Thread Management for Shared-Memory Multiprocessors

Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy

Department of Computer Science and Engineering
University of Washington
Seattle WA 98195

Abstract

Threads, or "lightweight processes," have become a common and necessary component of new languages and operating systems. Threads allow the programmer or compiler to express, create, and control parallel activities, contributing to the structure and performance of programs.

In this article, we discuss the many alternatives that present themselves when designing a support system for threads on a shared-memory multiprocessor. These alternatives influence the ease, granularity, and performance of parallel programming. We conclude with a brief survey of three contemporary thread management systems (Windows NT, Presto, and Multilisp), using them to illustrate the issues raised in this article.

Index Terms – thread, multiprocessor, operating system, parallel programming, performance

1 Introduction

Disciplined concurrent programming can improve the structure and performance of computer programs on both uniprocessor and multiprocessor systems. As a result, support for *threads*, or "lightweight processes," has become a common element of new operating systems and programming languages.

A thread is a sequential stream of instruction execution. A thread differs from the more traditional notion of a "heavyweight process" in that it separates the notion of execution from the other state needed to run a program (e.g., an address space). A single thread executes a portion of a program, while cooperating with other threads that are concurrently executing the same program. Much of what is normally kept on a per-heavyweight-process basis can be maintained in common for all threads in a single program, yielding dramatic reductions in the overhead and complexity of a concurrent program.

Concurrent programming has a long history. The operation of programs that must handle real-world concurrency (e.g., operating systems, database systems, and network file servers) can be complex and difficult to understand. Dijkstra [Dijkstra 68] and Hoare [Hoare 74, Hoare 78] showed that these programs can be simplified when structured as cooperating sequential threads that communicate at discrete points within the program. The basic idea is to represent a single task, such as fetching a particular file block, within a single thread of control, and to rely on the thread management system to multiplex concurrent activities onto the available processor. In this way, the programmer can consider each function being performed by the system separately, and simply rely on automatic scheduling mechanisms to best assign available processing power.

In the uniprocessor world, the principal motivations for concurrent programming have been improved program structure and performance. Multiprocessors offer an opportunity to use concurrency in parallel programs to improve performance, as well as structure. Moderately increasing a uniprocessor's power can require substantial additional design effort, as well as faster and more expensive hardware components. But, once a mechanism for interprocessor communication has been added to a uniprocessor design, the system's peak processing power can be increased by simply adding more processors. A shared-memory multiprocessor is one such design in which processors are connected by a bus to a common memory.

Multiprocessors lose their advantage if this processing power is not effectively utilized. If there are enough independent sequential jobs to keep all of the processors busy, then the potential of a multiprocessor can be easily realized: each job can be placed on a separate processor. However, if there are fewer jobs than processors, or if the goal is to execute single applications more quickly, then the machine's potential can only be achieved if individual programs can be parallelized in a cost-effective manner. Three factors contribute to the cost of using parallelism in a program:

Thread Overhead: The work, in terms of processor cycles, required to create and control a thread must be appreciably less than the work performed by that thread on behalf of the program. Otherwise, it is more efficient to do the work sequentially, rather than use a separate thread on another processor.

Communication Overhead: Again in terms of processor cycles, the cost of sharing information between threads must be less than the cost of simply computing the information in the context of each thread.

Programming Overhead: A less tangible metric than the previous two, programming overhead reflects the amount of human effort required to construct an efficient parallel program.

High overhead in any of these areas makes it hard to build efficient parallel programs. Costly threads can only be used infrequently. Similarly, if arranging communication between threads is slow, then the application must be structured so that little inter-thread communication is required. Finally, if managing parallelism is tedious or difficult, then the programmer may find it wise to sacrifice some speedup for a simpler implementation. Few algorithms parallelize well when constrained by high thread, communication, and programming costs, although many can flourish when these costs are low.

Low overhead in these three areas is the responsibility of the thread management system, which bridges the gap between the physical processors (the suppliers of parallelism) and an application (its consumer). In this article, we discuss the issues that arise in designing a thread management system to support low-overhead parallel programming for shared-memory multiprocessors. In the next section, we describe the functionality found in thread management systems. Section 3 discusses a number of thread design issues. In Section 4, we survey three systems for shared-memory multiprocessors, Windows NT [Custer 93], Presto [Bershad et al. 88], and Multilisp [Halstead 85], focusing our attention on how they have addressed the issues raised in this article.

2 Thread Management Concepts

2.1 Address Spaces, Threads, and Multiprocessing

An address space is the set of memory locations that can be generated and accessed directly by a program. Address space limitations are enforced in hardware to prevent incorrect or malicious programs in one address space from corrupting data structures in others. Threads provide concurrency within a program, while address spaces provide failure isolation between programs. These are orthogonal concepts, but the interaction between thread management and address space management defines the extent to which data sharing and multiprocessing are supported.

The simplest operating systems, generally those for older-style personal computers, support only a single thread and a single address space per machine. A single address space is simpler and faster since it allows all data in memory to be accessed uniformly. Separate address spaces are not needed on dedicated systems to protect against malicious users; software errors can crash the system but at least are localized to one user, one machine.

Even single-user systems can have concurrency, however. More sophisticated systems, such as Xerox's Pilot [Redell et al. 80], provide only one address space per machine, but support multiple threads within that single address space. Because any thread can access any memory location, Pilot provides a compiler with strong type-checking to decrease the likelihood that one thread will corrupt the data structures of another.

Other operating systems, such as UNIX, provide support for multiple address spaces per machine, but only one thread per address space. The combination of a UNIX address space with one thread is called a UNIX *process*; a process is used to execute a program. Since each process is restricted from accessing data that belongs to other processes, many different programs can run at the same time on one machine, with errors confined to the address space in which they occur. Processes are able to cooperate by sending messages back and forth via the operating system. Passing data through the operating system is slow, however; only parallel programs that require infrequent communication can be written using threads in disjoint address spaces.

Instead of using messages to share data, processes running on a shared-memory multiprocessor can communicate directly through the shared memory. Some UNIX systems allow memory regions to be set up as shared between processes; any data in the shared region can be accessed by more than one process without having to send a message by way of the operating system. The Sequent Symmetry's DYNIX [Seq 88] and Encore's UMAX [Enc 86] are operating systems that provide support for multiprocessing based on shared memory between UNIX processes.

More sophisticated operating systems for shared-memory multiprocessors, such as Microsoft's Windows NT and Carnegie Mellon University's Mach operating system [Tevanian et al. 87] support multiple address spaces *and* multiple threads within each address space. Threads in the same address space communicate directly with one another using shared memory; threads communicate across address space boundaries using messages. The cost of creating new threads is significantly less than that of creating whole address spaces, since threads in the same address space can share per-program resources. Figure 1 illustrates the various ways in which threads and address spaces can be organized by an operating system.

2.2 Basic Thread Functionality

At its most basic level, a thread consists of a program counter (PC), a set of registers, and a stack of procedure activation records containing variables local to each procedure. A thread also needs a control block to hold state information used by the thread management system: a thread can be *running* on a processor, *ready-to-run* but waiting for a processor to become available, *blocked*

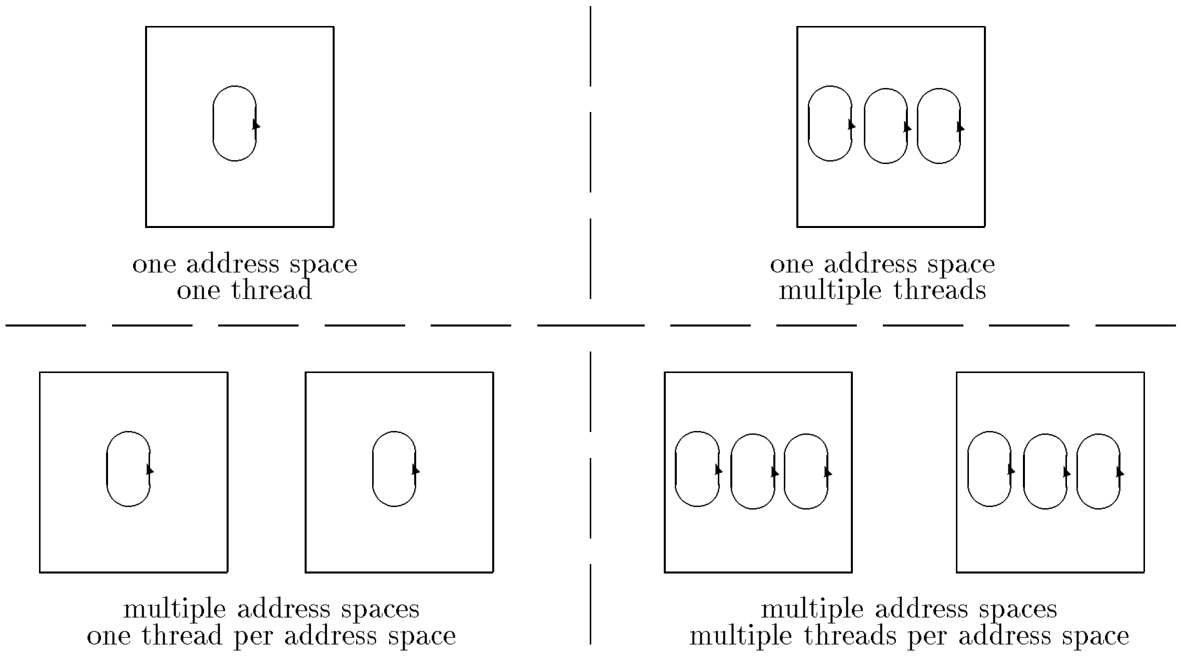

Figure 1: Threads and address spaces. MS-DOS is an example of a "one address space, one thread" system. A Java runtime engine is an example of one address space with multiple threads. The UNIX operating system is an example of multiple address spaces, with one thread per address space. Windows NT is an example of a system that has multiple address spaces and multiple threads per address space.

waiting for some other thread to communicate with it, or *finished.* Threads that are ready-to-run are kept on a *ready-list* until they are picked up by an idle processor for execution. There are four basic thread operations:

Spawn: A thread can create or "spawn" another thread, providing a procedure and arguments to be run in the context of a new thread. The spawning thread allocates and initializes the new thread's control block and places the thread on the ready-list.

Block: When a thread needs to wait for an event, it may block (saving its PC and registers) and relinquish its processor to run another thread.

Unblock: Eventually, the event for which a blocked thread is waiting occurs. The blocked thread is marked as ready-to-run and placed back on the ready-list.

Finish: When a thread completes (usually by returning from its initial procedure), its control block and stack are deallocated, and its processor becomes available to run another thread.

When threads can communicate with one another through shared memory, *synchronization* is necessary to ensure that threads don't interfere with each other and corrupt common data structures. For example, if two threads each try to add an element to a doubly-linked list at the same time, one or the other element may be lost, or the list could be left in an inconsistent state. *Locks* can solve this problem by providing mutually exclusive access to a data structure or region of code. A lock is acquired by a thread before it accesses a shared data structure; if the lock is held by another thread, the requesting thread blocks until the lock is released. (The code that a thread executes while holding a lock is called a *critical section.*) By serializing accesses, the programmer can ensure that threads only see and modify a data structure when it is in a consistent state.

When a program's work is split among multiple threads, one thread may store a result read by another thread. For correctness, the reading thread must block until the result has been written. This data dependency is an example of a more general synchronization object, the *condition variable*, which allows a thread to block until an arbitrary condition has been satisfied. The thread that makes the condition true is responsible for unblocking the waiting thread.

One special form of a condition variable is a *barrier*, which is used to synchronize a set of threads at a specific point in the program. In the case of a barrier, the arbitrary condition is "have all threads reached the barrier?" If not, a thread blocks when it reaches the barrier. When the final thread reaches the barrier, it satisfies the condition and *raises* the barrier, unblocking the other threads.

If a thread needs to compute the result of a procedure in parallel, it can first spawn a thread to execute the procedure. Later, when the result is needed, the thread can perform a *join* to wait for the procedure to finish and return its result. In this case, the condition is "has a given thread finished?" This technique is useful for increasing parallelism, since the synchronization between the caller and the callee takes place when the procedure's result is needed, rather than when the procedure is called.

Locks, barriers and condition variables can all be built using the basic block and unblock operations. Alternatively, a thread can choose to *spin-wait* by repeatedly polling until an anticipated event occurs, rather than relinquishing the processor to another thread by blocking. Although spin-waiting wastes processor time, it can be an important performance optimization when the expected waiting time is less then the time it takes to block and unblock a thread. For example, spin-waiting is useful for guarding critical sections that contain only a few instructions.

3 Issues In Thread Management

This section considers the issues that arise in designing and implementing a thread management system as they relate to the programmer, the operating system, and the performance of parallel programs.

3.1 Programmer Issues

3.1.1 Programming Models

The flexibility to adapt to different programming models is an important attribute of thread systems. Parallelism can be expressed in many ways, each requiring a different interface to the thread system and making different demands on the performance of the underlying implementation. At the same time, a thread system that strives for generality in handling multiple models is likely to be well-suited to none.

One general principle is that the programmer should choose the most restrictive form of synchronization that provides acceptable performance for the problem at hand. For coordinating access to shared data, messages are a more restrictive, and for many kinds of parallel programs, more appropriate form of synchronization than locks and condition variables. Threads share information by explicitly sending and receiving messages to one another, as if they were in separate address spaces, except that the thread system uses shared memory to efficiently implement message-passing.

There are some cases where explicit control of concurrency may not be necessary for good parallel performance. For instance, some programs can be structured around a Single Instruction

Multiple Data (SIMD) model of parallelism. With SIMD, each processor executes the same instruction in lockstep, but on different data locations. Because there is only one program counter, the programmer need not explicitly synchronize the activity of different processors on shared data, thus eliminating a major source of confusion and errors.

Perhaps the simplest programmer interface to the thread system is none at all: the compiler is completely responsible for detecting and exploiting parallelism in the application. The programmer can then write in a sequential language; the compiler will make the transformation into a parallel program. Nevertheless, the compiled program must still use some kind of underlying thread system, even if the programmer does not. Of course, there are many kinds of parallelism that are difficult for a compiler to detect, so automatic transformation has a limited range of use.

3.1.2 Language Support

Threads can be integrated into a programming language; they can exist outside the language as a set of subroutines that explicitly manage parallelism; or they can exist both within and outside the language, with the compiler and programmer managing threads together.

Language support for threads is like language support for object-oriented programming or garbage collection — it can be a mixed blessing. On one hand, the compiler can be made responsible for common bookkeeping operations, reducing programming errors. For example, locks can automatically be acquired and released when passing through critical sections. Further, the types of the arguments passed to a spawned procedure can be checked against the expected types for that procedure. This is difficult to do without compiler support.

On the other hand, language support for threads increases the complexity of the compiler, an important factor if a multiprocessor is to support more than one programming language. Further, the concurrency abstractions provided by a single parallel programming language may not do quite what the programmer wants or needs, making it necessary to express solutions in ways that are unnatural or inefficient.

A reasonable way of getting most of the benefits of language support without many of the disadvantages is to define both a language and a procedural interface to the thread management system. Common operations can be handled transparently by the compiler, but the programmer can directly call the basic thread management routines when the standard language support proves insufficient.

3.1.3 Granularity of Concurrency

The frequency with which a parallel program invokes thread management operations determines its *granularity.* A *fine-grained* parallel program creates a large number of threads, or uses threads that frequently block and unblock, or both. Thread management cost is the major obstacle to fine-grained parallelism. For a parallel program to be efficient, the ratio of thread management overhead to useful computation must be small. If thread management is expensive, then only *coarse-grained* parallelism can be exploited.

More efficient threads allow programs to be finer-grained, which benefits both structure and performance. First, a program can be written to match the structure of the problem at hand, rather than the performance characteristics of the hardware on which the problem is being solved. Just as a single-threaded environment on a uniprocessor can prevent the programmer from composing a program to reflect the problem's logical concurrency, a coarse-grained environment can be similarly restrictive. For example, in a parallel discrete-event simulation, physical objects in the simulated system are most naturally represented by threads that simulate physical interactions by sending messages back and forth to one another; this representation is not feasible if thread operations are too expensive.

Performance is the other advantage of fine-grained parallelism. In general, the greater the length of the ready-list, the more likely it is that a parallel program will be able to keep all of the available processors busy. When a thread blocks, its processor can immediately run another thread provided one is on the ready-list. With few threads though, as in a coarse-grained program, processors idle while threads do I/O or synchronize with one another.

The performance of a fine-grained parallel program is less sensitive to changes in the number of processors available to an application. For example, consider one phase of a coarse-grained parallel program that does fifty CPU-minutes worth of work. If the program creates five threads on a five processor machine, the phase finishes in just ten minutes. But, if the program runs with only four processors, then the execution time of the phase *doubles* to twenty minutes: ten minutes with four processors active followed by ten minutes with one processor active. (Preemptive scheduling, which could be used to address this problem, has a number of serious drawbacks, which are discussed in Section 3.2.2.) If the program had originally been written to use fifty threads, rather than five, then the phase could have finished in only thirteen minutes — a reasonable degradation in performance.

Of course, one could argue that the programmer erred in writing a program that was dependent on having exactly five processors. The program should have been parameterized by the number of processors available when it starts. But, even so, good performance can't be ensured if that number can vary, as it can on a multiprogrammed multiprocessor. We consider further the issues of multiprogramming in the next section.

3.2 Operating System Issues

3.2.1 Multiprogramming

Multiprogramming on a uniprocessor improves system performance by taking advantage of the natural concurrency between computation and I/O. While one program waits for an I/O request, the processor can be running some other program. Because the processor and I/O devices are kept busy simultaneously, more jobs can be completed per unit time than if the system ran only one program at a time.

A multiprogrammed multiprocessor has an analogous advantage. Ideally, periods of low parallelism in one job can be overlapped with periods of high parallelism in another job. Further, multiprogramming allows the power of a multiprocessor to be used by a collection of simultaneously running jobs, none of which by itself has enough parallelism to fully utilize the multiprocessor.

3.2.2 Processor Scheduling

Processor scheduling can be characterized by whether physical processors are assigned directly to threads or are first assigned to jobs and then to threads within those jobs. The first approach, called *one-level* scheduling, makes no distinction between threads in the same job and threads in different jobs. Processors are shared across all runnable threads on the system so that all threads make progress at relatively the same rate. In this case, threads from all jobs are placed on one ready-list that supplies all processors, as shown in Figure 2. Although this scheme makes sense for a uniprocessor operating system, it has some unpleasant performance implications on a multiprocessor.

The most serious problem with one-level scheduling occurs when the number of runnable threads exceeds the number of physical processors, because preemptive scheduling is necessary to allocate processor time to threads in a fair manner. With preemption, a processor can be taken away from one thread and given to another at any time. In a sequential program, preemption has a well-defined effect: the program goes from the running state to the not-running state as its one thread is preempted. The effect of preemption on the performance of a sequential program is also well-defined: if n CPU-intensive jobs are sharing one processor in a preemptive, round-robin fashion, then each job receives $1/n$th the processor and is slowed down by a factor of n (modulo the preemption and scheduling overhead).

For a parallel program, though, the effects of "untimely" processor preemption on performance

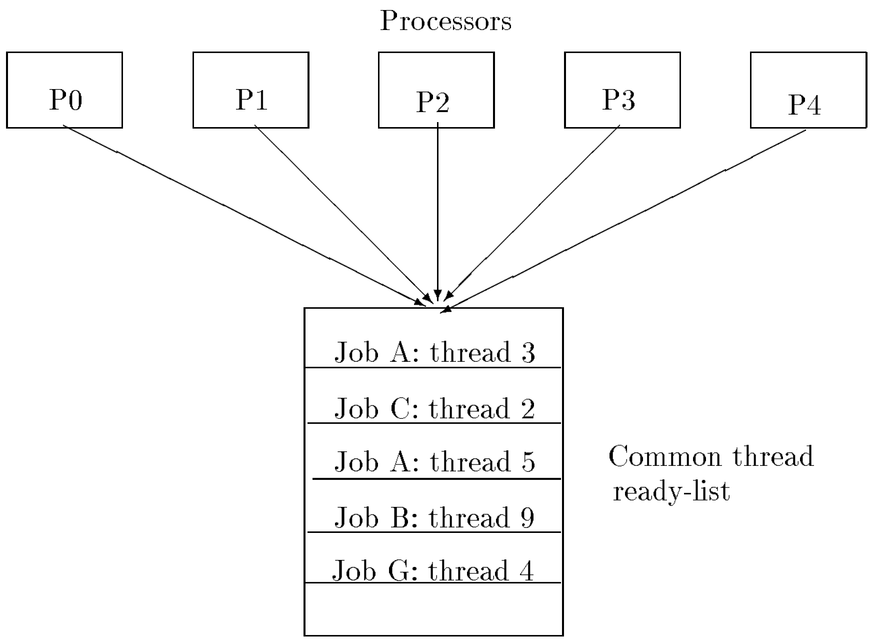

Figure 2: One-level thread scheduling

can be more dramatic. In the previous section, we saw how a coarse-grained program can be slowed down by a factor of two when the number of processors is decreased from five to four. That program exemplified a problem that occurs more generally with preemption and barrier-based synchronization. The program had an implicit barrier, which was the final instruction in the phase. Until all threads reached that instruction, the program could not continue. When one processor was removed, it took twice as long to reach the barrier because not all threads within the job could make progress at an equal rate.

Preemptive multiprocessor scheduling also affects program performance when locks are used, but for a different reason than with barriers. Suppose a thread holding a lock while in a critical section is unexpectedly preempted by the operating system. The lock will remain held until the thread is rescheduled. As threads on other processors try to acquire the lock, they will find it held and be forced to block. It is even possible that, as more threads block waiting for the lock to be freed, the number of that job's runnable threads drops to zero and the application can make no progress until the preempted thread is rescheduled. The overhead of this unnecessary blocking and unblocking slows down the program's execution.

In the previous section, we saw how fine-grained parallelism can improve a program's performance by increasing the chance that a processor will find another runnable thread when its current thread blocks. Unfortunately, a fine-grained parallel program that "packs" the ready-list interacts

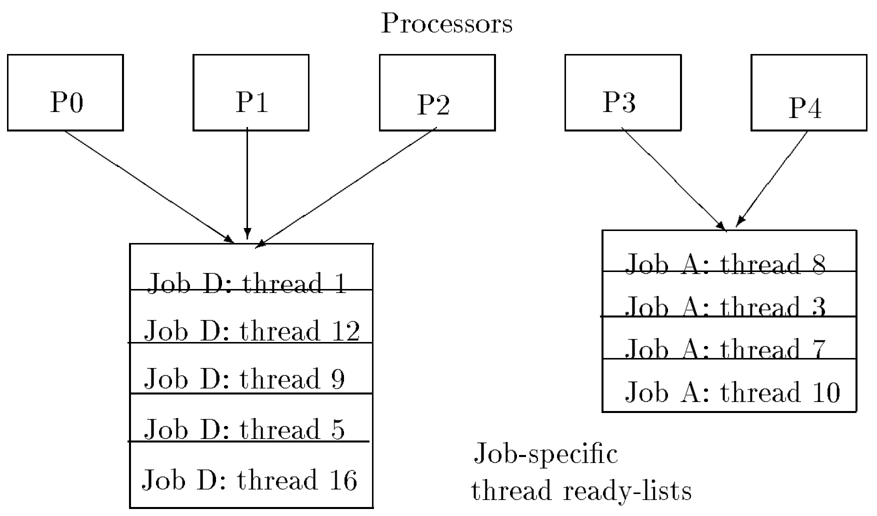

Figure 3: Two-level thread scheduling

badly with the behavior of a one-level scheduler. In particular, when a program's thread blocks in the kernel on an I/O request, the parallelism of the program can only be maintained if the kernel can schedule another of the program's threads in place of the one that blocked. This benefit, though, comes at the cost of increased preemption activity and diminished overall performance.

The problems of one-level scheduling are addressed by two-level schedulers. With a two-level scheduler, processors are first assigned to a job, and then threads within that job are executed only on the assigned processors. Each job has its own ready-list, which is used only by the job's processors, as shown in Figure 3. Thread preemption may no longer be necessary with a two-level scheduler since a preempted thread will only be replaced by another thread from the same job. Further, for long intervals, a processor runs only threads from the same application, so the cost of switching between threads is kept low.

In a two-level scheduling system, processors can be allocated to jobs either statically or dynamically. A static two-level scheduler never changes the number of processors given to a job from its initial allocation; if some of those processors are needed by another job, the operating system must preempt all of the job's processors. A dynamic scheduler can adapt the number of processors assigned to each job according to changing conditions.

Dynamic two-level scheduling can give better performance, because it overlaps periods of poor parallelism in one job with periods of high parallelism in another. One difficulty with a dynamic scheduler is that it requires more information from an application describing the current processor requirements. As a result, though, dynamic scheduling can also more easily handle changes in the number of running jobs. For example, when a job finishes, its processors can be re-allocated to a running job whose parallelism is increasing. To avoid the problems of one-level scheduling, though, it is crucial that the operating system coordinate with each application when it needs to preempt processors (e.g., to avoid preempting a processor when it would seriously affect performance). A dynamic scheduler always has the option, when it needs processors and no application has any available, of reverting to a static policy.

3.2.3 Kernel- vs. User-Level Thread Management

Processor scheduling controls the allocation of processors to jobs. The operating system must be responsible for processor scheduling because processors are a hardware resource and shifting a processor from one job to another involves updating per-processor address space hardware registers. Spawning a thread so that it runs on an already allocated processor, however, does not require modifying privileged state. Thus, thread management and scheduling within a job can be done entirely by the application instead of by the operating system. In this case, thread management operations can be implemented in an application-level library. The library creates virtual processors using the operating system's processor scheduling interface, and schedules the application's threads on top of these virtual processors.

Unlike processor allocation, where a single system-wide scheduling policy can be used, thread scheduling policies benefit from being application-specific. Some applications perform well if their threads are scheduled according to some fixed policy, such as first-in-first-out or last-in-first-out, but others need to schedule threads according to fixed, or even dynamically changing priorities. For example, consider a parallel simulation where each simulation object is represented by its own thread. Different objects become sequential bottlenecks at different times in the simulation; the amount of parallelism can be increased by preferentially scheduling these objects' threads.

It is difficult to provide sufficient thread scheduling flexibility with kernel-level threads. While the kernel could define an interface that allows each application to select its thread scheduling policy, it is unlikely that the system designer could foresee all possible application needs.

Thread management involves more than scheduling. A tradeoff exists between user- and kernel-level thread management. A user-level implementation provides more flexibility and better performance; implementing threads in the kernel guarantees a uniformity that eases the integration of threads with system tools.

The downside of having many custom-built thread management systems is that there is no "standard" thread. By implication, a kernel-level thread management system defines a single, system-wide thread model that is used by all applications. Operating systems that support only one thread model, like those that support only one programming language, can more easily provide sophisticated utilities, such as debuggers and performance monitors. These utilities must rely on the abstraction and often the implementation of the thread model, and a single model makes it easier to provide complete versions of these tools since their cost can be amortized over a large number of applications. Peripheral support for multiple models is possible, but expensive.

A standard thread model also makes it possible for applications to use libraries, or "canned" software utilities. In the same sense that a standard procedure calling sequence sacrifices speed for the ability to call into separately compiled modules, a standard thread model allows one utility to call into another since they both share the same synchronization and concurrency semantics.

It is important to point out that two-level scheduling does not imply that threads are implemented at the application level; the job-specific ready queues shown in Figure 3 could be maintained either within the operating system or within the application. Also, a user-level thread implementation does not imply two-level scheduling, even though threads *are* being scheduled by the application. This implication only holds in the absence of multiprogramming, or in cases where processors are explicitly allocated to jobs. For example, a user-level thread implementation built on top of UNIX processes that share memory suffers from the same problems relating to preemption and I/O as do one-level kernel threads because both are scheduled in a job-independent fashion.

3.3 Performance

The performance of thread operations determines the granularity of parallelism that an application can effectively use. If thread operations are expensive, then applications that have inherently fine-grained parallelism must be re-structured (if that is even possible) to reduce the frequency of those operations. As the cost of thread operations begins to approach that of a few procedure calls, several issues become performance-critical that, for slower operations, would merely be second-order effects.

Simplicity in the thread system's implementation is crucial to performance [Anderson et al. 89]. There is a performance advantage to building multiple thread systems, each tuned for a single type of application. Even simple features that are needed by only some applications, such as saving and restoring all floating point registers on a context switch, will markedly affect the performance of applications that do not need the functionality. Each context switch takes only tens of instructions; a feature that adds even a few more instructions must have a large compensating advantage to be worthwhile. For example, the ability to preemptively schedule threads within each job makes the thread management system more sluggish at several levels, because preemption must be disabled (and then reenabled) whenever scheduling decisions are being made. These scheduling decisions are on the critical path of all thread management operations.

Although kernel-level thread management simplifies the generation and maintenance of system tools, it increases the baseline cost of all thread management operations. Just trapping to the operating system can cost as much as the thread operation itself, making a kernel implementation unattractive for high-performance applications. Further, the generality that must be provided by a kernel-level thread scheduler hurts the performance of those applications needing only basic service. Kernel-level threads are less able to "cut corners" by exploiting application-specific knowledge. With a user-level thread system, the thread management system can be stripped down to provide exactly the functions needed by an application and no more. User-level thread operations also avoid the cost of trapping to the kernel.

Other performance issues have less to do with what a thread system does, than with how it goes about doing it. For example, using a centralized ready-list can limit performance for applications that have extremely fine-grained parallelism. The ready-list is a shared data structure that must be locked to prevent it from being modified by multiple processors simultaneously. Even if the ready-list critical sections consist only of simple enqueue and dequeue operations, they can become a sequential bottleneck, since there is little other work involved in spawning/finishing or blocking/unblocking a thread. An application for which thread overhead is twenty percent of the total execution time, and half of that overhead is spent accessing the ready-list, then its maximum speedup (the time of the parallel program on P processors divided by the time of the program on one processor) is limited to ten.

The bottleneck at the ready-list can be relieved by giving each processor its own ready-list. In this way, enqueueing and dequeueing of work can occur in parallel, with each processor using a different data structure. When a processor becomes idle, it checks its own list for work, and if that list is empty, it scans other processors' lists so that the workload remains balanced.

Per-processor ready-lists have another nice attribute: threads can be preferentially scheduled on the processor on which they last ran, thereby preserving cache state. Computer systems use caches to take advantage of the principle of *locality*, which says that a thread's memory references are directed to or near locations that have been recently referenced. By keeping references close to the processor in fast cache memory, the average time to access a memory location can be kept low. On a multiprocessor, a thread that has been re-scheduled on a different processor will initially find fewer of its references in that processor's cache. For some applications, the cost of fetching these references can exceed the processing time of the thread operation that caused the thread to migrate.

The role of spin-waiting as an optimization technique changes in the presence of high-performance

Basic	**Windows NT**	**Presto**	**Multilisp**
Spawn	thread_create;thread_resume	Thread::new; Thread::start	(future...)
Block	thread_suspend	Thread::sleep	*Touch unresolved future.*
Unblock	thread_resume	Thread::wakeup	*When future is resolved.*
Finish	thread_terminate	Thread::terminate	*Resolve this future.*

Table 1: The Basic Operations of Thread Management Systems

thread operations. If a thread needs to wait for an event, it can block, relinquishing its processor, or spin-wait. A thread must spin-wait for low-level scheduler locks, but in application code a thread should block instead of spin if the event is likely to take longer than the cost of the context switch. Even though context switches can be implemented efficiently, reducing the need to spin-wait, a hidden cost is that context switches also reduce cache locality.

4 Three Modern Thread Systems

We now outline three modern thread management systems for multiprocessors: Windows NT, Presto, and Multilisp. The choices made in each system illustrate many of the thread management issues raised in the previous section.

The thread management primitives for each of these systems are shown in Table 1. The table is organized to indicate how the primitives in one system relate to those in the others, as well as those provided by the basic thread interface outlined in Section 2.2.

Windows NT is an operating system designed to support Microsoft Windows applications on uniprocessors, shared memory multiprocessors, and distributed systems. Windows NT supports multiple threads within an address space. Its thread management functions are implemented in the Windows NT kernel. Since NT's underlying thread implementation is shared by all parallel programs, system services such as debuggers and performance monitors can be economically provided.

Windows NT's scheduler uses a priority-based one-level scheduling discipline. Because Windows NT allocates processors to threads in a job-independent fashion, a parallel program running on top of the Windows NT thread primitives (or even a user-level thread management system based on those primitives) can suffer from anomalous performance profiles due to ill-timed preemptive decisions made by the one-level scheduling system.

Presto is a user-level thread management system originally implemented on top of Sequent's DYNIX operating system, but later ported to DEC workstations. DYNIX provides a Presto program with a fixed number of UNIX processes that share memory. The Presto run-time system treats these processes as virtual processors and schedules the user's threads among them. Presto's thread interface is nearly identical to Windows NT's.

Presto is distinguished from most other thread systems in that it is structured for flexibility. Presto is easy to adapt to application-specific needs because it presents a uniform object-oriented interface to threads, synchronization, and scheduling. The object-oriented design of Presto encourages multiple implementations of the thread management functions and so offers the flexibility to efficiently accommodate differing parallel programming needs.

Presto has been tuned to perform well on a multiprocessor; it tries to avoid bottlenecks in the thread management functions through the use of per-processor data structures. Presto does not provide true two-level scheduling, even though the thread management functions (e.g., thread scheduling) are implemented in an application library accessible to the user, DYNIX, the base operating system, schedules the underlying virtual processors (UNIX processes) any way that it chooses. Although a Presto program can request that its virtual processors not be preempted, the operating system offers no solid guarantee. As a result, kernel preemption threatens the performance of Presto programs in the same was as it does Windows NT programs.

Although Windows NT and Presto are implemented differently, the interfaces to each represents a similar style of parallel programming in which the programmer is responsible for explicitly spawning new threads of execution *and* for synchronizing their access to shared data. This style is not accidental, but reflects the basic function of the underlying hardware — processors communicating through shared memory. One criticism often made of this style is that it forces the programmer to think about coordinating many concurrent activities, which can be a conceptually difficult task.

Multilisp demonstrates how thread support can be integrated into a programming language in order to simplify writing parallel programs. In Multilisp, a multiprocessor extension to LISP, the basic concurrency mechanism is the `future`, which is a reference to a data value that has not yet been computed. The `future` operator can be included in any Multilisp expression to spawn a new thread which computes the value of the expression in parallel. Once the value has been computed, the future *resolves* to that value. In the meantime, any thread that tries to use the future's value in an expression automatically blocks until the future is resolved. The language support provided by Multilisp can be implemented on top of a system like Windows NT or Presto using locks and condition variables.

With Multilisp, the programmer does not need to include any synchronization code beyond the future operator; the Multilisp interpreter keeps track of which futures remain unresolved. By contrast, using the Windows NT or Presto thread primitives, the programmer must add calls to the appropriate synchronization primitives wherever the data is needed. Multilisp, like Presto, uses per-processor ready-lists to reduce contention in scheduling operations.

5 Summary

This article has examined some of the key issues in thread management for shared-memory multiprocessors.

Shared-memory multiprocessors are now commonplace in both commercial and research computing. These systems can easily be used to increase throughput for multiprogrammed sequential jobs. However, their greatest potential – as yet not fully realized – is for accelerating the execution of single, parallelized programs.

As programmers make use of finer-grained parallelism, the design and implementation of the thread management system becomes increasingly crucial. Modern thread management systems must address the programmer interface, the operating system interface, and performance optimizations; language support and scheduling techniques for multiprogrammed multiprocessors are two areas that require further research.

References

[Anderson et al. 89] Anderson, T. E., Lazowska, E. D., and Levy, H. M. The Performance Implications of Thread Management Alternatives for Shared Memory Multiprocessors. In *1989 ACM SIGMETRICS and Performance '89 Conference on Measurement and Modeling of Computer Systems*, pages 49–60, May 1989.

[Bershad et al. 88] Bershad, B., Lazowska, E., and Levy, H. PRESTO: A System for Object-Oriented Parallel Programming. *Software Practice and Experience*, 18(8):713–732, August 1988.

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

[Dijkstra 68] Dijkstra, E. W. Cooperating Sequential Processes. In *Programming Languages*, pages 43–112. Academic Press, 1968.

[Enc 86] Encore Computer Corporation. *UMAX 4.2 Programmer's Reference Manual*, 1986.

[Halstead 85] Halstead, R. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transaction on Programming Languages and Systems*, 7(4):501–538, October 1985.

[Hoare 74] Hoare, C. A. R. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[Hoare 78] Hoare, C. A. R. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[Redell et al. 80] Redell, D. D., Dalal, Y. K., Horsley, T. R., Lauer, H. C., Lynch, W. C., McJones, P. R., Murray, H. G., and Purcell, S. C. Pilot: An Operating System for a Personal Computer. *Communications of the ACM*, 23(2):81–92, February 1980.

[Seq 88] Sequent Computer Systems, Inc. *Symmetry Technical Summary*, 1988.

[Tevanian et al. 87] Tevanian, A., Rashid, R. F., Golub, D. B., Black, D. L., Cooper, E., and Young, M. W. Mach Threads and the Unix Kernel: The Battle for Control. In *Proceedings of the 1987 USENIX Summer Conference*, pages 185–197, 1987.