First-Class User-Level Threads

Brian D. Marsh Michael L. Scott
Thomas J. LeBlanc Evangelos P. Markatos

Computer Science Department
University of Rochester
Rochester, NY 14627–0226
{marsh,scott,leblanc,markatos}@cs.rochester.edu

Abstract

It is often desirable, for reasons of clarity, portability, and efficiency, to write parallel programs in which the number of processes is independent of the number of available processors. Several modern operating systems support more than one process in an address space, but the overhead of creating and synchronizing kernel processes can be high. Many runtime environments implement lightweight processes (threads) in user space, but this approach usually results in second-class status for threads, making it difficult or impossible to perform scheduling operations at appropriate times (e.g. when the current thread blocks in the kernel). In addition, a lack of common assumptions may also make it difficult for parallel programs or library routines that use dissimilar thread packages to communicate with each other, or to synchronize access to shared data.

We describe a set of kernel mechanisms and conventions designed to accord *first-class status* to user-level threads, allowing them to be used in any reasonable way that traditional kernel-provided processes can be used, while leaving the details of their implementation to user-level code. The key features of our approach are (1) shared memory for asynchronous communication between the kernel and the user, (2) software interrupts for events that might require action on the part of a user-level scheduler, and (3) a scheduler interface convention that facilitates interactions in user space between dissimilar kinds of threads. We have incorporated these mechanisms in the Psyche parallel operating system, and have used them to implement several different kinds of user-level threads. We argue for our approach in terms of both flexibility and performance.

This work was supported in part by NSF grant number CCR-9005633, NSF IIP grant number CDA-8822724, and a DARPA/NASA Graduate Research Assistantship in Parallel Processing.

1. Introduction

It is often desirable, for reasons of clarity, portability, and efficiency, to write parallel programs in which the number of user processes (threads) is independent of the number of available processors. Processes provided by the kernel can be used to represent user-level threads, but this approach has two important disadvantages:

Semantic inflexibility. Users want, and different runtime environments define, threads of various kinds, many of which may be incompatible with the kernel's notion of process. Some environments expect one thread to run at a time, as in the coroutine-like scheduling of Distributed Processes [7] and Lynx [22]. Some want to build cactus stacks, with dynamic allocation of activation records, as in Mesa [12]. Some want to use customized scheduling policies, such as priority-based scheduling, within an application. Some want to have processes interact using communication or synchronization mechanisms that are difficult to implement with kernel-provided operations. Some simply want to create a very large number of threads, more than the kernel can support.

Poor performance. Processes are employed in parallel programs for the sake of performance, not just conceptual clarity, so the overhead of kernel process management is important. Kernel-provided operations will almost always be slower than user-provided operations, partly because of the overhead of switching into and out of supervisor mode, but also because of the overhead of features required by some, but not all, applications. A thread may or may not need its own address space, priority, private file descriptors, or signal interface. It may or may not need to save floating point registers on context switches. Features such as these, provided by the kernel but often unused in user space, can incur unwarranted costs.

To overcome these problems, it has become commonplace to construct lightweight thread packages in user space [4, 9, 23, 27]. These packages multiplex a potentially large number of user-defined threads on top of a single kernel-implemented process. Within limits, they also allow the user to implement customized processes, communication, and scheduling inside an application.

User-level thread packages avoid kernel overhead on thread operations and satisfy our need for flexibility, but they also introduce new problems:

Blocking system calls. User-level thread packages require that the kernel provide a full set of non-blocking system calls. Otherwise, a system call performed by a single user-level thread will prevent the execution of other runnable threads. Many kernel interfaces include non-blocking implementations of some important system calls (I/O in particular), but most provide a large number of blocking calls as well.[1]

Lack of coordination between scheduling and synchronization. Synchronization between threads, either in the same address space or in overlapping address spaces, may be adversely affected by kernel scheduling decisions. A thread that is preempted by the kernel may be performing operations for which other, running threads must wait. In the simplest case, a preempted thread may hold a mutual exclusion lock, forcing other threads to spin (wasting cycles) or block (denying the application its fair share of available CPU resources). More generally, a preempted thread may be performing a computation for which other threads are waiting or will wait in the future, thereby slowing execution.

Lack of conventions for sharing among thread packages. When using user-level thread packages, a program may need to synchronize access to data shared among more than one kind of thread. This claim is one of the premises behind *multi-model parallel programming*[21], the simultaneous use of more than one model of parallelism, both in different applications and in different pieces of a single application. Spin locks are an easily implemented solution, but are not always appropriate. Blocking synchronization (e.g. semaphores) requires a mechanism that allows one kind of thread to invoke the scheduling operations of a different kind of thread. If true data abstractions (with code) are to be shared among thread packages, it must be possible for a single body of code to invoke appropriate operations for any relevant kind of thread.

All of these new problems arise because user-level threads are not recognized or supported by the kernel. Our goal is to grant *first-class status* to user-level threads, allowing them to be used in any reasonable way that traditional kernel-provided processes can be used, while leaving the details of their implementation to user-level code. For example, first-class status requires that threads be able to execute I/O and other blocking operations without denying service to their peers, and that different kinds of threads, in separate but overlapping address spaces, be able to synchronize access to shared data structures. This definition of first-class status is of necessity informal: we do not have an exhaustive list of required characteristics for threads. Rather, we have attempted to provide user-level code with the same sort of timely information and scheduling options normally available to the kernel, with the expectation that most, if not all, of the operations reasonable in the kernel will then become reasonable in user space.

In the next section we present the rationale for our approach and a brief overview of the specific mechanisms we propose. We describe our mechanisms in more detail in section 3. We discuss how our mechanisms support the construction of first-class user-level threads in section 4. We discuss related work in section 5. The implementation of our mechanisms, while not of production quality, nonetheless supports some useful performance experiments; we report on these in section 6. Our conclusions are presented in section 7.

[1] Edler et al. [10] note that blocking system calls in Unix include not only *read*, *write*, *open*, and *close*, but also *ioctl*, *mkdir*, *rmdir*, *rename*, *link*, *unlink*, *stat*, and others.

2. Rationale

Our approach was developed as part of the Psyche parallel operating system [19-21], running on the BBN Butterfly Plus multiprocessor [3]. The design of our thread mechanisms was heavily influenced by the need to support multi-model parallel programming, the primary goal of Psyche. In particular, we have attempted to ensure that kernel assumptions about the nature of threads are minimized. For example, we do not assume that threads will have contiguous, array-based stacks. Similarly, we do not assume that every address space will be multi-threaded; we want to allow an address space to contain a single kernel-level process, with no thread package above it, and yet still be able to interact via shared memory and user-level synchronization mechanisms with other address spaces.

The design of our mechanisms was also influenced by the need to provide acceptable performance on a NUMA (NonUniform Memory Access) multiprocessor such as the Butterfly. In particular, the high cost of moving thread state from one NUMA processor to another has motivated us to avoid migration whenever possible.

We designed the Psyche kernel interface to be used primarily by user-level thread packages, rather than application programmers. We assume a user-level thread package will create and maintain state for threads, and that the bulk of short-term scheduling occurs in user space. The kernel remains in charge of coarse-grain resource allocation and protection.

By maintaining thread state in user space, we can satisfy our goals of flexibility and performance. A user-level thread package can organize thread state any way it likes. Most thread operations, including creation, destruction, synchronization, and context switching, can occur in user space, without entering the kernel.[2] Kernel intervention is required only for protected operations (e.g. system calls) and coarse-grain resource allocation (e.g. preemptive scheduling).

When the kernel does intervene during execution, the kernel and user-level thread package need to cooperate, so that each has access to information maintained by the other. For example, when the kernel performs a blocking system call, it needs to identify the user thread making the call, so that a subsequent response can be directed to the appropriate thread. When a scheduling event is detected by the kernel (e.g. timer expiration, preemption), it needs to interrupt execution and notify the thread package of the scheduling event. When the execution of a thread is interrupted by the kernel, the state of the thread must be saved

[2] Some architectural features used to implement context switching, such as register windows on the SPARC, may require the use of a privileged instruction. Most architectures can implement thread operations without a kernel trap however, and even in the case of the SPARC, the required system call does not suffer from excessive generality.

in a location accessible to the thread package. When data abstractions are shared across address spaces, operations that must synchronize need access to information about the scheduling of different kinds of threads.

These general observations suggest that the kernel have access to thread state information maintained by the thread package, that the thread package accept scheduling interrupts from the kernel, and that thread package schedulers provide a standard interface. Specifically, in our solution:

(1) The kernel and the thread package share important data structures. Kernel/user shared data makes it easy to convey information efficiently (in both directions) when synchronous communication is not required. Read-only access to kernel-managed data is one obvious example: no system call is needed to determine the current processor number or process id. In the opposite direction, user-writable data can be used to specify what ought to happen in response to kernel-detected events, such as timer expiration. This mechanism allows changes in desired behavior to occur frequently, for example when switching between threads in user space. By allowing the kernel and user to communicate efficiently, we allow them to communicate more frequently.

(2) The kernel provides the thread package with software interrupts (signals, upcalls) whenever a scheduling decision may be required. Examples include timer expiration, imminent preemption, and the commencement and completion of blocking system calls. Timer interrupts support the time-slicing of threads. Warnings prior to preemption allow the thread package to coordinate synchronization with kernel-level scheduling. An interrupt delivered each time a thread blocks in the kernel makes every system call non-blocking by default, without modifying or replacing the kernel interface, and provides a uniform entry mechanism into the user-level scheduler when a thread has blocked or unblocked. Single-threaded applications can disable selected interrupts in order to have blocking calls.

(3) The operating system establishes a standard interface for user-level schedulers, and provides locations in which to list the functions that implement the interface. Abstractions shared between thread packages can then invoke appropriate operations to block and unblock different kinds of threads. Although the kernel never calls these operations itself, it identifies them in the kernel/user data area so that user-level code can invoke them without depending on the referencing environment of any particular programming language or thread package.

We now describe these mechanisms and their use in more detail.

3. Mechanisms

In Psyche, kernel processes are used to implement the *virtual processors* that execute user-level threads. In many respects our notion of virtual processor resembles the kernel-implemented threads of multiprocessor operating systems such as Mach [1] and Topaz (Taos) [24]. Virtual processors are created in response to a system call, very much like traditional kernel-implemented processes. To obtain true parallelism within an application, one creates a virtual processor (often in the same address space) on each of several different physical processors. It is possible to create more than one virtual processor in the same address space on the same processor, though parallel programs seldom do so. On each node of the physical machine, the kernel time-slices between virtual processors residing on that node.

3.1. Shared Kernel/User Data Structures

Figure 1 presents the kernel/user data structures used for scheduling in Psyche. These structures are rooted in a set of pseudo-registers on every physical processor, mapped read-only into every user address space at a static, fixed address. The pseudo-registers contain the identity of the physical processor, and pointers to the currently executing virtual processor and the currently active address space. The kernel changes these indicators as appropriate when context switching between virtual processors.

The address space and virtual processor pointers refer to data structures writable by the user. The address space data structure lies at the beginning of the data in the address space, and is created by the kernel along with the address space. This data structure contains the software interrupt vectors used by all virtual processors in the address space, together with additional information not relevant to scheduling. The kernel defines a default action for each type of interrupt, which it performs if the appropriate vector is null.

The virtual processor data structure contains the bulk of the information required to coordinate kernel- and user-level process management. It resides in a location specified by the user when the virtual processor is created. Among its contents are (1) a pointer to a stack on which to deliver software interrupts, (2) a collection of flags and values controlling the behavior of interrupts, and (3) a pointer to a data structure representing the current thread. This last data structure identifies scheduler routines appropriate for the thread, and contains additional thread state (stack, saved registers, etc.). Software interrupts are discussed in more detail in section 3.2; scheduler interfaces are discussed in section 3.3.

The thread data structure also contains a thread identifier, which the kernel uses to create a link between a requested operation and the thread making the request. When a thread makes a blocking system call, the kernel records the thread identifier, and notifies the thread package. The identifier is used in the kernel to distinguish different calls, functionality provided implicitly by the kernel process in other systems. When the system call completes, the kernel once again notifies the thread package, passing the thread identifier as a parameter.

3.2. Software Interrupts

When it wishes to deliver a software interrupt, the Psyche kernel first checks the virtual processor data structure to see whether interrupts are currently disabled (masked). If not, it obtains the address of the appropriate user-level handler from the address space data structure and the address of the appropriate interrupt stack from the virtual processor data structure. It pushes the user's old program counter and stack pointer onto the stack (along with volatile registers), pushes any additional information needed to describe the interrupt, sets the interrupt masking flag in the virtual processor data structure, updates the

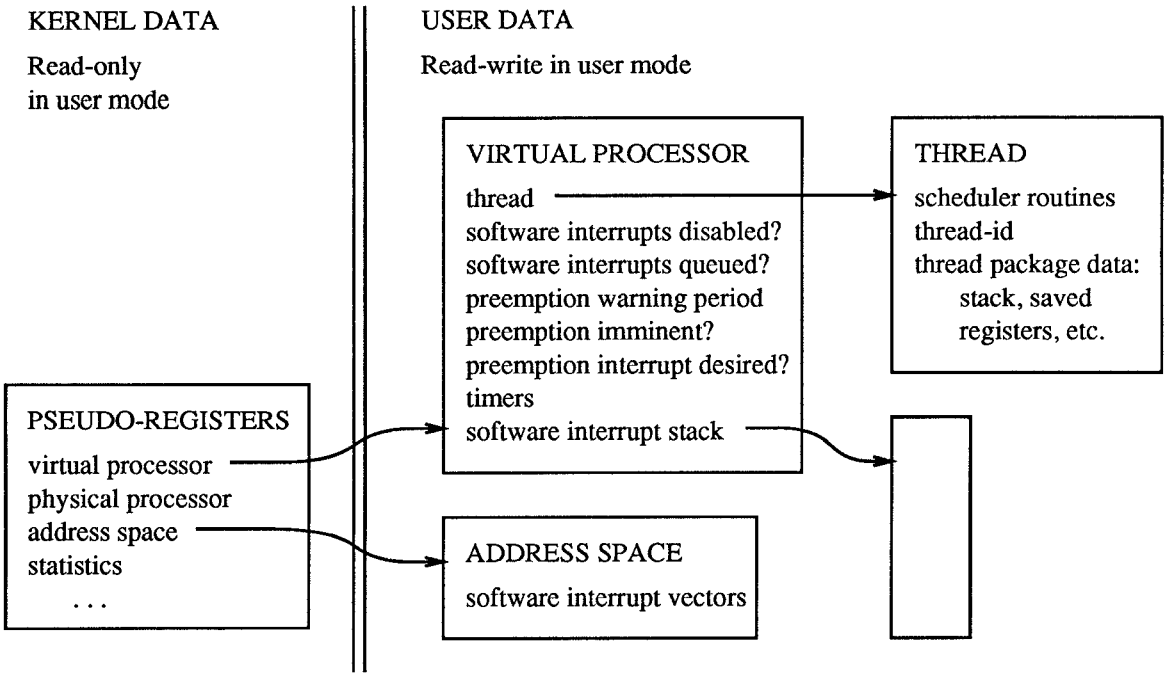

Figure 1: Shared Kernel/User Data Structures in Psyche

program counter and stack pointer, and enters user space. The user-level interrupt handler is then free to do as it pleases with the information it has been given. There is no return into the kernel from a software interrupt; the kernel retains no information about the interrupt after entering user space.

If the kernel wishes to deliver a software interrupt while interrupts are masked, it queues the interrupt instead, and sets a flag in the virtual processor data structure indicating that one or more interrupts are queued. The kernel sorts the queue based on the relative importance of interrupts. Program faults, for example, are queued ahead of timer expirations. We have not found it necessary to set the interrupt priorities in user space, but could do so by adding a priority field to the software interrupt vectors stored in the address space data structure.

As with hardware interrupts, the handlers for software interrupts should be designed to finish quickly. Once it has done everything necessary with the information on the interrupt stack (or has switched to a different stack), the typical handler re-enables interrupts and inspects the flag that indicates whether interrupts were queued. If it finds the bit set, it gives the kernel an opportunity to deliver a queued interrupt by executing a system call that blocks the virtual processor until the kernel has an interrupt to give it.

Software interrupts are always delivered by the local instance of the kernel, which runs in mutual exclusion with user code. If the kernel sees a flag indicating that interrupts are masked, it can assume that they will remain so until control re-enters user space. Likewise if user code sees a flag indicating that interrupts are queued, it can assume that they will remain so until the virtual processor executes a "block until interrupt" system call. The kernel delivers software interrupts only (1) at the moment they arise, provided they are not masked or queued, or (2) in response to this system call. These mechanisms suffice to avoid any race conditions between the kernel and user-level code. As in conventional device drivers, deadlock is avoided by masking software interrupts when acquiring a lock (or other resource) that might be required by an interrupt handler.

The following is a partial list of software interrupts in Psyche:

virtual processor initialization
thread blocked in the kernel
thread unblocked in the kernel
signal from another virtual processor
timer expiration
imminent preemption
program faults

Every virtual processor begins execution in the handler for the initialization interrupt. (Interrupt vectors are specified in the address space data structure, which must already exist when the virtual processor is created.) It also enters an interrupt handler in response to timer expiration, signals from other virtual processors, and various sorts of faults (divide by zero, protection violation, etc.). Slightly more complicated rules apply to blocking system calls and to virtual processor preemption.

When a system call must block for a large amount of time, the kernel delivers a software interrupt that allows the user-level thread package to run a different thread. When the operation completes, the kernel delivers a second interrupt that allows the thread package to reschedule the first thread. A single-threaded application can disable the scheduling hooks (thereby arranging for traditional blocking calls) by specifying null handlers for the interrupts associated with system calls.

For the sake of locality on our NUMA machine, virtual processors do not migrate among physical processors. Notification that a thread has unblocked in the kernel is delivered to the same virtual processor (running on the same physical processor) that received the earlier notification that the thread had blocked in the kernel. A user-level scheduler is free to move threads among the virtual processors in its address space (and some of our thread packages do so), but we did not want to build a migration assumption into the kernel interface. A well-written user-level scheduler on a NUMA machine will move threads only when it has to, or when their state is unusually small.

The principal blocking system call in Psyche is an RPC-like mechanism called the *protected procedure call* (PPC).[3] PPC requests are directed at an address space; the kernel chooses an idle or random virtual processor of the target address space and delivers an interrupt to it.[4] The kernel may or may not immediately deliver a "blocked in the kernel" interrupt to the virtual processor that was running the client thread. An interrupt always gets delivered when the requested operation is to execute on another processor, but it may be delayed when the operation can be executed locally, thereby allowing the server's virtual processor to execute instead. If the server finishes quickly (executing a "reply from PPC" system call within a single quantum), the kernel simply resumes the client. If the server does not finish quickly, the kernel delivers a "blocked in the kernel" interrupt to the client at the start of the following quantum, and an "unblocked in the kernel" interrupt when the server finally replies.

To minimize undesirable interactions between kernel-level scheduling and user-level synchronization, the kernel provides each virtual processor with a *two-minute warning* prior to preemption. User-level thread packages can set the actual duration of the warning (subject to a kernel-enforced maximum) by writing a value in the virtual processor data structure. They can also indicate whether it suffices for the kernel to set a warning flag, or whether an interrupt is required.

One use of the two-minute flag is to avoid acquiring a spin lock near the end of the virtual processor quantum. If the warning period exceeds the maximum length of a critical section, then a virtual processor will usually avoid preemption while holding a spin lock if it yields the processor voluntarily rather than acquire a lock after the warning bit is set. Software interrupts might consume a part of the "two-minute" period; a virtual processor can reduce the probability of inopportune preemption even further by masking software interrupts while holding any lock (not just those that might be required by a software interrupt handler).[5]

The two-minute warning flag provides an inexpensive way for user-level code to poll the kernel to determine if preemption is imminent. The two-minute warning interrupt, by contrast, allows a virtual processor to perform necessary clean-up actions when it is about to lose the processor. Such actions are likely to vary from one application and thread package to another. In an application performing loosely-synchronized heuristic search, the two-minute warning handler might flush the knowledge of the current thread to a globally-visible blackboard. A program that uses a central run queue for threads can use the two-minute warning to save the current thread in the queue. A program that uses separate run queues for each processor (the more likely alternative on a NUMA machine) can save the current thread in a global data structure that other virtual processors will examine if they run out of local work. If an about-to-be-preempted thread is working on an important computation that needs to be continued on another physical processor, the two-minute warning handler can save the state of the thread and then send an explicit interrupt to another virtual processor in the same address space, prompting it to migrate and run the preempted thread. The two-minute warning handler could even modify the state of any locks held by the current thread in such a way that other threads desiring to acquire the lock will block instead of spinning.

There is no guarantee that a fixed two-minute warning interval will always be sufficient to implement these actions. As a result, inopportune preemption is still possible, even with the two-minute warning. However, the goal of the two-minute warning is to minimize the likelihood of inopportune preemption, not to prevent it entirely. As long as the two-minute interval suffices most of the time, a periodic failure to deal with preemption adequately is unlikely to significantly affect performance.

3.3. Scheduler Interfaces

The thread data structure shared between the kernel and user identifies scheduler routines that can be used to block and unblock the current thread. (Additional fields can specify operations to create and destroy threads as well, but we do not require them.) The purpose of the scheduler interface is to facilitate the construction of data abstractions shared between dissimilar thread packages. The code and data of an atomic queue, for example, could be shared between thread packages A and B. If a thread in package A attempts to remove an item from the queue when it is empty, the dequeue operation can trace pointers from the pseudo-registers to find the scheduler routines of the currently-executing thread. It can place a pointer to the current thread, and to the unblock routine, in the data of the queue, and then call the block routine. When a thread in package B enqueues an item later, it will find the saved routine and call it, unblocking the thread from A. If the thread packages lie in a single address space, then all

[3] Protected procedure calls subsume I/O in Psyche. Requests for I/O are cast as PPCs to I/O server programs. The servers themselves use memory-mapped device registers and low-level (non-blocking) system calls to access physical devices.

[4] Since PPC requests are directed at an address space, we decided to associate all software interrupt vectors (containing the address of the interrupt handlers) with an address space, rather than with a virtual processor. In retrospect, this decision was a mistake, since in our implementation on a NUMA multiprocessor each interrupt vector must be translated into an address in local memory. In addition, if several different kinds of threads share an address space, interrupts must be redirected to a handler in the appropriate thread package.

[5] An application might be reluctant to yield the processor following a two-minute warning interrupt, preferring instead to make use of the rest of its quantum. If this concern proved to be a serious disincentive, or if fairness were of paramount importance, the kernel could add the unused portion of the current quantum to the beginning of the next.

of the scheduler operations will be invoked with ordinary procedure calls. If the thread packages lie in distinct but overlapping address spaces, then the unblock routine will be invoked via PPC. Further details can be found in [21].

3.4. Putting it All Together

By mirroring the behavior of a physical machine, with memory and interrupts, our approach provides the writers of thread packages with a familiar model of concurrent programming. System implementors are accustomed to using this model in operating systems, and in signal-based programs in Unix. Day-to-day programmers need never see the kernel interface; we assume that system calls will almost always be filtered through a thread-package library or language run-time system.

A typical thread package employs one virtual processor on each of several physical processors (see Figure 2). The virtual processors share a collection of scheduling routines and data, including the state of user-level threads. The pseudo-registers on each processor point to data structures describing the currently executing virtual processor and its address space. The thread field in the virtual processor data structure points into the data of the thread package. The software interrupt vectors in the address space data structure, and the scheduler operation list in the thread data structure, point into the scheduling routines of the thread package. Each virtual processor will execute scheduler routines at startup and in response to program faults, timers, and PPC requests. It will also execute scheduler routines when a system call blocks in the kernel, and when that call completes. By polling the two-minute warning flag or asking for two-minute interrupts, each virtual processor can arrange to execute scheduler code immediately prior to preemption and, by yielding explicitly, immediately after resumption. In an application whose level of available parallelism fluctuates dynamically, virtual processors can yield when they run out of work. Running virtual processors can re-awaken their peers with explicit signals when new work is created or arrives via PPC.

We have not found the implementation of thread packages on top of Psyche to be especially difficult. Three of the five packages available at present were ported from other systems. The first port took over a month, mainly because it uncovered kernel bugs, while the other two ports took less than a month each. All of the packages were integrated into a general system for cross-model synchronization and communication over the course of a two-month period [16].

4. Discussion

Returning to the issues enumerated in section 1, we now consider the degree to which our mechanisms support the construction of first-class user-level threads.

Semantic flexibility. In order to provide the implementors of user-level thread packages with as much flexibility as possible, we have attempted to minimize the assumptions embodied in the kernel. In particular, the kernel leaves space management (including the allocation of interrupt stacks) to user-level code, and most thread operations can be implemented entirely in user space. To ensure the integrity of scheduling within the thread package, the kernel provides software interrupts at every point where a scheduler action might be required. In our experiments with Psyche, we have successfully ported or implemented Multilisp futures [11], Uniform System tasks [25], Lynx threads [22], heavyweight single-threaded programs, and two different thread libraries.

Performance. As in all user-level thread packages, the ability to create, destroy, schedule, and synchronize threads without the assistance of the kernel keeps the cost of these operations low. Shared data structures allow the

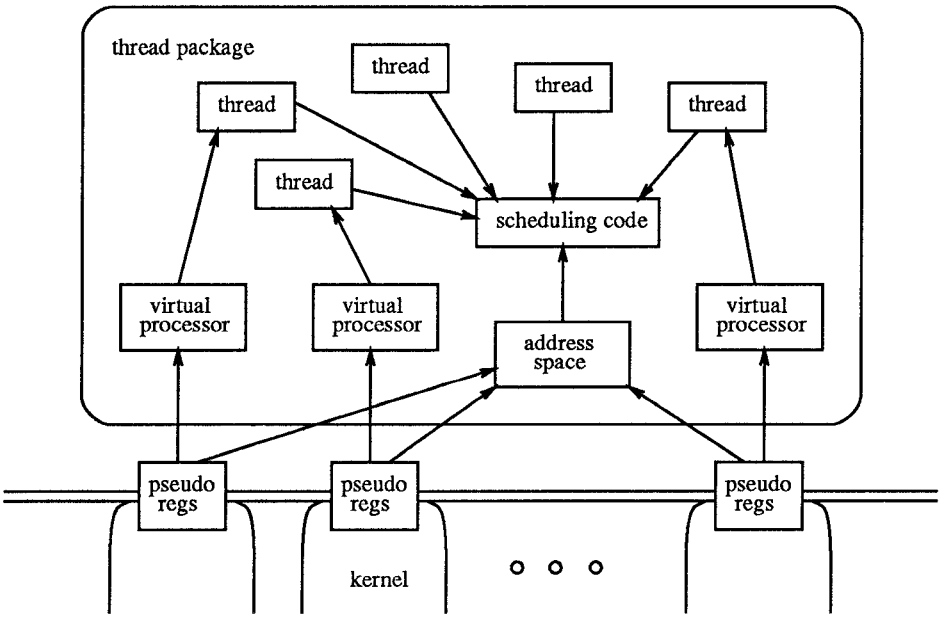

Figure 2: Typical Psyche Thread Package

kernel and user to provide each other with information efficiently and asynchronously. A virtual processor can change its interrupt stack, for example, simply by changing a pointer. In Psyche this facility allows a thread package to minimize the amount of parameter copying during a protected procedure call. Parameters arrive on the interrupt stack of a virtual processor; the interrupt handler reassigns the stack to the thread that is to perform the requested operation, and allocates a new stack for future interrupts.

Nonblocking System Calls. A general-purpose "blocked in the kernel" interrupt has the attractive property of providing hooks for user-level scheduling without requiring two different classes of system calls (blocking and non-blocking). Scheduler code, when needed, is triggered automatically via interrupts. Code that invokes system calls that usually return right away, but that may on occasion block, need not check for blocking. Library routines that may be invoked by both single- and multi-threaded applications need not worry about whether to use blocking or non-blocking calls: they can use any call they like, knowing that multi-threaded applications will handle scheduler interrupts when needed, and that single-threaded applications will disable them. The existence of "blocked in the kernel" interrupts also means that it is acceptable for user-level code to trigger kernel operations implicitly, via mechanisms other than system calls. In Psyche, for example, a bus error resulting from an attempt to call a subroutine at an invalid address can, under certain circumstances, be interpreted as a request for a PPC.

Coordination of scheduling and synchronization. The two-minute warning mechanism can be used to avoid undesirable interactions between user-level synchronization and kernel-level scheduling. It provides a virtual processor with time to pursue one of several courses of action just prior to preemption. It is useful even in a single-threaded address space, in order to synchronize access to data structures shared with other address spaces. The length of the two-minute warning, and even the decision as to whether or not to provide a two-minute warning, can be established dynamically.

Conventions for sharing among thread packages. By standardizing the interfaces to user-level schedulers, and by listing the entry points of the current scheduler in a well-known location, we allow user-level thread packages to synchronize access to shared data, and even to share synchronizing code. Threads in the same address space can invoke each other's scheduler operations with ordinary procedure calls. Threads in different address spaces can invoke them via PPC. Because they are listed in the standard kernel/user data structures, scheduler routines can be found by tracing pointers from a well-known, static address; no help is required from compilers, linkers, or run-time support routines.

There is nothing sacred about the layout or contents of kernel/user data structures in Psyche, or about the set of software interrupts (assuming it covers all interesting events). Additional data or interrupts might be required in a production-quality system. The key point is that software interrupts allow a user-level thread package to establish its own scheduling policies, and that kernel/user shared data not only permits fast asynchronous communication across the kernel interface, but also allows the user-level thread package to control the behavior of the software interrupt system.

5. Related Work

Several of the mechanisms we use to support first-class user-level threads have been used for other purposes in earlier systems. For example, our use of shared data between the kernel and user is not new; the *user structure* (u-dot) of Unix 4.3BSD, which is readable in user space, contains information about the current process, and is also used to implement machine-dependent mechanisms such as the "signal trampoline" [13]. None of the Unix user structure is shared between processes however, and none of it is writable in user mode, so it cannot be used to convey information from a thread package to the kernel.

As part of the Symunix project at New York University, Edler et al. [10] have proposed a set of parallel programming extensions to the Unix kernel interface, including support for user-level scheduling. In particular, they describe a new "meta-system-call" that provides an asynchronous wrapper for existing blocking calls, and a quantum-extending mechanism designed to avoid preemption during critical sections. The meta-system-call specifies which Unix system call to perform, and provides additional return arguments indicating whether the call completed immediately or will be announced by a later signal. The goal of the mechanism is to admit asynchronous system calls, while introducing as small a change to the kernel interface as possible, and maintaining compatibility with existing Unix programs.

The temporary non-preemption mechanism employs a counter in user space at a location known to the kernel. When entering a critical section, user-level code can increment the counter. Within reason, the kernel will refrain from preempting a process when the counter is non-zero. It subtracts any time spent beyond the normal end of quantum from the beginning of the process's next quantum. This mechanism suffices to avoid performance problems due to preemption during fine-grain critical sections. It may be cheaper than the two-minute warning, since it incurs the overhead of an extra clock interrupt only when the process is actually in a critical section at the normal end of the quantum. An ability to request temporary non-preemption at the end of the quantum does not suffice, however, for a program that requires asynchronous notification to trigger an explicit action.

At the University of Washington, Anderson et al. [2] have explored user-level scheduling in the context of the Topaz operating system on the DEC SRC Firefly multiprocessor workstation, an UMA machine. They rely on software interrupts from the kernel to provide user-level thread packages with appropriate scheduling hooks, but their mechanisms differ from ours in several respects, and they have used their mechanisms to address problems we have not considered, including page faults and upward-compatible simulation of traditional kernel threads.

For each address space, Anderson et al. maintain a pool of virtual processors (called "scheduler activations") in the kernel. When a scheduler activation is preempted or blocks in the kernel, the kernel freezes its state and sends a new activation from the pool up into user space. The new activation (and any other running activations in the same address space) can examine the state of the old activation. The kernel reclaims old activations (returning them to the pool) only when explicitly notified by the user-level scheduler that their state is no longer required.

Because they are running on an UMA machine, Anderson et al. can reasonably *recover* from preemption in critical sections, rather than avoid it. When an activation is preempted on one physical processor they immediately preempt an activation on a second processor, and send a new activation into user space on the second processor, passing it *both* of the old activations as arguments. This mechanism obviates the need to worry about how much work can be accomplished during our two-minute interval. It requires, however, that an address space be multi-threaded (and running on more than one processor at a time) if it wants to handle preemption interrupts. It also requires that an application move threads between processors if it wants to clean up the state of a preempted thread (e.g. by completing a critical section). Neither of these requirements is restrictive in the environment for which scheduler activations were designed, but both would be problematical for Psyche. In order to write multi-model parallel programs, we want to allow a single-threaded application component to receive preemption interrupts so that it can synchronize and share data with other application components. Since we are using a NUMA machine, we also want to be able to reach a clean point at preemption time without requiring thread migration.

All of the mechanisms discussed above allow user programs to control their scheduling behavior. Black [6] has proposed instead that programs provide the operating system with scheduler hints. He describes a set of system calls in Mach that allow a virtual processor (a Mach *thread)* to suggest to the kernel that it be de-scheduled, or that possession of the processor be handed off to some specified virtual processor. These calls can be used, for example, to yield control when spinning for a lock (in the hope that the holder of the lock will run instead) or to pass control to the holder by name, if known. As with scheduler activations, the issue of preemption in critical sections is handled through detection and recovery; the difference lies in delaying recovery until some other thread waits for the lock, and in letting that thread take responsibility for solving the problem. Once again, efficient recovery depends on cheap migration.

The various approaches to dealing with preemption are principally motivated by the unpredictable nature of kernel scheduling. There is less of a need for special mechanisms that deal with preemption however, if the machine is shared using physical partitions instead of time-slicing. Under processor partitioning [8, 18, 26] each application receives a set of dedicated processors for a relatively long period of time; preemption is only used to reallocate processors for medium-term scheduling. Processor partitioning may be the preferred scheduling policy for self-contained, compute-intensive parallel programs, particularly on machines with large numbers of processors. Effective processor partitioning may be difficult to implement however, if the boundaries between programs are poorly defined—as in large multi-model applications—or if there are only a small number of processors to allocate. Processor partitioning may also waste cycles if applications fail to balance their computation or I/O evenly among processes. In these cases, a mechanism that can be used to avoid inopportune preemption, such as the two-minute warning, is an attractive alternative.

User-level scheduling (and cooperation with the kernel scheduler) is only one aspect of multi-model parallel programming. The more general problem has been addressed in part by the Presto [4] and Agora [5] programming environments, at Washington and at CMU, respectively. Presto is an unusually flexible user-level thread package. It is constructed in an object-oriented style, with an internal structure into which users can plug a wide variety of process and communication abstractions. It suffers, however, from the standard problems of user-level thread packages on a traditional operating system: multi-model programs cannot span languages and address spaces, and performance can suffer from blocking in the kernel and preemption. Agora is a collection of libraries and interface compilers that allow users to connect distributed programs with stylized shared memory abstractions, much as an RPC stub generator allows users to connect those programs with a message-passing abstraction. Agora is built on top of Mach [1], and uses Mach's kernel-implemented threads.

6. Performance Implications

The two goals of our work are flexibility and performance for user-level threads. To achieve acceptable performance, features provided by a thread package must be cheap enough to use frequently. In this section we argue that a system providing kernel support for first-class user-level threads can perform better than either a system based on kernel-implemented processes or a system employing conventional user-level thread packages.

Our performance figures were derived from our implementation of Psyche on the BBN Butterfly Plus multiprocessor [3], which contains MC68020 processors clocked at 16 MHz. Our experiments quantify the performance advantages of first-class user-level threads over both kernel processes and conventional user-level threads.

6.1. Comparison to Kernel-Implemented Processes

It is widely recognized that kernel-implemented processes are inherently more expensive than user-implemented threads. The difference can be attributed not only to trap overhead, but also to the degree of functionality that must be designed into a process abstraction meant to meet the needs of disparate applications. The resulting difference in context switch time is often substantial. Weiser, Demers, and Hauser [27] report a context switch time of 77 µs for user-level threads in the Portable Common Runtime on a SPARC-based workstation. The context switch time for the thread package used in the Psyche experiments is 51 µs. Anderson [2] reports a time of 37 µs in his FastThreads package on the CVAX processor. Comparable times for kernel-implemented processes are at least an order of magnitude slower in each case: 550 µs in Psyche on the Butterfly, 441 µs in Topaz on the CVAX. Depending on the frequency of synchronization among threads, the impact of this disparity on application performance can be very large.

To assess the impact of context switch time on performance, we measured the running time of two applications, using two different implementations for each: one based on user-level threads and a user-level context switch mechanism, the other based on kernel processes and kernel scheduling. The first application performs Gaussian elimination on a 512×512 element matrix. The second sorts an array of 4608 elements. Each application was run using 16 physical processors and 128 threads of control.

In the first implementation, 128 kernel processes (virtual processors) were created and distributed evenly among the 16 physical processors. In the second implementation, 128 user-level threads were created and distributed evenly among 16 virtual processors, which were mapped one-to-one with physical processors. The results appear in the following table.

application	kernel-level virtual processors	user-level threads
Gaussian elimination	33.8 sec	22.0 sec
parallel sort	44.3 sec	27.1 sec

As can been seen from the table, performance improves by 35% when using user-level context switching in the Gaussian elimination program, and by 39% in the parallel sort. The improvement is large in both cases because of the frequency of context switching in these applications. The sort program has 4608 centralized barriers, each of which requires context switching among the 8 threads (or processes) on a physical processor. Since a context switch between kernel processes takes 550 μs, we can expect the sort program to spend almost 18 seconds during execution just to context switch between kernel processes. The Gaussian elimination program has only 512 barriers, but each is a tree barrier, which can introduce context switches for each level in the tree [15].

The performance advantage of user-level threads is substantial in these cases, and we would expect many parallel applications to produce comparable results on other systems. To show that these are not pathological examples, particularly with respect to the number of threads in use, we measured the speedup of the Gaussian elimination program on 16 processors as we varied the number of threads per processor from 1 to 8, thereby varying the amount of work performed by each thread. The results appear in Figure 3.[6] Threads in excess of the physical level of parallelism induce additional overhead no matter how they are implemented. With user-level threads, speedup on 16 processors degrades from 14.2 with one thread per processor to 11.1 with 8 threads per processor. The impact on application speedup is much more pronounced in the case of kernel threads, however: they degrade from a speedup of nearly 14.0 with one kernel process per processor to 7.3 with 8 kernel processes per processor.

6.2. Comparison to Conventional Thread Packages

Even on a uniprocessor the completion time of an application will suffer if user-level threads block in the kernel and thus deny service to their peers. Good thread packages therefore utilize non-blocking portions of the kernel interface whenever possible. Ignoring the possibility that threads on different processors might be waiting for one another, simply blocking the threads on *one* processor can have a serious impact on performance. A thread package that makes a blocking system call every 20 ms, with an expected service time of 5 ms, will be able to use no more than three quarters of the available CPU

[6] We believe the anomaly at 2 processors to be an artifact of the tree barrier implementation.

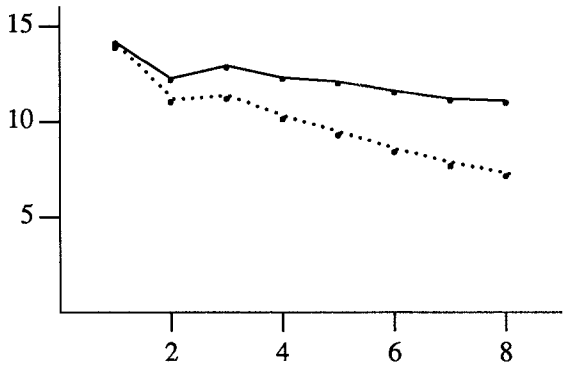

Figure 3: Speedup on 16 physical processors as a function of the number of kernel processes (dotted) or user threads (solid) per processor.

cycles, even if it always has runnable threads. Without requiring an explicitly non-blocking interface, our approach allows a thread package to regain control of the processor whenever any system call blocks.

If kernel-level scheduling is not coordinated with user-level synchronization, threads may also block for locks that are held by threads on preempted virtual processors, or for conditions that can be made true only by threads on preempted virtual processors. Zahorjan et al. [28] report performance degradations in the neighborhood of 25% when processes may be preempted at arbitrary times, while sharing a lock that is in use 75% of the time. Leutenneger [14] describes the variation in performance degradation as a function of lock utilization. For a lock that is in use 50% of the time, he reports that round-robin scheduling performs 10% worse than a processor allocation scheme in which the processes of a given application always run concurrently. For a lock that is in use 80% of the time, performance degradation increases to 57%.

Similar effects can occur in programs with condition synchronization. One of the most common models of parallel programming employs a collection of worker processes, one per processor, which repeatedly dequeue and execute tasks from a central work queue [11,25,26]. One of the things that a task may do is generate more tasks. It will often do so only if it is the last task of a certain kind to finish. Central work queue programs can thus be considered a generalization of barriers; parallel execution continues as long as the queue remains non-empty, and stops when no more tasks can be generated until some preempted task has completed.

Barrier programs can be expected to suffer from inopportune preemption more than do programs based on spin locks, because the probability is high that a process will be working on something critical (i.e. progress towards the barrier) at any given time. Tucker and Gupta [26] observe that the impact of preemption on work-queue based programs can be reduced by introducing a mechanism to preempt worker processes only after they finish a task and before another task is removed from the work queue. We have experimented with this technique in Psyche using a Uniform System program with the iterative structure characteristic of Gaussian elimination, the grassfire

algorithm, and the dynamic programming algorithm for transitive closure and all-pairs shortest path.

Our Uniform System program runs with one virtual processor worker on each of 16 physical processors. It proceeds through a series of 200 phases. At the beginning of each phase it generates 210 tasks and places them in the work queue. It does not generate more tasks until the existing ones have completed, thereby achieving an implicit barrier between phases. Virtual processors spin when they discover that all tasks have been removed from the queue, but not all have completed. Each task requires approximately 2.5 ms to complete. The quantum size is 100 ms, implying that 16 processors should be able to finish about 3 barriers (phases) per quantum.

We measured the completion time of our program under various levels of multiprogramming, with and without avoidance based on the two-minute warning flag. We varied the level of multiprogramming by placing unrelated virtual processors (executing in an infinite loop) on each of the physical processors. The two minute warning, when used, is set to 3 ms; workers check the flag prior to dequeuing a new task, and yield if it is set. Average completion times appear in the following table; individual runs varied by ± 0.2 seconds.

multiprogramming level		two-minute warning disabled	enabled
(no competitors)	1	8.50	8.58
(1 competitor)	2	40.1	16.0
	3	61.6	23.8

As can be seen in the table, the slowdown in execution matches the multiprogramming level closely when the two-minute warning is used. (We believe the slightly better than linear slowdown to be due to reduced contention.) Without the two-minute warning, preemption of a worker in the middle of a task causes all other workers to spin between the time that the work queue is exhausted and the next time the preempted worker gets to run, significantly increasing the execution time. By using the two-minute warning to avoid untimely preemption, we improve performance by a factor of two or more in the presence of multiprogramming. In the absence of multiprogramming, the two-minute warning imposed a performance penalty of less than one percent. This penalty stems from checking the flag prior to executing every task, and from yielding the processor explicitly, instead of being preempted.

In an earlier version of this same experiment, we observed little benefit from using the two-minute warning because we used an implementation of the Uniform System in which all tasks were created by a single task-generator process. Although the two-minute warning could still be used by worker processes to avoid taking a task off the work queue just prior to preemption, it could not be used to avoid preemption of the critical task generator. Even if the task generator is able to execute just prior to preemption, we would expect the work queue to be exhausted long before the task generator is again ready for execution, causing every worker process to spin.

One possible solution to this problem is to use the two-minute warning interrupt as a signal to save the state of the task-generator process and migrate it to another processor. Depending on the amount of state to be migrated, and the cost of migration on a given architecture, this option may or may not be viable every quantum. In our case, migration every quantum was not cost effective, and therefore the two-minute warning mechanism was not sufficient to solve the problem. We first had to decentralize task generation, allowing any process to generate tasks at the appropriate time. We then used the two-minute warning to ensure that a process did not begin task generation unless it could complete it.

7. Conclusions

In our attempts to provide support for multi-model parallel programming, we have encountered flexibility and performance problems with both conventional kernel-implemented processes and user-level thread packages. Kernel-implemented processes do not provide the variety of semantics required by parallel programs, and are too expensive to use for fine-grain operations. User-level thread packages suffer performance losses when threads block in the kernel or are preempted in critical sections. They also lack a mechanism for blocking and unblocking each other's threads, preventing the construction of multi-model programs.

To address these problems we employed three mechanisms which together accord first-class status to user-level threads:

- The kernel and the thread package communicate using shared memory whenever possible to avoid the need for synchronous interaction. Shared memory provides qualitative as well as quantitative benefits, because it makes it feasible to change the parameters of the kernel/user interface as often as every thread context switch. This capability is especially important when scheduling threads of more than one kind on a single virtual processor.
- The kernel provides the thread package with software interrupts whenever a scheduling decision may be required. In conjunction with shared flags, a full set of software interrupts allows a thread package to reacquire the processor when one of its threads blocks in the kernel, and to coordinate synchronization with kernel-level scheduling.
- The operating system establishes a standard interface to the scheduler routines (i.e. block and unblock) of user-level threads. Interface conventions allow data abstractions incorporating blocking synchronization to be shared by dissimilar thread packages, with scheduler operations invoked by shared code.

Support for first-class threads in Psyche has allowed us to construct a wide variety of user-level thread packages, and to employ them in the construction of multi-model programs. Our largest demonstration, now running in our robot lab, employs four different user-level process models in an integrated checkers-playing program [17]. A vision module, using a central queue of Uniform System tasks, analyzes video camera input to determine the most recent move of a human opponent on a conventional checkers set. A strategy module, using message passing between multi-threaded Lynx processes, performs parallel alpha-beta search to chose an appropriate counter-move. A motion-planning module, written in Multilisp, develops a plan to effect the move with a PUMA robot arm. The arm controller itself is a single-threaded C program. All four modules share an abstract representation of the state

of the board, and block and unblock each other as necessary to coordinate their work. Our experiences with this application and others suggest that first-class user-level threads provide both flexibility and performance, and offer advantages over both kernel-implemented processes and conventional user-level threads.

Acknowledgments

We would like to thank Tom Anderson, Brian Bershad, David Black, Jan Edler, Carla Ellis, Ed Lazowska, and Hank Levy for their help in improving this paper. We would also like to thank the many other researchers at Rochester who have contributed to the design and implementation of the Psyche kernel and its applications.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian and M. Young, "Mach: A New Kernel Foundation for UNIX Development," *Proceedings of the Summer 1986 USENIX Technical Conference*, June 1986, pp. 93-112.

[2] T. E. Anderson, B. N. Bershad, E. D. Lazowska and H. M. Levy, "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism," *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, October 1991.

[3] BBN Advanced Computers Incorporated, "Inside the Butterfly Plus," Cambridge, MA, October 1987.

[4] B. N. Bershad, E. D. Lazowska, H. M. Levy and D. B. Wagner, "An Open Environment for Building Parallel Programming Systems," *Proceedings of the First ACM Conference on Parallel Programming: Experience with Applications, Languages and Systems*, 19-21 July 1988, pp. 1-9.

[5] R. Bisiani and A. Forin, "Multilanguage Parallel Programming of Heterogeneous Machines," *IEEE Transactions on Computers 37*:8 (August 1988), pp. 930-945.

[6] D. L. Black, "Scheduling Support for Concurrency and Parallelism in the Mach Operating System," *Computer 23*:5 (May 1990), pp. 35-43.

[7] P. Brinch Hansen, "Distributed Processes: A Concurrent Programming Concept," *Communications of the ACM 21*:11 (November 1978), pp. 934-941.

[8] M. Crovella, P. Das, C. Dubnicki, T. J. LeBlanc and E. P. Markatos, "Multiprogramming on Multiprocessors," TR 385, Computer Science Department, University of Rochester, February 1991 (revised May 1991).

[9] T. W. Doeppner, Jr., "Threads: A System for the Support of Concurrent Programming," Technical Report CS-87-11, Department of Computer Science, Brown University, 1987.

[10] J. Edler, J. Lipkis and E. Schonberg, "Process Management for Highly Parallel UNIX Systems," Ultracomputer Note #136, Courant Institute, N. Y. U., April 1988.

[11] R. H. Halstead, Jr., "Multilisp: A Language for Concurrent Symbolic Computation," *ACM Transactions on Programming Languages and Systems 7*:4 (October 1985), pp. 501-538.

[12] B. W. Lampson and D. D. Redell, "Experience with Processes and Monitors in Mesa," *Communications of the ACM 23*:2 (February 1980), pp. 105-117.

[13] S. J. Leffler, M. K. McKusick, M. J. Karels and J. S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*, The Addison-Wesley Publishing Company, Reading, MA, 1989.

[14] S. Leutenegger, "Issues in Multiprogrammed Multiprocessor Scheduling," Ph. D. Thesis, TR 954, Department of Computer Sciences, University of Wisconsin — Madison, August 1990.

[15] E. Markatos, M. Crovella, P. Das, C. Dubnicki and T. J. LeBlanc, "The Effects of Multiprogramming on Barrier Synchronization," TR 380, Computer Science Department, University of Rochester, May 1991.

[16] B. D. Marsh, "Multi-Model Parallel Programming," Ph. D. Thesis, Computer Science Department, University of Rochester, July 1991.

[17] B. Marsh, C. Brown, T. LeBlanc, M. Scott, T. Becker, P. Das, J. Karlsson and C. Quiroz, "The Rochester Checkers Player: Multi-Model Parallel Programming for Animate Vision," TR 374, Computer Science Department, University of Rochester, June 1991.

[18] C. McCann, R. Vaswani and J. Zahorjan, "A Dynamic Processor Allocation Policy for Multiprogrammed Shared Memory Multiprocessors," TR 90-03-02, Department of Computer Science and Engineering, University of Washington, March 1990 (Revised February 1991).

[19] M. L. Scott, T. J. LeBlanc and B. D. Marsh, "Design Rationale for Psyche, a General-Purpose Multiprocessor Operating System," *Proceedings of the 1988 International Conference on Parallel Processing*, August 1988, pp. 255-262.

[20] M. L. Scott, T. J. LeBlanc and B. D. Marsh, "Evolution of an Operating System for Large-Scale Shared-Memory Multiprocessors," TR 309, Computer Science Department, University of Rochester, March 1989.

[21] M. L. Scott, T. J. LeBlanc and B. D. Marsh, "Multi-Model Parallel Programming in Psyche," *Proceedings of the Second ACM Symposium on Principles and Practice of Parallel Programming*, March 1990, pp. 70-78.

[22] M. L. Scott, "The Lynx Distributed Programming Language: Motivation, Design, and Experience," *Computer Languages 16*:3/4 (1991), pp. 209-233.

[23] Sun Microsystems, Inc., "Lightweight Processes," in *SunOS Programming Utilities and Libraries*, 27 March 1990. Sun Part Number 800-3847-10.

[24] C. P. Thacker and L. C. Stewart, "Firefly: A Multiprocessor Workstation," *IEEE Transactions on Computers 37*:8 (August 1988), pp. 909-920.

[25] R. H. Thomas and W. Crowther, "The Uniform System: An Approach to Runtime Support for Large Scale Shared Memory Parallel Processors," *Proceedings of the 1988 International Conference on Parallel Processing*, August 1988, pp. 245-254.

[26] A. Tucker and A. Gupta, "Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors," *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, 3-6 December 1989, pp. 159-166.

[27] M. Weiser, A. Demers and C. Hauser, "The Portable Common Runtime Approach to Interoperability," *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, 3-6 December 1989, pp. 114-122.

[28] J. Zahorjan, E. D. Lazowska and D. L. Eager, "The Effect of Scheduling Discipline on Spin Overhead in Shared Memory Parallel Systems," *IEEE Transactions on Parallel and Distributed Systems* 2:2 (April 1991), pp. 180-198.