (February, 1980). Cited on page 29.

[162] G. Pfister, W. Brantley, D. George, S. Harvey, W. Kleinfelder, K. McAuliffe, E. Melton, V. Norton, and J. Weiss, “The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture”, *Proc. 1985 International Conference on Parallel Processing*, pp. 764–771 (August, 1985). Cited on pages v, 5, 33, and 271.

[163] Gregory F. Pfister and V. Alan Norton, “‘Hot Spot’ Contention and Combining in Multistage Interconnection Networks”, *IEEE Transactions on Computers* **34** (10), pp. 943–948 (October, 1985). Cited on page 24.

[164] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey, “Plan 9 from Bell Labs”, *Proc. UKUUG Conference*, pp. 1–9 (July, 1990). Cited on page 19.

[165] R. Rashid, A. Tevanian, Jr., M. Young, D. Golub, R. Baron, D. Black, W. J. Bolosky, and J. Chew, “Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures”, *IEEE Transactions on Computers* **37** (8) (August, 1988). Cited on page 33.

[166] R. F. Rashid and G. Robertson, “Accent: A Communication Oriented Network Operating System Kernel”, *Proc. 8th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 164–175 (December, 1981). Cited on pages 19 and 33.

[167] Richard F. Rashid, “Designs for Parallel Architectures”, *UNIX Review* (April, 1987). Cited on page 15.

[168] S. Reinhardt, “A Data-Flow Approach to Multitasking on CRAY X-MP Computers”, *Proc. 10th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 107–114 (December, 1985). Cited on page 18.

[169] D. M. Ritchie, “The UNIX System: The Evolution of the UNIX Time-sharing System”, *AT&T Bell Laboratories Tech. J.* **63** (8), pp. 1577–1594 (October, 1984). Cited on page 259.

[170] Dennis M. Ritchie and Ken Thompson, “The UNIX Time-Sharing System”, *Communications of the ACM* **17** (7), pp. 365–375 (July, 1974). Cited on page 251.

[171] James Rothnie, “Overview of the KSR-1 Computer System”, Kendall Square Research Report TR 9202001 (March, 1992). Cited on page 16.

[172] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser, “CHORUS Distributed Operating Systems”, pp. 305−370 in *Computing Systems*, USENIX Association (Fall 1988). Cited on page 19.

[173] Larry Rudolph and Zary Segall, “Dynamic Decentralized Cache Schemes for MIMD Parallel Processors”, *Proc. 11th International Symposium on Computer Architecture (ISCA)*, pp. 340−347 (June, 1984). Cited on page 21.

[174] Lawrence Rudolph, “Software Structures for Ultraparallel Computing”, Ph.D. Thesis, Courant Institute, NYU (February, 1982). Cited on pages 4, 22, 23, 51, 77, 85, and 302.

[175] Jack Schwartz, “A Taxonomic Table of Parallel Computers, Based on 55 Designs”, NYU Ultracomputer Note #69, New York University (1983). Cited on page 15.

[176] Jacob T. Schwartz, “Ultracomputers”, *ACM Transactions on Programming Languages and Systems (TOPLAS)* **2** (4), pp. 484−521 (October, 1980). Cited on page 23.

[177] Michael L. Scott, Thomas J. LeBlanc, and Brian D. Marsh, “Evolution of an Operating System for Large-Scale Shared-Memory Multiprocessors”, Technical Report 309, University of Rochester (March, 1989). Cited on page 35.

[178] Michael L. Scott, Thomas J. LeBlanc, and Brian D. Marsh, “Multi-Model Parallel Programming in Psyche”, *Proc. 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)* (March, 1990). Cited on page 35.

[179] Sequent Computer Systems, “Balance 8000 Guide to Parallel Programming”, Document MAN-1000-00, Sequent Computer Systems, Portland Oregon (July, 1985). Cited on page 32.

[180] Abraham Silberschatz, James L. Peterson, and Peter B. Galvin, *Operating System Concepts*, Addison-Wesley (1991). Cited on page 232.

[181] Brian V. Smith and others, *XFIG − Facility for Interactive Generation of Figures under X11, Release 2.1.3* (1992). Cited on page vi.

[182] Burton J. Smith, “Architecture and Applications of the HEP Multiprocessor Computer System”, *Real Time Signal Processing IV, Proceedings of SPIE*, pp. 241−248 (1981). Reprinted in *Supercomputers: Design and Applications*, ed. Kai Hwang, IEEE Computer Society Press, 1984. Cited on pages 6 and 22.

[183] Norris Parker Smith, “Wallach: Convex Cured Mach Woes, is Moving Onward”, *HPCwire* (April 7, 1995). Cited on page 20.

[184] Marc Snir, "On Parallel Search", *Proc. Principles of Distributed Computing Conference (PODC)*, pp. 242−253 (August, 1982). Cited on page 23.

[185] Mark S. Squillante and Edward D. Lazowska, "Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling", University of Washington Technical Report UW-CSE-89-06-01 (June, 1989). There is also an unpublished revision dated February, 1990. Cited on page 25.

[186] W. Y. Stevens, "The Structure of System/360 − Part 2: System Implementations", pp. 711−715 in *Computer Structures: Principles and Examples*, ed. Siewiorek, Bell, and Newell, McGraw-Hill, New York (1982). Cited on page 17.

[187] Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley (1986). Cited on page 11.

[188] R. J. Swan, A. Bechtolsheim, K. Lai, and J. Ousterhout, "The Implementation of the Cm* Multi-Microprocessor", *Proc. NCC* **46**, pp. 645−655, AFIPS Press (1977). Cited on page 29.

[189] R. J. Swan, S. H. Fuller, and D. P. Siewiorek, "Cm* − A Modular, Multi-Microprocessor", *Proc. NCC* **46**, pp. 637−644, AFIPS Press (1977). Cited on page 29.

[190] Andrew S. Tanenbaum, *Operating Systems: Design and Implementation*, Prentice-Hall (1987). Cited on page 232.

[191] Patricia J. Teller, "Translation-Lookaside Buffer Consistency in Highly-Parallel Shared-Memory Multiprocessors", Ph.D. Thesis, Courant Institute, NYU (1991). Cited on pages 213 and 235.

[192] T. J. Teorey, "Properties of Disk Scheduling Policies in Multiprogrammed Computer Systems", *Proc. AFIPS Fall Joint Computer Conference (FJCC)* **41**, pp. 1−11 (1972). Cited on page 68.

[193] Avadis Tevanian, Jr., Richard F. Rashid, David B. Golub, David L. Black, Eric Cooper, and Michael W. Young, "MACH Threads and the UNIX Kernel: The Battle for Control", *Proc. USENIX Conference* (June, 1987). Cited on page 19.

[194] Charles P. Thacker, Lawrence C. Stewart, and Edwin H. Satterthwaite, Jr., "Firefly: A Multiprocessor Workstation", *IEEE Transactions on Computers* **37** (8), pp. 909−920 (August, 1988). Cited on page 34.

[195] Tim Theisen and others, *Ghostview Version 1.1 − An X11 User Interface for Ghostscript* (1992). Cited on page vi.

[196] K. Thompson, "UNIX Time-Sharing System: UNIX Implementation", *Bell Sys. Tech. J.* **57** (6), pp. 1931−1946 (1978). Cited on pages 252 and 263.

[197] James E. Thornton, "Parallel Operation in the Control Data 6600", pp. 730–742 in *Computer Structures: Principles and Examples*, ed. Siewiorek, Bell, and Newell, McGraw-Hill, New York (1982). Cited on page 17.

[198] Andrew Tucker and Anoop Gupta, "Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors", *Proc. 12th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 159–166 (December, 1989). Cited on page 25.

[199] S. G. Tucker, "The IBM 3090 System: An Overview", *IBM Sys. J.* **25** (1), pp. 4–19 (1986). Cited on page 28.

[200] Roeland van Krieken, "Pentium Chip's Dual Personality", *Byte* **19** (12), pp. 211–212 (December, 1994). Cited on pages 45, 187, and 274.

[201] Mark Weiser, Alan Demers, and Carl Hauser, "The Portable Common Runtime Approach to Interoperability", *Proc. 12th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 114–122 (December, 1989). Cited on page 20.

[202] Matt Welsh, *Linux Installation and Getting Started*, Morse Telecommunicatons, Long Beach, NY (October 24, 1994). Cited on page vi.

[203] James Wilson, "A Buddy System Based On Parallel Queue Primitives", Ultracomputer System Software Note #40, New York University (May, 1983). Cited on page 4.

[204] James Wilson, "Supporting Concurrent Operations On A Parallel Queue", Ultracomputer System Software Note #41, New York University (January, 1984). Cited on page 4.

[205] James Wilson, "Operating System Data Structures for Shared-Memory MIMD Machines with Fetch-and-Add", Ph.D. Thesis, Courant Institute, NYU (June, 1988). Cited on pages 23, 80, 82, 85, 168, 170, 277, 278, and 340.

[206] R. W. Wisniewski, L. I. Kontothanassis, and M. L. Scott, "Scalable Spin Locks for Multiprogrammed Systems", *Proc. 8th International Parallel Processing Symposium* (April, 1994). Cited on page 22.

[207] David Wood, "Parallel Queues and Pools, An Evaluation", M.S. Thesis, Courant Institute, NYU (January, 1989). Revised version available as NYU Ultracomputer Note #150. Cited on pages 81, 82, 123, 127, 141, 310, 353, and 354.

[208] W. A. Wulf and C. G. Bell, "C.mmp: A Multi-Miniprocessor", *Proc. AFIPS Fall Joint Computer Conference (FJCC)* (December, 1972). Cited on page 28.

[209] W. A. Wulf, E. Cohe, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack, "HYDRA: The Kernel of a Multiprocessor Operating System", *Commun. ACM* **17** (6), pp. 337–345 (June, 1974). Cited on page 28.

[210] W. A. Wulf, R. Levin, and S. P. Harbison, *HYDRA: An Experimental Operating System*, McGraw-Hill, New York (1980). Cited on page 28.

[211] W. A. Wulf, R. Levin, and C. Pierson, "Overview of the HYDRA Operating System", *Proc. 5th ACM Symposium on Operating Systems Principles (SOSP)* (November, 1975). Cited on page 28.

[212] X/Open Company, Ltd., *Go Solo−How to Implement and Utilize the Single UNIX Specification*, X/Open and Prentice Hall PTR (1995). Cited on page 17.

[213] Pen-Chung Yew, Nian-Feng Tzeng, and Duncan H. Lawrie, "Distributing Hot-Spot Addressing in Large-Scale Multiprocessors", *IEEE Transactions on Computers* **C-36** (4), pp. 388−395 (April, 1987). Cited on page 24.

[214] Michael Young, Avadis Tavanian, Jr., Richard Rashid, David Golub, Jeffrey Eppinger, Jonathan Chew, William Bolosky, David Black, and Robert Baron, "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System", *Proc. 11th ACM Symposium on Operating Systems Principles (SOSP)* (November, 1987). Cited on page 19.

[215] John Zahorjan, Edward D. Lazowska, and Derek L. Eager, "Spinning Versus Blocking in Parallel Systems with Uncertainty", Technical Report 88-03-01, Dept. of Computer Science, University of Washington (March, 1988). Cited on pages 25 and 188.

[216] Roman Zajcew, Paul Roy, David Black, Chris Peak, Paulo Guedes, Bradford Kemp, John LoVerso, Michael Leibensperger, Michael Barnett, Faramarz Rabii, and Durriya Netterwala, "An OSF/1 UNIX for Massively Parallel Multicomputers", *Proc. USENIX Conference*, pp. 449−468 (Winter, 1993). Cited on page 19.

Index

N

O

S

Z