User-Level Threads and Interprocess Communication

Michael J. Feeley, Jeffrey S. Chase, and Edward D. Lazowska

Department of Computer Science and Engineering, FR-35
University of Washington
Seattle, WA 98195

Technical Report 93-02-03

Abstract

User-level threads have performance and flexibility advantages over both Unix-like processes and kernel threads. However, the performance of user-level threads may suffer in multiprogrammed environments, or when threads block in the kernel (e.g., for I/O). These problems can be particularly severe in tasks that communicate frequently using IPC (e.g., multithreaded servers), due to interactions between the user-level thread scheduler and the operating system IPC primitives. Efficient IPC typically involves processor handoff that blocks the caller and unblocks a thread in the callee; when combined with user-level threads, this can cause problems for both caller and callee, particularly if the caller thread should subsequently block.

In this paper we describe a new user-level thread package, called OThreads, designed to support blocking and efficient IPC for a system based on traditional kernel threads. We discuss the extent to which these problems can be solved at the user level without kernel changes such as *scheduler activations*. Our conclusion is that problems caused by application-controlled blocking and IPC can be resolved in the user-level thread package, but that problems due to multiprogramming workload and unanticipated blocking such as page faults require kernel changes such as scheduler activations.

1 Introduction

User-level threads provide a significant performance benefit over both kernel threads and Unix processes. In a user-level thread system, common thread operations such as creation, scheduling and synchronization are supported by a runtime library included with user programs. This library provides thread support with code that runs at user-level without the need for costly kernel calls, multiplexing user-level threads on top of whatever execution abstraction is provided by the kernel (e.g., processes, kernel threads, or scheduler activations). User-level support also gives applications

This work was supported in part by the National Science Foundation under Grants No. CCR-8619663 and CCR-8907666, by the Washington Technology Center, and by the Digital Equipment Corporation Systems Research Center and External Research Program.

the flexibility to change scheduling policies or to tailor other aspects of thread management. These benefits have led to widespread acceptance of user-level threads for parallel programming as well as for supporting concurrency on uniprocessors.

There are, however, problems with user-level threads that are multiplexed on top of kernel threads or processes. These problems occur when a user-level thread performs blocking operations such as I/O or IPC, and in presence of multiprogramming. When a user-level thread makes an I/O call, for example, the kernel thread on which it is running blocks. For the duration of the call, the processor is available but the user-level scheduler has no kernel thread on which to run ready threads. If a new kernel thread is created to compensate for this, a problem occurs when the I/O completes, as there are now two kernel threads running user-level threads on top of one processor. Anderson points out this *two-level scheduling* problem and proposes that *scheduler activations* replace kernel threads as the execution vessel provided by the kernel [Anderson et al. 91]. Scheduler activations provide a mechanism for the kernel to upcall the user-level thread scheduler to inform it of kernel scheduling changes that effect it — such as when one of its threads blocks or unblocks. The kernel ensures that the number of scheduler activations never exceeds the number of processors. This problem was also identified by researchers working on the Psyche parallel operating system. Pscyhe uses kernel threads, but upcalls the user-level to inform it of scheduling changes [Marsh et al. 91].

Anderson's solution requires changes to the operating system kernel, as does the Pscyhe approach. Changing the kernel is difficult; a production quality implementation must be general, reliable and efficient. As a result, though research prototypes exist, systems with scheduler activations are not presently available. We agree with Anderson and others that, in the long run, scheduler activations (or something like it) is the only complete solution to the two-level scheduling problem. But, until scheduler activations are widely available, the benefits of user-level threads are severely restricted by the problems associated with blocking and IPC. In this paper we show that it is possible to get many of the benefits provided by scheduler activations on a standard operating system (e.g., OSF/1 or Mach) by changing the user-level thread package.

OThreads

The purpose of this paper is to describe a new user-level thread package we designed and built, called *OThreads*, that provides a partial solution to the two-level scheduling problem without the need for kernel changes; OThreads runs on top of kernel threads. The approach we take is to use the user-level thread scheduler to maintain an invariant in each *domain* (process) on the number of kernel threads that will run there. Kernel threads are started, as needed, in anticipation of blocking calls or in response to increases in parallelism and stopped when unblocking results in an excess of kernel threads running in the domain.

OThreads provides only a partial solution; in particular, it assumes that blocking can be anticipated by the application, and that multiprogramming will be limited. We show that, given these assumptions, OThreads is a workable compromise, allowing domains to use user-level threads and user-level synchronization without paying high performance costs for blocking operations such as I/O, RPC, and IPC.

Organization of Paper

The rest of this paper is organized as follows. In Section 2, we discuss the two-level scheduling problem as it relates to an environment of cooperating domains communicating by IPC and show where existing thread packages suffer from two-level scheduling problems or otherwise fail to support this environment. In Section 3, we present OThreads and describe its approach to solving the problems related to blocking and IPC. In Section 4, we evaluate OThreads and provide performance measurements of the overhead associated with kernel thread management in OThreads and comparisons with another thread package (Mach's CThreads). In Section 5 we discuss the importance of user-level threads to an environment of cooperating domains. We conclude and summarize our work in Section 6.

2 The Problems with Supporting IPC and User-Level Threads

In this section we discuss, in more detail, the problems associated with using user-level threads on an operating system with traditional kernel threads, e.g., OSF/1, Mach or Windows NT. These problems occur when applications that use user-level threads make blocking calls or communicate using IPC. To illustrate the problems, we begin by describing an example application environment in which both user-level threads and frequent IPC must be supported. Section 5 discusses why user-level threads are needed in this environment.

2.1 Cooperating-Domain Environment

OThreads was developed as part of the Opal project, whose purpose is to explore new operating system models for wide-address (e.g., 64-bit) architectures [Chase et al. 92a, Chase et al. 92b]. Opal is an operating system environment in which all applications execute in a shared, potentially persistent, virtual address space. Typically, each Opal application runs in a private *protection domain* that limits its threads' access to the shared virtual address space; applications can communicate through shared memory, but only by mutual consent — protected communication still requires IPC.

Opal programs are typically structured as a group of cooperating protection domains with different memory access privileges, e.g., a client and some servers, as shown in Figure 1. Logically, threads move between these passive protection domains in a controlled way (with protected RPC) to change their memory access privileges. In effect, every protection domain is an RPC server. New domains are created as idle RPC servers; their parents (or peers) activate them with RPC calls. A system with even a low multiprogramming workload may have many protection domains, with frequent RPC calls between them. Making these calls fast requires *donating IPC*, which hands-off the processor from caller to callee as part of the call. This minimizes the cost of the control transfer between domains, avoiding the generic scheduling operation that would otherwise be required. This is the approach taken by LRPC [Bershad et al. 89], Mach [Draves 90] and Windows NT [Custer 93].

The threads that run in Opal domains are user-level threads; each domain has a user-level scheduler

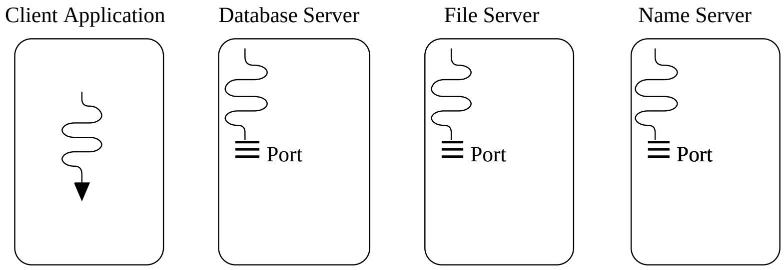

Figure 1: Example of Cooperating Domains — A client domain with one active thread and three *idle* server domains.

that manages the threads in that domain. As we discuss in Section 5, user-level threads are important for a number of reasons including their support for fast user-level synchronization among threads in a single domain as well as for threads in different domains that are sharing memory.

The current Opal prototype is a server above Mach 3.0, together with a collection of specialized runtime libraries and linking utilities. In the prototype, Opal protection domains are just Mach tasks; the Opal server arranges for these tasks to use the same virtual-physical mapping. Communication between domains is with mach_msg(); donating IPC requires that the sending thread block (e.g., to receive a reply). Each domain has a designated *port* on which it receives incoming RPC — *idle* domains must have at least one thread that is blocked in a *receive* operation waiting for incoming RPC, as shown in Figure 1.

In summary, Opal is an example of an environment that needs user-level threads but where using them is problematic. Opal emphasizes the use of cooperating protection domains where threads make frequent IPC calls from domain to domain. This model has three key characteristics that make using user-level threads difficult:

- there are many *idle* domains,
- threads may block, e.g., for RPC,
- IPC is common so it should be fast, *donating* when possible.

In the rest of this section we examine the difficulties that user-level thread packages built on kernel threads have satisfying these demands. In Section 3 we show how OThreads overcomes these difficulties without requiring changes to the operating system.

2.2 The Two-Level Scheduling Problem

The fundamental problem with using user-level threads that are built on kernel threads is caused by the lack of coordination between the user-level thread scheduler in an application and the kernel thread scheduler in the operating system. Without this coordination, the user-level scheduler must assume that the kernel threads it uses to multiplex user-level threads are not scheduled by the

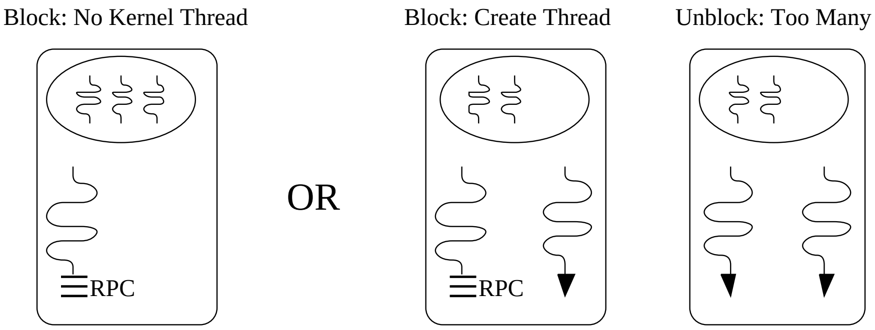

Figure 2: Unsatisfactory Alternatives for Dealing with Blocking

system, i.e. they are always running. When this assumption is violated, a problem known as *two-level scheduling* results. The system is unable to correctly schedule these threads because doing so requires information known only to the user-level scheduler. Lacking this information, the system might preempt a thread at an inopportune time (e.g., when it is holding a spinlock [Zahorjan et al. 91]) or it might leave an important thread suspended while choosing to run an unimportant one.

We are primarily interested in two-level scheduling that occurs when an application makes a blocking system call or when it communicates using IPC. There are other sources of two-level scheduling, such as multiprogramming, but they are not considered in detail here because they are caused by events that cannot be anticipated by the application; as a result, their solutions seem to require changes to the operating system. In contrast, as we show in Section 3, two-level scheduling caused by anticipated events, such as blocking and IPC, can be solved through careful engineering of the user-level thread package.

Figure 2 shows what happens when a thread blocks, in the kernel, to make a remote call (or to do I/O). In this example, there are three other user-level threads waiting to run; but the only kernel thread available to the application is blocked. To make effective use of the processor and run the other threads while that thread is blocked, the system would need to start another kernel thread. If it does, however, two-level scheduling problems result when the returning RPC (or I/O completion) causes there to be more kernel threads running than processors. The key to solving the blocking problem is to balance the number of active kernel threads in the application so that there are always as many as are needed to run the available user-level threads but never more than the number of processors.

The problems associated with local IPC differ because IPC involves transfering control from one domain to another, rather than blocking a thread in one domain. As mentioned earlier, the most effective way to make this transfer is for the caller to donate its processor to the calee for the duration of the call. The difficulty arises if a callee thread blocks. As before, to handle this blocking, the system needs to start another kernel thread. But in which domain, the caller or the callee? One or both domains may have user-level threads waiting to run. In either case, when the first thread unblocks, a two-level scheduling problem will exist. If the caller received the new thread, as in Figure 3, the situation is complicated by the fact that the two-level scheduling now involves multiple domains.

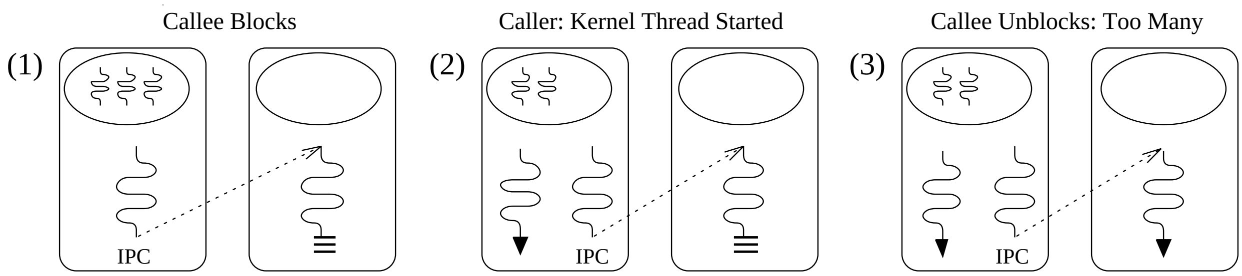

Figure 3: Donating IPC — Callee Thread Blocks

2.3 Examples of Problems with Supporting this Environment

In this section we outline three popular user-level thread packages built on top of kernel threads. Each is specialized to a particular operating environment and each suffers two-level scheduling problems when used in an environment of cooperating domains.

2.3.1 Presto and Amber

Presto is a user-level thread package designed to support fine-grain parallel programming, but not communication [Bershad et al. 88]. When a Presto application is started, it is assigned some number of processors for its exclusive use. Presto creates this number of *scheduler* threads, which spin on the thread ready queue waiting for user-level threads to run. If a Presto thread issues a call that blocks in the kernel (to wait for a message, for example), the kernel thread running the user-level thread is lost to the application for the duration of the call. Amber is a thread package based on Presto for supporting *distributed* parallel programming [Chase et al. 89, Feeley et al. 91].

In both Presto and Amber, the number of kernel threads is never greater than the number of processors allocated to it — no two level scheduling problems. Nevertheless, there are two key problems that make Presto and Amber unsuitable for our environment.

Key Problems

- Blocking calls reduce the number of kernel threads available to a Presto domain. Unlike in many parallel applications, blocking is common in our environment; thus, there would be many periods where there are idle processors and ready user-level threads, but no kernel threads on which to run them.
- When idle, Presto scheduler threads *spin* on the user-level thread ready queue waiting for work. This means that each of the many idle domains in our environment would be consuming valuable processor cycles.

2.3.2 NewThreads

NewThreads is a user-level thread package similar to Presto, designed to support *distributed* IPC [Felten & McNamee 92a, Felten & McNamee 92b]. As with Presto and Amber, a NewThreads application is assigned some number of processors and when idle has that number of *spinblockers* spinning on the ready queue waiting for work. But unlike Presto, one of these spinning threads is also checking a designated *receive* port for messages. When a message is received, the user-level thread waiting for the message is awakened. By using asynchronous IPC, NewThreads avoids two-level scheduling problems due to communication.

Key Problems

- Asynchronous communication works well for distributed IPC but increases the latency of local communication because it disallows the use of donating IPC.
- As with Presto and Amber, idle domains consume processors by spinning on the ready queue.

2.3.3 CThreads

CThreads is the user-level thread package used by many Mach servers and applications [Cooper & Draves 88]. CThreads was designed to support calls to Mach message primitives from a user-level thread, but it does this without trying to avoid two-level scheduling problems.

A CThreads application can specify a limit on the number of kernel threads that will be created by the scheduler to run user-level threads (the *creation limit*). Kernel threads are created, up to this limit, in response to increases in user-level parallelism and are blocked when parallelism decreases. Unlike in Presto, Amber and NewThreads, idle domains have no running kernel threads — just what we want.

IPC is supported by placing a limit on the number of kernel threads that will block in *receive* operations (the *block limit*). As long as that limit has not been reached, mach_msg() calls block in the kernel, in the usual way. When at the limit, however, only the user-level thread is blocked, freeing the kernel thread to run other user-level threads. By setting the creation limit higher than the block limit, the application can ensure that there will always be enough kernel threads to run its user-level threads. This, however, causes two-level scheduling problems because the creation limit must be greater than the available processors (by at least the block limit). To mitigate this problem, the CThreads *spinlock* primitive was modified to temporarily drop the kernel priority of a thread that is spinning on a held lock. This strategy ensures that a spinner will not preempt the thread that holds the lock; it works well on a uniprocessor, but increases spinlock latency on a multiprocessor. Restartable atomic sequences are another effective strategy for dealing with spinlock preemption on uniprocessors [Bershad et al. 92].

Another way that communication can be supported in CThreads is to set the creation limit equal to the number of processors and use asynchronous IPC, as is done in NewThreads. This would solve the two-level scheduling problem for IPC — but not other forms of blocking — at the expense of common case IPC performance.

Key Problem

- To deal with blocking, there will often be more kernel threads than processors. This severely limits the performance of CThreads synchronization (even with spinlock changes) as we show in performance measurements in Section 4.2.

3 Overview of OThreads

OThreads is a new thread package designed to support fast IPC and blocking from within user-level threads that are built on kernel threads. Our approach is to maintain a *local invariant* in each domain on the number of kernel threads that will run there. The thread package will start and stop kernel threads in order maintain this invariant. New kernel threads are started in response to increases in user-level parallelism and in anticipation of calls from user-level threads that will block in the kernel, e.g., I/O or RPC. Kernel threads are stopped when an I/O completion or incoming message would increase the number of kernel threads beyond the maximum. The application tells the thread package where blocking and communication occur by annotating certain calls as described below; other than this, the application interface is similar to that of most other user-level thread packages.

Maintaining the local invariant only solves part of the two-level scheduling problem. We also need a global invariant that ensures that the aggregate of all the local maximums does not exceed the number of processors on the system. The current implementation of OThreads maintains only the local invariant. In Section 3.3 we outline our proposal for managing the global invariant; with the exception of Section 3.3, the rest of the paper describes OThreads as it is currently implemented.

3.1 The Activation Pool

The OThreads interface and implementation are similar to that of traditional user-level thread packages such as those described in Section 2.3. The main difference is that OThreads adds a new module (C++ class) called the *activation pool*. This module is responsible for managing kernel threads for the thread package and for handling incoming RPC, keeping the right number of kernel threads blocked waiting to receive messages. Figure 4 shows the organization of the thread package. It is divided into three parts: the application, thread scheduler and activation pool. The interfaces among these parts are shown with edges pointing in the direction of the call from one module to another.

The activation pool exports the following interface to the thread scheduler.

- ActivationRequest()
- ActivationIdle()
- Begin_BlockingCall()
- End_BlockingCall()

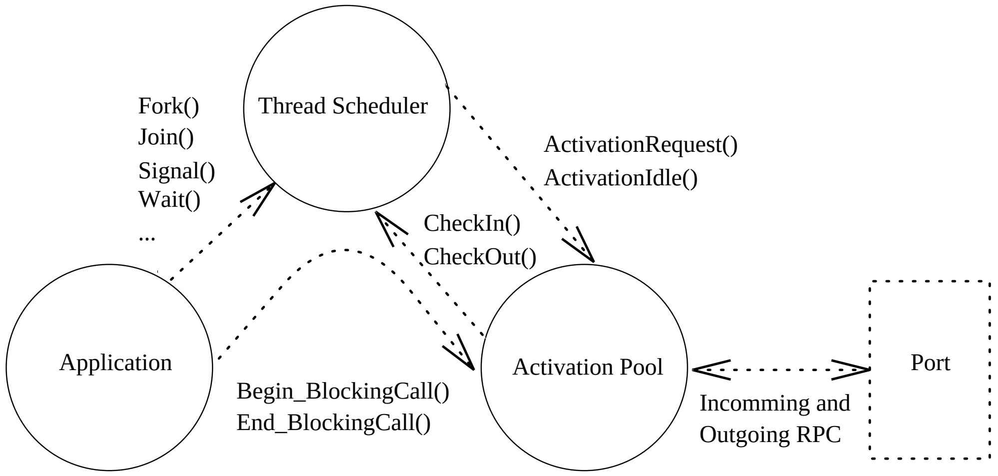

Figure 4: OThreads Interface

The thread scheduler calls the activation pool to request new kernel threads (ActivationRequest) and to stop idle threads (ActivationIdle). The Begin_BlockingCall and End_BlockingCall methods are exported by the thread scheduler to the application. They are used by the application to inform the activation pool of operations that might block in the kernel. When an incoming RPC arrives, the activation pool calls the thread scheduler to check the thread in (CheckIn); it calls (CheckOut) prior to blocking a thread (e.g., to receive another message).

The activation pool encapsulates kernel thread management and communication; this organization has several advantages. OThreads applications and the thread scheduler are essentially the same as they would be for a traditional thread package such as Presto. They are isolated from knowledge of the kernel primitives for concurrency (e.g., kernel threads) and communication (e.g., Mach ports) by the activation pool. We expect that this will allow the same applications and thread scheduler to work with various kernel mechanisms such as: scheduler activations instead of kernel threads, or LRPC or Windows NT LPC instead of Mach ports. Changes will be isolated to the activation pool module (about 700 lines of C++ code).

3.2 Maintaining the Local Invariant

The local invariant limits the number of kernel threads that will run user-level threads in a domain to a set value, the *domain maximum.* The invariant is defined by the following rules.

1. The number of kernel threads running in a domain is be equal to the minimum of the number of runnable user-level threads and the domain maximum.

2. The number of kernel threads blocked waiting to receive incoming RPC in a domain (*listener* threads) is equal to the difference between the domain maximum and the number of kernel threads running in the domain.

For example, if the domain maximum is d, an *idle* domain has no running kernel threads and d threads blocked waiting to receive messages and a *saturated* domain has d running kernel threads and no threads blocked on the receive port.

There are three events that can change the number of kernel threads running in a domain. When any of them occur, the scheduler and activation pool cooperate to maintain the local invariant. The remainder of this section describes each of these events.

3.2.1 Changes in User-Level Parallelism

Changes in user-level parallelism occur as the result of calls from the application. The number of runnable threads increases when a new thread is started with a call to Fork() or when a blocked thread is restarted, e.g., mutex Unlock(), or Signal() on a user-level condition variable. The number decreases when a thread terminates or blocks, e.g., Join(), Lock() on a held mutex, or Wait() on a condition variable. When parallelism increases, the scheduler might have to start a kernel thread and when it decreases it might have to stop one.

Increase When user-level parallelism increases, the scheduler checks to see if it should start another kernel thread. If the number of kernel threads is less than the maximum, it starts one; otherwise, it places the new user-level thread on the ready queue.

Decrease When user-level parallelism decreases, the scheduler informs the activation pool by calling ActivationIdle() in the context of the idling kernel thread. If another *listener* is needed, the activation pool creates a new listener user-level thread and blocks the thread on the receive port; otherwise, it stops the thread.

3.2.2 Handling Blocking

Applications signal calls that might block by bracketing them with Begin_BlockingCall() and End_BlockingCall(). For example, a file I/O operating would look like this.

```
Begin_BlockingCall();
read( ... );
End_BlockingCall();
```

These calls inform the activation pool of blocking events and allow it to create and block kernel threads as needed to maintain the local invariant. All calls that could potentially block need to be bracketed in this way, even those that may not actually block (e.g., the read() could hit in the file cache).[1]

[1] If the call doesn't block, the effect will be to momentarily check the thread out from the scheduler, make the call, then check it back in; the added overhead is small if the ready queue is empty and is the kernel thread start/stop time if it is not (see Table 1).

Begin_BlockingCall() The calling user-level thread is checked out from the scheduler. If there are user-level threads on the ready queue waiting to run, ActivationRequest() is called to start a new kernel thread; if not, a new listener kernel thread is created if necessary.

End_BlockingCall() An attempt is made to check the thread back in. If the number of kernel threads is below the maximum, the thread continues without interruption; otherwise, the user-level thread is placed on the ready queue and the kernel thread is stopped.

3.2.3 Handling IPC

Communication between domains can be either *donating* or *non-donating*. Donating IPC hands off the processor from caller to callee and is usually required for fast IPC mechanisms such as LRPC; other calls such as RPC between nodes must be non-donating. In Section 2.2 we discussed the problems caused by the combination of donating IPC and blocking. The problem occurs when a callee blocks, temporarily freeing a processor, and the caller has runnable user-level threads but the callee does not. In this case, we would like to be able to give the caller a kernel thread for the duration of the blocking call, then stop it when the call unblocks. To do this, however, we need the ability to coordinate between domains which is not currently part of OThreads (see Section 3.3). Instead, we attempt a compromise solution in which we use donating IPC only when the caller ready queue is empty.

Outgoing RPC If the ready queue is empty, the thread checks out from the scheduler and uses donating IPC to make the call; a combined send/receive operation blocks the thread on the domains' receive port, if more listeners are needed, or stops the thread by blocking it on a special *idler* port, if not. If the ready queue is not empty, asynchronous IPC is used to send the message without checking out or blocking the thread; the kernel thread then switches to the next user-level thread on the ready queue.

Incoming RPC When a message is received, a listener thread unblocks and tries to check into the thread scheduler. It succeeds if the number of active kernel threads is less than the maximum or if the call is a donating call (determined from the message header); otherwise, the listener user-level thread is placed on the ready queue and the kernel thread is stopped.

3.3 Maintaining the Global Invariant

In the previous section we describe how OThreads controls the number of kernel threads running in a domain in the face of changes in user-level parallelism, blocking, and IPC. This solves the two-level scheduling problem only when the sum of all the domain maximums on a node is no greater than the number of processors on the node. In the face of a heavily multiprogrammed workload, in particular, maintaining this *global invariant* on kernel threads requires coordination between domains that is not part of the current OThreads implementation. We believe, however, that a scheme similar to [Tucker & Gupta 89] would solve this problem. Our proposed solution assumes that every domain is running OThreads and that domains trust each other to behave properly as we describe below. If that trust is violated, the worst that can happen is that performance will degrade due to two-level scheduling problems; it is not possible for a domain to do more serious harm to another domain.

The Activation Server

The *activation server* is responsible for maintaining the global invariant. When a new application starts, it binds with the activation server by sending it a message. This causes a page of shared memory to be set up that is writable by both the activation server and the application. The application is also given read-only access to the shared areas setup for the other applications on the node. The shared page is initialized with a structure that the application can use to read these other shared areas; the server keeps this structure up to date, as applications come and go from the system, using non-blocking techniques to synchronize with the application.

There are three values stored on the shared page: the ready *queue size*, the *processor count* and the *temporary processor count*. The queue size is maintained by the OThreads user-level scheduler; it represents the number of user-level threads that are ready to run but are waiting for a processor — in effect, a request for more kernel threads. The processor and temporary processor counts together represent the number of kernel threads allocated to the domain; the temporary count is used to allow one domain to temporarily donate a processor to another domain (e.g., for the duration of an RPC). Both processor counts are maintained by the activation server, running sometimes in its own domain and sometimes in the application's domain; it ensures that the aggregate number of kernel threads in all domains does not exceed the number of available processors. Processors are transferred from one domain to another by *donating* IPC that decrements the caller processor count and blocks its kernel thread and then unblocks a listener thread in the callee, incrementing the callee's processor count.

Changes in User-Level Parallelism When user-level parallelism increases, the queue size increases. The activation server (or a thread in any other domain) can detect this and donate a kernel thread to the domain. Deciding when to do this depends upon the processor allocation policy implemented by the activation server. When user-level parallelism decreases and a kernel thread becomes idle, the domain can look for other domains that have non-zero queue sizes — by using the list maintained for it on the shared page — or it can donate the processor to the activation server.

Handling Blocking Anticipated blocking is handled as before, with Begin and End BlockingCall. But now, if the ready queue is empty when a thread checks out prior to a blocking call, the processor can be *temporarily* donated to another domain that has runnable threads, incrementing that domain's temporary processor count. If another kernel thread is needed in the domain when the thread unblocks, domains with temporary processors are the first to be asked to give up a processor. If all processors are busy running user-level threads, the unblocked thread calls the activation server to request a thread and then stops itself. The activation server finds a victim kernel thread in some domain and *preempts* it, telling it to donate its processor to the domain that called the server. Preemption is problematic because, in essence, it is the two-level scheduling problem in another form. For example, care must be taken not to preempt a user-level thread while it is holding a spinlock. Psyche solves this problem by allowing victim threads more time, following the preemption, to exit from a critical section; in scheduler activations, if a thread is preempted while in a critical section, it is rescheduled to run just long enough for it to exit. We could do something similar since, in OThreads, user-level threads have a *non-preemptable* flag that is set while they are holding spinlocks.

Handing IPC All IPC can now be donating, allowing us to use the fastest mechanisms available for every IPC. Previously, we could only do this when the caller domain had no threads on its ready queue, in case the callee blocked. Now, if a callee thread blocks and there are runnable user-level threads in the caller but not the callee, the processor is temporarily donated to the caller for the duration of the call.

The Advantages of Shared Memory

The implementation of the *activation server* benefits from the Opal environment; shared memory is used to reduce the number of cross domain calls needed to coordinate processor allocation. A thread in any domain can access the (queue-size, processor-count, temporary-processor-count) triple of the other domains. It does this by following a linked list that allows it to traverse the shared pages of the domains; it has read-only access to these pages. This list is maintained for it by the server in its shared memory region; each domain has a separate list to prevent an error that corrupts the list in one domain from propagating to other domains. The use of shared memory means that most processor allocation decisions can be made from within an *application domain,* avoiding a protected call to the server. When a kernel thread becomes idle, for example, it can search for a domain in need of processors by looking in shared memory.

4 Evaluation and Performance

In this section we evaluate how well OThreads supports our target domain. In Section 4.1 we show the overhead caused by kernel thread management in OThreads and in Section 4.2 we compare OThreads to CThreads.

4.1 Performance

We implemented OThreads on Mach 3.0 running on a DEC PMAX (MIPS R-3000). It is designed to run either as part of the runtime support for our Opal prototype or independently from Opal.

Operation	*Overhead (μsec)*
begin/end blocking call	2.4
start a kernel thread	140
stop a kernel thread	230

Table 1: Thread Management Overhead

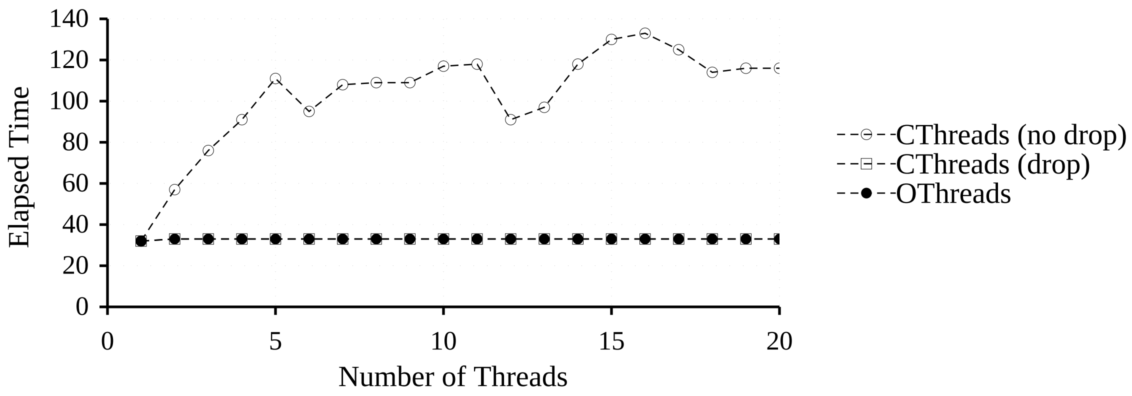

Figure 5: Spinlock Performance

The implementation consists of 2200 lines of Mach independent C++ code (including comments), 450 lines of Mach dependent code and 200 lines of MIPS assembly code. Machine and host operating system dependencies are isolated to facilitate portability.

Table 1 shows the overhead associated with the basic thread management primitives that distinguish OThreads. The combined time for Begin_BlockingCall() and End_BlockingCall() is 2.4 μsec. If a kernel thread needs to be created (ActivationRequest()), the cost is 140 μsec. If the attempt to check back in fails and the kernel thread must be stopped (ActivationIdle()), the cost is 230 μsec. Thus, the worst case overhead for creating a kernel thread prior to a blocking call and idling the unblocked thread when it tries to check back in is 373 μsec. This time includes the overhead associated with blocking and unblocking kernel threads using mach_msg() and the cost of the kernel scheduling that results from the brief period where there are two kernel threads running in the domain.

4.2 Comparison with CThreads

Figure 5 shows measurements taken of a test application that performs 30 million spinlocks[2] (elapsed time is shown in seconds) with from 1 to 20 user-level threads. Each 100,000 steps, threads make an RPC to another domain on the machine where they block on a barrier; once all threads have arrived, the RPCs return for the next 100,000 steps. Doing RPC in this way requires the CThreads application to create a kernel thread for every user-level thread; this is because, with each RPC, a kernel thread is lost to the caller for the duration of the call and no call returns until they have all been made.

The graph shows measurements for OThreads and two versions of CThreads, one with normal spinblocking and another that uses the CThreads strategy of dropping the kernel priority of a spinning thread. Both OThreads and the standard CThreads version (that drops priorities) get the expected linear performance; a CThreads version based on restartable atomic sequences should have similar performance. The degraded performance of the *no drop* version of CThreads demonstrates

[2]We used normal load and store instructions to simulate the spinlock used in this test. This gives spinlock performance on the order of that found with architectures that, unlike the MIPS-3000, support synchronization instructions such as atomic test-and-set.

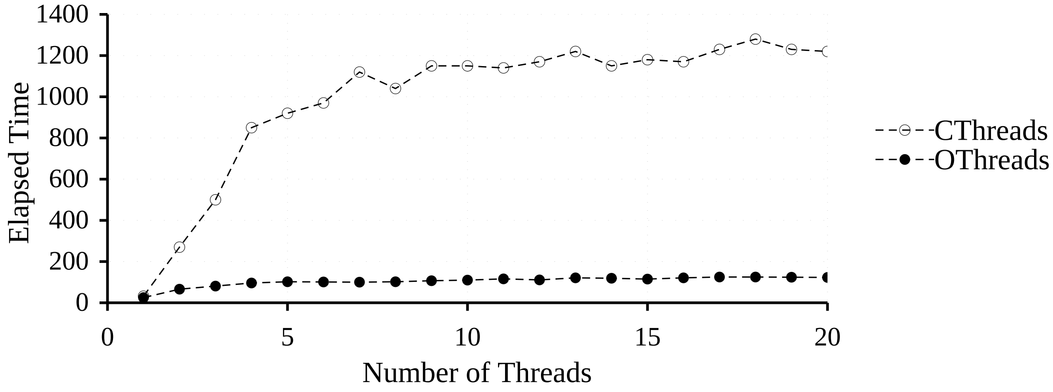

Figure 6: Barrier Performance

the two-level scheduling problem as it relates to spinlocks: if the kernel preempts the lock holder, no thread can make progress until it is rescheduled.

Figure 6 shows measurements taken of a similar application that performs 30 million *blocking* synchronization operations on a user-level barrier, in between RPCs to the remote barrier. The graph shows that OThreads performance is more than an order of magnitude better than CThreads. To see why, recall that in CThreads there is separate kernel thread for every user-level thread, while in OThreads there is only a single kernel thread. In CThreads, when a user-level thread blocks on the local barrier, its kernel thread is also blocked. In OThreads, on the other hand, when a user-level thread blocks, the kernel thread switches to the next runnable user-level thread; since there is always at least one thread waiting to run, the kernel thread is never blocked. This graph dramatically demonstrates one of the performance costs associated with having too may kernel threads running in a domain.

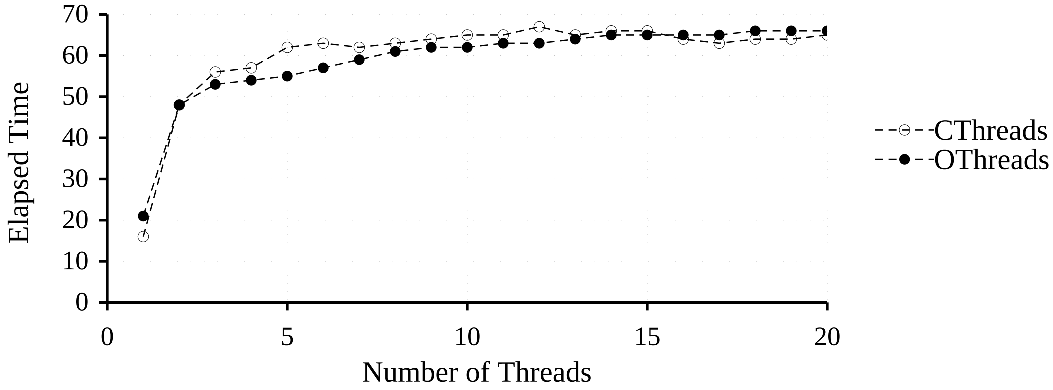

Figure 7: IPC Performance

Figure 7 shows the performance of an application where each thread performs 100,000 cross domain calls. This graph demonstrates that OThreads performance is comparable with CThreads; the overhead associated with thread management in OThreads adds little to the overall time of IPC. Note that the increase in running time is due to the increase in kernel scheduling overhead needed as the number of threads increases.

5 The Importance of User-Level Threads and IPC

The performance and flexibility advantages of user-level threads have been described in detail elsewhere. This section describes additional motivation for user-level threads and efficient IPC in our environment.

5.1 The Importance of Context Independent Thread State

The goal of the Opal structure is to facilitate efficient cooperation using shared memory. In part, we do this by providing a uniform way for applications to name memory regions and pass memory access permissions to their peers. We must also eradicate any task-specific context needed to interpret the contents of memory regions that may be shared. The shared virtual address space plays an important role by ensuring a consistent interpretation of pointers in shared data structures. User-level thread support also contributes to this goal, because user-level threads are independent of any task-specific kernel state. Since a user-level thread that is idle (e.g., blocked on a mutex) stores its execution state in user-accessible memory, the system can treat idle threads like any other piece of data; they can be stored on disk and even passed from one domain to another. Similarly, locks, mutexes, condition variables, etc., are ordinary data items in memory, possibly holding pointers to blocked threads. A blocking user-level thread writes no task-specific state into memory; in particular, it does not store the name of a Mach thread port.

5.2 Cross Domain Synchronization

Another important reason we need user-level threads in Opal is for cross-domain synchronization. A key feature of traditional thread packages such as Amber and Presto is their support for fast user-level synchronization between threads in a multithreaded domain. OThreads is unique in that it provides this same user-level synchronization for threads from different domains that share memory. A single lock object can be used to support cross-domain synchronization while continuing to provide fast synchronization among threads when they are in the same domain.

Figure 8 shows an example of two domains that are synchronizing on a condition variable C, in memory shared between them. In the left-hand side of the figure, a thread in domain B blocks in a Wait() on C. This places the thread on a *waiter* list associated with the condition variable. Then, on the right, a thread in domain A signals C, causing the blocked thread to be restarted in domain B. To wakeup the thread, the Signal procedure in A must make a protected RPC call to B's thread scheduler.

For this to work, threads need a *domain-independent* name that can be placed on the *shared* waiter queue when the thread blocks and then used to wakeup the thread when it is signaled. Domain independence is necessary because the thread could be signaled from a *foreign* domain, as in this example. Thread names must also convey enough information and privilege for the cross-domain

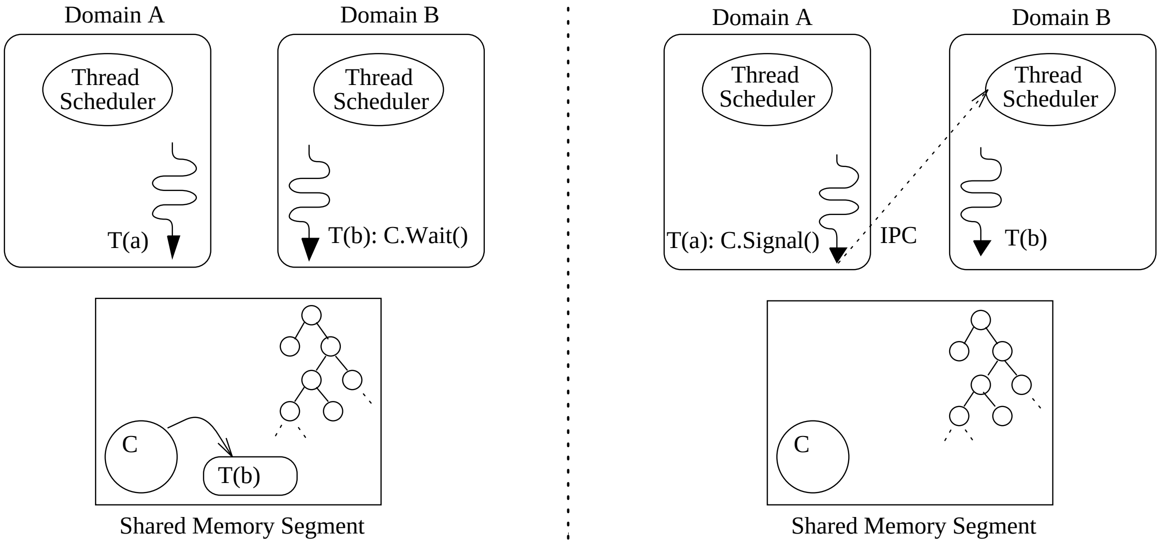

Figure 8: Cross-Domain Synchronization — User-level thread in domain B *waits* on condition variable C, then thread in domain A *signals* it.

wakeup to occur. In OThreads, thread names consist of a combination of the thread's address and its Opal *capability*[3]. Opal makes this possible in two ways: first, the single address space ensures that the thread's address is domain independent; second, permission in Opal is conveyed using capabilities that are domain independent and stateless. To wakeup a thread, a check is made to see if the thread is local; if it is, the thread's address is used to wakeup the thread with a local procedure call; otherwise, an RPC call is made to the thread's domain, as shown in Figure 8.

Operation	*Latency (μsec)*
local mutex	36
cross-domain mutex, one domain	63
cross-domain mutex, two domains	870

Table 2: Mutex *Ping-Ping* Performance

Table 2 shows the performance of a single iteration of a ping-pong test between two threads. Shown is the time to acquire and release a mutex lock, and signal and wait on a condition variable. The *local mutex* time of 36 μsec is the time for two threads in the same domain to synchronize using purely local synchronization, similar to that found in Amber or Presto. The next two times show the time for synchronizing on an OThreads cross-domain mutex. For two threads that are in the same domain, synchronization latency is 63 μsec while it is 870 μsec if the threads are in different domains. This shows that local synchronization can still be fast while supporting cross-domain synchronization with the same lock object.

[3]These pointers are virtual addresses concatenated with a hard-to-guess key.

5.3 Mutual Exclusion on a Uniprocessor

Another useful role for user-level threads is to provide mutual exclusion in a multithreaded domain on a uniprocessor. On a uniprocessor, the user-level thread scheduler ensures that there is never more than one kernel thread running user-level threads in a domain. When a message is received, a kernel thread unblocks just long enough to place a user-level thread for the incoming RPC on the ready queue. This provides a simple way to implement a multithreaded server — common to the Opal environment — without needing any locks in the server to guarantee mutual exclusion. This can have a positive effect on performance on architectures, like the MIPS R3000, that lack hardware synchronization instructions [Bershad et al. 92].

6 Conclusion

We built a new user-level thread package called OThreads that supports IPC and blocking for an operating system with traditional kernel threads, e.g., OSF/1, Mach or Windows NT. We are motivated by the need to provide user-level thread support in an environment of cooperating domains (tasks) where, in effect, each domain is an RPC server and where communication between domains is common. Servers are initially idle and have no processors allocated to them; clients call servers using RPC that, when possible, donates a processor from client to server for the duration of a call.

This work demonstrates that, in the absence of a heavy multiprogramming workload, the two-level scheduling problems associated with IPC and anticipated blocking from a user-level thread can be solved at the user-level without the need for kernel changes such as Anderson's scheduler activations. Our approach is to have each domain maintain a local invariant on the maximum number of kernel threads that are running user-level threads in a domain. This maximum can be coordinated across a node so that the total of these maximums is less than or equal to the number of processors on the node (this is not part of the current implementation). The thread package maintains this invariant by creating and blocking kernel threads as needed in response to events that change the amount of user-level parallelism in a domain, or the number of kernel threads running in the domain or both.

References

[Anderson et al. 91] Anderson, T. E., Bershad, B. N., Lazowska, E. D., and Levy, H. M. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles*, pages 95–109. ACM SIGOPS, October 1991.

[Bershad et al. 88] Bershad, B. N., Lazowska, E. D., and Levy, H. M. Presto: A system for object-oriented parallel programming. *Software — Practice and Experience*, 18(8):713–732, August 1988.

[Bershad et al. 89] Bershad, B. N., Anderson, T. E., Lazowska, E. D., and Levy, H. M. Lightweight remote procedure call. In *Proceedings of the Twelfth ACM Symposium on Operating System Principles*, pages 102–113. ACM SIGOPS, December 1989.

[Bershad et al. 92] Bershad, B. N., Redell, D. D., and Ellis, J. R. Fast mutual exclusion for uniprocessors. In *Proceedings of the Fifth Conference on Architectural Support for Programming Languages and Operating Systems*, pages 223–233. ACM SIGPLAN, September 1992.

[Chase et al. 89] Chase, J. S., Amador, F. G., Lazowska, E. D., Levy, H. M., and Littlefield, R. J. The Amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 147–158, December 1989.

[Chase et al. 92a] Chase, J. S., Levy, H. M., Baker-Harvey, M., and Lazowska, E. D. How to use a 64-bit virtual address space. Technical Report 92-03-02, University of Washington, Department of Computer Science and Engineering, March 1992. Shortened version published as *Opal: A Single Address Space System for 64-Bit Architectures*, Third IEEE Workshop on Workstation Operating Systems (WWOS-III), April 1992.

[Chase et al. 92b] Chase, J. S., Levy, H. M., Lazowska, E. D., and Baker-Harvey, M. Lightweight shared objects in a 64-bit operating system. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 1992. University of Washington CSE Technical Report 92-03-09.

[Cooper & Draves 88] Cooper, E. C. and Draves, R. P. C threads. Technical Report CMU-CS-88-154, Department of Computer Science, Carnegie-Mellon University, June 1988.

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

[Draves 90] Draves, R. A revised ipc interface. In *USENIX Workshop Proceedings, MACH*, pages 101–121, October 1990.

[Feeley et al. 91] Feeley, M. J., Bershad, B. N., Chase, J. S., and Levy, H. M. Dynamic node reconfiguration in a parallel-distributed environment. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 114–121, July 1991.

[Felten & McNamee 92a] Felten, E. W. and McNamee, D. Improving the performance of message-passing applications by multithreading. In *Proceedings of the Scalable High Performance Computing Conference SHPCC-92*, pages 84–89, April 1992.

[Felten & McNamee 92b] Felten, E. W. and McNamee, D. Newthreads 2.0 user's guide. Technical report, Department of Computer Science and Engineering, University of Washington, August 1992.

[Marsh et al. 91] Marsh, B. D., Scott, M. L., LeBlanc, T. J., and Markatos, E. P. First-class user-level threads. In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles*, pages 110–121. ACM SIGOPS, October 1991.

[Tucker & Gupta 89] Tucker, A. and Gupta, A. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *Proceedings of the Twelfth ACM Symposium on Operating System Principles*, pages 159–166. ACM SIGOPS, December 1989.

[Zahorjan et al. 91] Zahorjan, J., Lazowska, E. D., and Eager, D. L. The effect of scheduling discipline on spin overhead in shared memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):180–198, April 1991.