Scheduler Activations on BSD: Sharing Thread Management Between Kernel and Application*

Christopher Small and Margo Seltzer
Harvard University
{chris,margo}@eecs.harvard.edu

Abstract

There are two commonly used thread models: kernel level threads and user level threads. Kernel level threads suffer from the cost of frequent user-kernel domain crossings and fixed kernel scheduling priorities. User level threads are not integrated with the kernel, blocking all threads whenever one thread is blocked. The Scheduler Activations model, proposed by Anderson et al. [ANDE91], combines kernel CPU allocation decisions with application control over thread scheduling. This paper discusses the performance characteristics of an implementation of Scheduler Activations for a uniprocessor BSD system, and proposes an analytic model for determining the class of applications that benefit from its use. Our implementation required fewer than two hundred lines of kernel code and provides an order of magnitude performance improvement over process-level facilities.

1 Introduction

Processes are heavy-weight and, normally, do not share resources. Both communication and switching between processes is expensive. In addition, applications have no control over how processes are scheduled. As an alternative, threads are a lighter-weight abstraction; multiple threads share an address space and its resources, and communication can be accomplished through shared data. The two common styles of implementation for threads parallel the differences between processes and threads: kernel level threads are, effectively, processes that share code and data space, where user level threads are implemented at the application level, multiplexing several threads of control within a single process, with no support from the kernel.

Kernel level threads share some of the disadvantages of processes. Switching between them is slow, taking an order of magnitude more time than a user level thread context switch. Also, they are scheduled by the kernel, with no application control. This can negatively affect performance. For example, if threads have different priorities, and the priorities are not visible to the kernel, a low-priority thread may be scheduled in place of a high-priority one. User level threads systems control scheduling decisions, but because they are not integrated with the kernel, when one thread blocks (e.g. to perform I/O), all of the user level threads sharing the process are blocked.

Anderson et al. [ANDE91] proposed a third model, *Scheduler Activations.* This model divides responsibility for thread management between the kernel and the application. Multiple *virtual processors* are created by the application, and the kernel schedules the virtual processors on the physical processor or processors of the machine. The application creates and schedules threads, assigning them to virtual processors as its sees fit. If a thread blocks (e.g. to perform I/O), the virtual processor running the thread blocks, but another virtual processor can be scheduled in its place on the physical processor. When the first virtual processor becomes ready to run (e.g. the I/O completes), instead of restarting the processor the kernel notifies the application-level thread scheduler that the thread can be scheduled. Assignment of threads to virtual processors is done entirely by the application, at the user level.

Because the user level controls thread scheduling, switching between threads can be accomplished without kernel intervention. The cost of switching between two threads at user level can be close to that of a procedure call; once the next thread is known, thread yield is accomplished by saving the registers of the current thread, changing the stack pointer, and restoring the state of the next thread. In addition, synchronization can be accomplished at user level; if the hardware provides an atomic test-and-set, uncontested locks can be acquired with a handful of instructions.

The Scheduler Activations model is intuitively appealing. The implementation described in the literature shows that its performance characteristics meet the author's goals. However, because no implementation existed for widely available hardware and operating systems, it was not possible to directly take advantage of the original research. Additionally, the existing implementations run

*This research is supported by the Sloan Foundation.

only on multiprocessors; it was not clear that the model was appropriate for a uniprocessor.

This work describes an implementation of Scheduler Activations for commonly available uniprocessor hardware (x86-based "PC"-class computers) and system software (BSD/386[1] v1.0 [BSDI93]). The implementation required about 150 lines of new kernel code, and allows for low-cost management of threads at the user level.

2 Related Work

Anderson et al.'s implementation of the Scheduler Activations model [ANDE91] was for the Taos operating system running on the DEC Firefly multiprocessor. Because the Firefly was not widely available, and Taos (written in Modula-2+) does not run on conventional hardware, the original implementation was not widely used. Barton-Davis et al. [BART92] have implemented Scheduler Activations for Mach 3.0 [ACET86] running on a Sequent multiprocessor. Although Sequent hardware is available commercially (as is Mach), it is certainly not common; the goal of this work is to implement and analyze scheduler activations on widely available hardware and software platforms.

A first step towards kernel level thread support was proposed by Aral et al. [ARAL89]. The `sfork` system call allowed a process to control the degree to which it shared resources with its children. Silicon Graphics' IRIX operating system and Plan 9 from Bell Labs [PIKE90] follow this model, by supplying an `sfork`-like facility. Along those lines, the 4.4BSD-Lite kernel release [CSRG94] includes structural changes that simplify the implementation of an `sfork`, although the call itself is not supplied. The presence of `sfork` is not sufficient to support Scheduler Activations. While `sfork` allows applications to control the level of sharing between processes, it does not provide for the application level scheduling needed for Scheduler Activations, and the context/thread switch time is quite high. Our work uses `sfork` as a tool for building Scheduler Activations.

Once kernel support for Scheduler Activations is provided the last component required is a thread library. A standard interface for thread support is being developed by POSIX [POSIX93]. There are several freely available user-level threads packages that are based on the draft POSIX standard, e.g. the Pthreads package [MUEL93] for SunOS, and Provenzano's portable implementation [PROV94]. Provenzano's package tries to work around the problem of a system call blocking all threads in the process It includes wrappers around system calls that block which, in many cases, can convert a blocking call into a non-blocking equivalent. In this way only the thread that issues the blocking call is suspended. Some operating systems (e.g. OSF/1) provide thread support, based on the POSIX draft standard, in the kernel.

3 Implementation

An implementation of the Scheduler Activations model requires both kernel and user level support. We implemented the kernel support in BSD/386 Version 1.0 [BSDI93], a derivative of the Net/2 BSD release [43BSD]. We also implemented a simple user level package, Myth, which works with the kernel support to provide the functionality of Scheduler Activations.

The user level package is a simple test vehicle for the model, and does not match the draft POSIX specification. The kernel changes are not tied to a particular user level library package; any user level package written to the simple interface exported by our kernel will work just as well.

Because this system is designed to work on a uniprocessor, some of the features of Anderson et al.'s original implementation are not duplicated. For example, there is no mechanism for the kernel to recall a virtual processor once it has been allocated by the application (as there is only one physical processor, the number of virtual processors allocated to the application is unrelated to the number of physical processors allocated to it). Because virtual processors are never recalled by the kernel, we do not need to worry about the application being forced to deschedule a thread that is in a critical section. We therefore saw no reason to implement the critical section cleanup code found in Anderson et al.'s implementation (section 3.3 of [ANDE91]).

3.1 Kernel-Library Interface

Our stock BSD/386 kernel was augmented with a prototype `sfork` call, supplied by Mike Karels of BSDI. The `sfork` call allows a process to create a child process that shares its resources, but does not allow user level control of process scheduling, or any notification of process blocking and unblocking. We added new a system call (`setsgrp`) to allow a process to register itself as being part of a "scheduler activation group". In addition, `msleep` and `mwakeup` are new system calls that support the Myth locking mechanism.

There is no "upcall" primitive in BSD. Rather than design and implement an "upcall", we reused the existing *signal delivery mechanism.* When a signal is posted to a process, the current execution state of the process is saved by the kernel, and a user-designated *signal handler* is run. When the signal handler returns, the normal flow of execution is resumed.

Two signals are used by the kernel-library interface.

[1] BSD/386 and BSD/OS are trademarks of Berkeley System Design, Inc.

The first signal, `blocked`, is sent to notify the user level threads package when a thread blocks in the kernel. The second signal, `ready`, is sent when a thread that had been blocked in the kernel is ready to run. (Instead of adding two new signals, we reused two existing signals, `SIGUSR1` for `blocked`, and `SIGUSR2` for `ready`.)

At initialization time, a Myth application `sfork`s some number of "spare" virtual processors, which sleep, waiting for work. Each virtual processor enters the kernel via the `sigsuspend` call; the kernel puts the virtual processor to sleep until a signal is received.

The kernel needs to determine when a virtual processor is about to block or becomes unblocked. In BSD/386, there is a single point through which processes pass when they block, the kernel function `tsleep`. Code was added in `tsleep` that determined whether a process that is about to block is a virtual processor (i.e., it is a member of a "scheduler activations group"). If so, the kernel looks for a sleeping virtual processor in the same group to take the place of the blocking virtual processor, and sends it `blocked`. The sleeping virtual processor wakes up and yields to a waiting thread.

When the blocked virtual processor wakes up, rather than allowing it to resume immediately, the kernel sends it `ready`. The signal handler manipulates the saved execution state of the virtual processor; it saves the state of the now runnable thread and replaces the execution state with that of a thread chosen by the user level scheduler. This effectively forces the original thread to yield to the chosen thread. When the signal handler returns, the kernel restarts the virtual processor, running in the context of the chosen thread. The user level scheduler can give the virtual processor to the idle loop, causing it to sleep in the kernel, or choose a waiting worker thread, if it wishes to increase the application's degree of parallelism.

3.2 Library-Application Interface

The Myth library offers the services of a minimal threads package. It includes calls to fork a child thread, join with (wait for the termination of) a child thread, and yield the virtual processor to another thread. Calls to create and manipulate user level locks are also provided.

The library is implemented using the kernel-library interface described in section 3.1. At initialization, an application specifies its desired maximum level of parallelism, mp (the maximum number of virtual processors). The library `sfork`s $mp-1$ virtual processors, which suspend themselves waiting for work. As new threads are created, the virtual processors are woken by the user level scheduler (by sending them a `blocked` signal), whereupon they awake and yield to a waiting thread.

The user level scheduler selects waiting threads round-robin; no thread priority scheme is implemented. The Myth library could be extended to support more complex scheduling algorithms.

The threads library is not POSIX-compatible, although there are no major obstacles to a POSIX-compatible library being built on top of the provided kernel-library interface.

The following calls are used to create and manage threads:

- *th_init(mp):* initialize the threads package and create $mp-1$ additional virtual processors.

- *th_fork(func, arg):* fork a new child thread. The child is started, running *func* with argument *arg*[2]. A thread structure and stack are allocated for the new thread, and the stack is initialized so that when the thread is scheduled the function will be called. The thread structure is then added to the pool of ready threads.

- *th_yield():* yield the virtual processor to another thread. The state (register set) of the current thread is saved in the thread structure, and the state of the next thread is loaded. If no other thread is ready to run, the current thread is allowed to immediately continue.

- *th_join(thread):* wait for the termination of the specified thread, and accept its return value. The current thread is marked as blocked, and the thread structure of the thread being joined with is marked with the identity of the blocked thread. When the joined thread terminates, it marks the joining thread as runnable.

3.3 Locking

Along with the thread creation and management calls described above, the Myth user level library supplies synchronization primitives. Because threads share an address space, synchronization can be done using shared variables. If the underlying hardware supports an atomic test-and-set instruction (as the x86 does), an uncontested lock can be acquired in a handful of instructions. Only when a requested lock is held by another thread is it necessary for the thread to block in the kernel and wait for the lock to become available.

A lock variable holds either zero (if unlocked), or the id of the last thread to attempt to acquire the lock. To acquire a lock, an application uses the x86 atomic test-and-set instruction (`xchg`), setting the value of the lock to its thread id. If the lock is currently held (the previous value is not zero), the process enters the kernel, sleeping on the location of the lock.

[2] Since the threads are running in the same address space, multiple arguments can be collected in a `struct` in the heap, and a pointer to the `struct` can be passed as the argument.

When a thread releases a lock, it determines if any thread has attempted to acquire the lock (if the current value of the lock is not its thread id). If so, the process enters the kernel and wakes all virtual processors sleeping on the lock.

There is a possible race condition in the above scenario. Assume that thread A holds a lock and thread B wants to obtain it. If thread B is descheduled after it checks the lock but before entering the kernel, thread A may release the lock before thread B sleeps. If this happens, thread B will sleep after thread A has broadcast the wakeup, and B will never awake. To prevent this from happening, the kernel tests to see if the lock has become available (its value is now zero) before suspending the virtual processor. If the lock is available, the kernel returns without sleeping, and the process tries again to obtain the lock.

If the lock is uncontested, the lock can be acquired in a handful of instructions. If the lock is contested, the overhead of testing it at user level is small relative to the cost of entering and blocking in the kernel. The code in Figure 1 implements the user-level portion of the hybrid user-kernel locking scheme.

```
th_mutex_lock(mutex *m)
{
    /* obtain lock */
    while (xchg(m, my_thread_id()) != 0)
        msleep(m);
}

th_mutex_unlock(mutex *m)
{
    if (xchg(m, 0) != my_thread_id())
        mwakeup(m)
}
```

Figure 1: User level locking code

3.4 System Call and Library Support

This work concentrates on providing a mechanism for sharing thread scheduling and context switching duties between the kernel and user level, but support for thread context switching is not sufficient to build multi-threaded applications. For example, several of the calls found in the C library assume a single thread of control, and do not synchronize access to data (e.g. `printf(3)`). A production-quality threads package must include calls that use synchronization primitives to protect shared data.

Some system calls can not be fixed this way. For example, using standard system calls in a multi-threaded application, there is no way to to ensure that `lseek(fd, 0, SEEK_SET)` followed by `read(fd, buf, bufsize)` will read the data found at the beginning of the file; the thread may be descheduled between the `lseek` and the `read`, and a different thread may change the file pointer. Applications must be written to work around these limitations, or the system call interface must be expanded to include new, stateless calls.

The common system and library calls that are unsafe in a multi-threaded application are:

- calls that use buffered I/O: `printf`, `fprintf`, `scanf`, `fscanf`, `gets`, `fgets`, `puts`, `fputs`, `fread`, and `fwrite`.
- `lseek` combined with `read` or `write`.
- `malloc` and `free`, which manipulate the global heap.
- calls which use static data, e.g. `strtok`, `getpwnam`, and much of the X11 library.

3.5 Code Sizes

About 150 lines of C code (including comments) were added to the BSD/386 kernel, in addition to the changes to support `sfork`. The modifications to `tsleep` that control signaling the user level threads package required about sixty lines. Small changes (approximately twenty lines) were needed in the code that posts signals, so that sufficient information is passed to the user level handler to save and restore the state of the thread. The `msleep`, `mwakeup`, and `setsgrp` system calls are each less than ten lines long.

The user level threads package consists of about 800 lines of machine-independent C code, 800 lines of machine-dependent code (e.g. that is aware of x86 calling conventions and manipulates the stack), and 90 x86 assembler instructions (implementing low-level setup and thread switching). When compiled, the library takes up approximately 10KB of code and data space. (For comparison, on the test platform, the `''hello world''` program is 28KB.)

4 Performance Analysis

In order to determine when it is worthwhile to architect an application as a multi-threaded system, the costs and benefits need to be weighed. The costs include time spent initializing the system, fork and join time, context switch overhead, and per-operation synchronization time. The benefit is the possibility of increased throughput, by overlapping computation with synchronous I/O.

The functional benefit of a thread system is the ability to parallelize an application. The performance benefit is limited by the level of parallelism possible, given the application, and by the overhead imposed by the system. Intuitively, the overhead of user level threads is substantially lower than that of kernel level threads, but an operation-by-operation comparison needs to be made. On top of the primitive operations, there is initialization and upcall overhead associated with using Scheduler Activations. Along with the cost of an upcall, the frequency of upcalls needs to be determined.

To determine which applications would benefit from using our system, we measured the per-operation cost of each Myth primitive. We then constructed an analytic model of the system, which can be used to determine whether a particular application would benefit from using our system.

We ran all tests on a "PC"-class 486DX2-66 running in single-user mode. The CPU has an 8KB on-chip level one cache, and the hardware includes 256KB off-chip level two cache (20ns static RAM), and 8MB of main memory. The system was configured with an 800KB disk cache. All microbenchmarks fit into main memory; no paging was observed.

BSD/386 provides two clocks for measuring performance, the time-of-day clock and the resource usage timer. The resolution of the time-of-day clock is on the order of one microsecond (838 nanoseconds), as compared with the 10 millisecond granularity of the resource usage timer. Although the resource usage timer gives separate values for system time and user time, because of the four orders of magnitude difference in granularity, it is of limited utility for timing short-duration operations. Instead, the time-of-day timer was used, and our times are reported as elapsed times in microseconds.

Some thread operations are of very short duration relative to our clock granularity. Where this was the case, each test run measured the elapsed time required to perform 100,000 operations.

4.1 Locking and Unlocking

When moving to a multi-threaded system, synchronization performance is critical. Performance tests of the multiprocessing LIBTP transaction processing package showed that 25% of the total time spent updating database meta-data was accounted for by kernel locking calls [SELT92]. Our microbenchmarks (in Table 1) show that an uncontested lock can be acquired at user level in less than a tenth the time required to acquire a lock via a kernel call.

The locking microbenchmark repeatedly acquires and releases a single lock. The loop is run 100,000 times. The time required for each lock/unlock pair ("Myth" in Table 1) was measured at 4.1μs. Because the time spent locking and unlocking is so small, function call overhead has a measurable impact on the results. We measured the time to call a function of one argument ("nullfunc(arg)" in Table 2), and found it to be 0.3μs. The two calls (one each to lock and unlock) in the inner loop of the Myth Lock/Unlock test take up 0.6μs, or 15% of the total time of 4.1μs.

The C code executed by the lock and unlock functions compiles to a total of 81 x86 instructions. By recoding the bodies of the functions in assembler ("Myth-asm"), the time can be reduced to 48 instructions and 1.8μs (including the 0.6μs overhead for the two calls).

BSD/386 does not export locking primitives, although the file system provides a facility for locking byte ranges of files, `fcntl`. As implemented, this facility is not particularly efficient; it uses a linked list of locked byte ranges, sorted by start address. Depending on whether locks are obtained in increasing or decreasing order, time to acquire and release locks can be constant or quadratic with the number of locks. Other structures (e.g. interval trees [CORM90]) would be much more efficient. To fairly compare the overhead of the locking code, and not the performance of the internal data structures, our test repeatedly acquired and released a single lock (the first byte of the file) ("BSD-file"). Each test run consisted of locking and releasing the lock 100,000 times. Acquiring a file lock took 88.5μs, nearly twenty times as long as it took to acquire a lock at user level.

It is not entirely fair to compare user-level locks with kernel-level file locks. We implemented a simple kernel-based locking service, roughly equivalent to the kernel portion of the hybrid locking scheme ("Kernel"). Each run consisted of locking and releasing the lock 100,000 times. The results show that a simple kernel lock/unlock takes about half as long as a file lock/unlock, but still eleven times as long as a user-level lock/unlock.

Of the 45.5μs to acquire and release a lock, we can attribute 13.8μs to system call overhead (a null system call takes 6.9μs, "null syscall" of Table 2). Most of the remaining 31.7μs is taken up in the unlock call. We found that unlock spends most of its time in the kernel's `wakeup` function, which is used to wake any processes sleeping on the lock. The body of `wakeup` disables interrupts (`splhigh()`), searches for processes blocked on the lock, and re-enables interrupts (`splx()`). We measured the cost of disabling and re-enabling interrupts ("splhigh/splx"), and found that on our test system it takes 24.5μs, or roughly 78% of the time spent in the kernel locking code.

Without this overhead, the lock/unlock time for a simple kernel lock would be roughly 20μs, or six times that of user level locks (and 13.8μs of this cost is system call overhead.) If the kernel unlock code could determine that there were no processes waiting for the lock before calling `wakeup` (which we expect to be the normal case), the interrupt level would not need to be changed, and the lower cost could be achieved.

Test		*num runs*	*time per lock/unlock*	*std dev*
Lock/Unlock	Myth	10	4.1μs	0.04%
	Myth-asm	10	1.8μs	0.11%
	BSD-file	10	88.5μs	0.08%
	Kernel	10	45.5μs	0.04%

Comparison of user level locking with kernel level locking. For uncontested locks, user level locking is at least 20 times faster. *Myth*: time to lock/unlock an uncontested lock at user level. *Myth-asm*: time using a hand-tuned version of the same code. *BSD-file*: time to lock/unlock a byte of a file using `fcntl`. *Kernel*: simple kernel-based locking scheme.

Table 1: Microbenchmark Results – Lock/Unlock

Test	*num runs*	*time per operation*	*std dev*
nullfunc(arg)	10	0.3μs	0.64%
null syscall	10	6.9μs	0.17%
splhigh/splx	10	24.5μs	0.08%

Components of overhead associated with user level and kernel level locking. More than half of kernel level locking time is spent disabling and re-enabling interrupts. *nullfunc*: time to call a null function of one argument. *null syscall*: time to perform a null system call. *splhigh/splx*: time required to disable and re-enable interrupts on the test platform.

Table 2: Microbenchmark Results – Locking Overhead

4.2 Fork/Join

The overhead of forking a new thread, and later joining with it, is important to some types of multi-threaded applications. For example, a graphical application may fork a new thread for each input event. In cases like this, the cost of forking a child thread (and later joining with it) needs to be fairly low.

In our system, the majority of the work required to fork a new thread is performed in the parent thread. The parent allocates the new thread structure for the child, allocates and initializes a stack for the child, and links the child into the list of threads. It then calls the `th_new_work()` function so that if there are any sleeping virtual processors, one is woken to service the new thread. When the child thread is scheduled, a small (five instruction) thread root routine is run that invokes the child thread's function.

The Myth fork micro benchmark forks a child thread and then joins with the child. The child immediately returns. This loop is run 100,000 times.

The time required to fork a thread, 102.8μs (Table 3), seems high for the amount of work that is being done. Profiling showed that there are 13 lock/unlock operation pairs for each fork. At a cost of 4.1μs each, the locking overhead for a fork is 53.3μs, or about half of the total time spent. Further investigation showed that during a thread fork, thirty calls are made to `th_self`, the function that returns a pointer to the current thread's structure. (Twenty-six of these calls are made by the thirteen lock/unlock calls). Although `th_self` takes very little time itself, (0.79μs), the thirty calls per fork add up to 23.7μs. Inlining `th_self` would reduce the cost to approximately 0.51μs per call, reducing the thread fork time from 103μs to roughly 96μs .

For comparison, we measured the cost of a native `fork/wait`, as well as a `sfork/wait` pair. Each test run looped 1,000 times. The `sfork` call, which reuses parent resources, takes twenty times as long as a thread fork. The `fork` call, which duplicates parent resources, takes about four times as long as `sfork`. Most of the structures that `fork` copies (e.g. credentials, open file table) are fairly small, but the page tables of the VM space and the process' stack are large, and account for most of that time.

4.3 Yield

Another major advantage of a user level threads system is that a context switch is very light weight. At the root of it, a thread context switch is little more work than a function call: first the yielding thread saves its register state, and then the new thread loads its register state and continues. A process-level context switch is much more expensive, requiring entering the kernel and updating the state of the virtual memory system. The yield microbenchmark forks a child thread and repeatedly yields to the child thread; the child immediately yields to the parent. This loop is run 50,000 times (100,000 yields). We measured the yield time as 7.8μs (Table 4). Included in the cost of the yield are one lock/unlock pair (4.1μs) and a call to `th_self` (0.79μs); the remaining time, 2.9μs, is taken up by roughly 100 machine instructions.

There is no explicit yield function provided by BSD/386. We obtained process context switch times us-

Test		*num runs*	*time per fork*	*std dev*
Fork	Myth	10	102.8μs	0.39%
	BSD-sfork	10	2076μs	0.27%
	BSD-fork	10	8470μs	0.11%
th_self		10	0.79μs	0.32%

Comparison of forking a thread with forking a process. Thread fork is two orders of magnitude faster than process fork. *Myth:* time to fork a new thread. *BSD-sfork*: time to `sfork` a child process that shares all resources with the parent. *BSD-fork*: time to `fork` a new process. *th_self*: cost of looking up a pointer to the thread's descriptor structure.

Table 3: Microbenchmark Results – Fork

Test		*num runs*	*time per yield*	*std dev*
Thread yield		10	7.8μs	0.02%
Context switch	2 process	60	105.7μs	6.17%
	4 process	60	131.6μs	6.16%
	8 process	60	148.2μs	5.75%
	16 process	60	158.1μs	9.22%
	20 process	60	162.9μs	9.96%

Comparison of thread yield (context switch) with process context switch. Thread context switch is 15 times faster than process context switch. *Thread yield*: user level thread switch time. *Context switch*: process context switch times, switching between 2, 4, 8, 16, and 20 processes.

Table 4: Microbenchmark Results – Yield

ing Larry McVoy's *lmbench*[3] suite [MCVOY94]. The lmbench suite computes yield times by connecting several processes via pipes in a circle, and passing a token (a single character) around the circle. The cost of transmitting a character through a pipe is computed separately and subtracted from the time required to pass the token. On an unloaded system, the remaining time can attributed to process context switch time. The token is passed 2,000 times between the processes. The lmbench suite tests context switch times for switching between different numbers of processes (2, 4, 8, 16, and 20) and different sized processes (the base 144KB code space of the benchmark, which is shared between the processes, is augmented by a dynamically allocated data space of 4KB, 16KB, 32KB, and 64KB).

There is no single number that describes the context switch time; there is a factor of fourteen difference between the low (two processes, no added data space, 106μs) and the high (twenty processes, 64KB data space, 1186μs). Because virtual processors share code and data space, the best comparison is with the context switch times of processes with no extra data. We measured times for 2, 4, 8, 16, and 20 processes. We (and McVoy[4]) attribute the increased context switch time to an increasing number of L2 cache misses as the amount of `proc` data grows. The L2 cache is only 256KB, which is not large enough to hold all of the active user and kernel code and kernel data when there are a large number of active processes.

We found a much larger timing variation on the process context switch tests than on other ones. We collected data on 60 runs, as compared with the ten runs for the other tests, to attempt to reduce the size of the standard deviation, but were not completely successful.

Virtual processors share a common page table, so it may be possible for the kernel to switch more quickly between two virtual processors in the same scheduler activation group than between two unrelated processes. On the 486, the instruction that sets the page table register is specified as taking four clock cycles[5]; on other architectures, this may take substantially more time.

4.4 Signal Delivery and Upcalls

The native BSD signal delivery mechanism is used to perform "upcalls" from the kernel to a virtual processor. Because the overhead of signal delivery occurs whenever a thread becomes blocked or unblocked, this cost has a significant impact on application performance.

To estimate the cost of the kernel posting a signal to a process, we measured two costs: the time required for an application to enter the kernel, send and handle a signal, and return (126μs, Table 5), and the time required to just enter the kernel's signal code and return (10μs). Subtracting the latter from the former gives 116μs, an es-

[3] `lat_ctx.c,v 1.1 1994/11/18 08:49:48 lm Exp`

[4] personal communication, April 1995

[5] A `movl` into `CR3` sets the page table pointer [INTEL92].

timate of the cost of posting a signal from the kernel and handling it at user level.

The test first registers a signal handler for `blocked`, and then repeatedly sends the signal to itself (using `kill(mypid, blocked)`). This is done 100,000 times, and the per-`kill` call is computed. Then, the test repeatedly sends signal 0 to itself, which enters the signal handling code and performs validation on the process id but does not actually post the signal. This loop is run 100,000 times, and the per-call cost is derived.

The impact of this cost on application performance is discussed in detail in section 5, below.

4.5 Initialization Overhead

Although initialization overhead is a one-time cost, if it is large, it can have a noticeable impact on short-running applications. We measured initialization time, in order to determine if it was reasonable. On our test platform, initialization of the thread system takes 11ms[6], with more than half of the time (6.7ms) spent allocating and initializing thread stack space.

Each thread requires a separate stack on which to run. Allocating and initializing a stack (16KB by default) is fairly costly, but a stack can be reused after a thread has terminated. We need only allocate enough stacks to support the maximum number of threads that run concurrently.

For convenience, Myth allocates stacks in groups of 16 (256KB) and keeps them on a free list. When a thread is forked, a stack is taken from the free list. If the free list is empty, a new group of 16 stacks is allocated and added to the list.

Our decision to allocate 16 stacks at a time is arbitrary. We measured the time required to allocate and initialize from one to 64 stacks, and found that the time is linear in the number of stacks.

5 Analytic Model

On a uniprocessor, there are three reasons to use a threads package.

- First, it may be that the application is most naturally structured using multiple threads, e.g. an application that uses separate threads for processing audio and video streams, or that implements a divide-and-conquer algorithm. The overhead of using the threads package should be low enough so that its cost does not outweigh the benefit of simplifying the structure of the application.
- Second, a machine-independent threads package may be available that runs on both uniprocessors and multiprocessors. The application may be prototyped on a uniprocessor, and run on the multiprocessor only used when the application is complete.
- Third, the application may benefit from increased parallelism, by overlapping computation and I/O. For example, a database server could fork a thread to handle each incoming request; while a pending client request is waiting for the disk, the server is able to accept, and process, new requests.

The first motivation is not well suited to an analytic model; it is a qualitative, rather than quantitative, decision. The second is based on compatibility, which is also not determined by the performance of the system. However, the third is amenable to analysis.

The performance improvement of such an application is limited by the relative I/O and CPU time. As recent research pointed out [SELT95], the greatest possible speedup comes with an application that spends equal time on computation and I/O: we can completely overlap computation with I/O. In general, the minimum time that an application can take to complete is the larger of the amount of time spent computing or performing I/O.

If a single-threaded application waits for I/O operations to complete, and the I/O system is not already saturated, it may be possible to overlap the time spent waiting for the I/O with other computation, by having separate threads wait for I/O and perform computation. If the overhead of using the thread package is less than the added CPU time available to the application because of the increased parallelism, it is worthwhile to use the threads package.

5.1 Cost-Benefit Model

As a concrete example, let us return to the idea of a database server. A server accepts requests for data, and services them either from its in-memory cache or by reading the data from the disk. While the data is being read from the disk, a single-threaded server can not service requests for data from its cache. If the server were to fork a new thread to handle each disk request, it would be able to accept new cache requests before the pending disk request completes.

The relative cost of cache and disk requests, and the cache hit ratio come into play as well. For purposes of exposition, consider the following (somewhat artificial) scenario: assume that it takes nineteen times as long to satisfy a request from the disk as from the cache, and the cache has a 95% hit ratio. The server is then spending an equal amount of time servicing disk and cache requests; the product of the cache hit ratio and the cache service time (cache time) is equal to the product of the cache miss ratio and the disk service time (disk time). By forking a thread to service each disk request, if the disk is currently

[6] Over 10 runs, standard deviation of 1.58%

Test		*num runs*	*time per signal/handle*	*std dev*
Signal	BSD	10	126.7μs	0.1%
	BSD-overhead	10	10.8μs	0.5%
Upcall estimate		—	115.9μs	—

Estimate of upcall time, which is a major cost of using Scheduler Activations. The estimate is the cost of sending a signal minus the overhead of invoking the `kill` system call. *BSD*: cost of sending and handling a signal. *BSD-overhead*: cost of sending signal 0 (which tells the kernel to process the request but not actually post a signal). *Upcall*: cost estimate of an upcall.

Table 5: Microbenchmark Results – Signal

running at no more than 50% capacity, the multi-threaded server can handle twice as many client requests.

If the disk is currently saturated, disk requests will be satisfied no more quickly with a multi-threaded server than with a single-threaded server. However, the disk will only be saturated if no time is being spent in computation; if this is the case, all requests require disk access, and there is no purpose in trying to overlap computation with I/O. Similarly, if the application is CPU bound, it is spending no time waiting for I/O; overlapping I/O with computation will not change the performance of the system.

The speedup is bounded by the ratio of the amount of time spent waiting for I/O to complete to the total time spent; at best, computation can overlap all I/O. From this speedup we must subtract the overhead required to create and manage the multiple threads; any balance is the benefit seen by the application.

The time spent in an application can be divided into three categories:

- *CT (compute time):* time spent computing.
- *ST (synchronous I/O time):* time spent performing synchronous I/O (I/O that must be completed before computation can continue).
- *AT (asynchronous time):* time spent performing I/O that can be overlapped with computation.

The total time spent by a single-threaded application is the sum of these components,

$$CT + ST + AT$$

If we are able to completely overlap asynchronous I/O with computation, the total time is reduced to:

$$max(CT + ST, AT + ST)$$

The maximum possible benefit is:

$$min(AT, CT)$$

From this we need to subtract the cost of using the threads package. Ignoring fixed start-up costs (which can be amortized over an arbitrary running time), we need to include each "upcall" signal at thread startup, block, and unblock (116μs each). If we assume that signals are sent only because of a thread issuing a blocking I/O call, the number of signals sent is a function of the blocking I/O calls. We can derive the cost given:

- *RB (asynchronous I/O requests that block):* the number of asynchronous I/O requests that block (and hence cause a context switch).

If a fixed number of worker threads are forked at application start-up, we can amortize this start-up cost and need only be concerned with the run-time cost of switching between the virtual processors on which they are scheduled. As a thread blocks, a `ready` signal is sent to a sleeping virtual processor, which performs a thread context switch to a ready thread (116μs+7.8μs, or roughly 124μs). When the I/O is complete, another `ready` signal is sent, causing another context switch (124μs). The benefit (cost) is therefore:

$$min(AT, CT) - 2 \times RB \times 124\mu s$$

If threads are forked while the process is running, the cost is higher. If we assume that a new thread is forked to process a pending I/O, the switching overhead increases, because of the protocol used between kernel and user. When the server forks a new worker thread (pictured in Figure 2), the library package wakes an idle virtual processor (by sending it the `blocked` signal), which tells it to look for a ready thread. The virtual processor yields to the ready thread, which issues its I/O and blocks in the kernel. By blocking, it causes the kernel to send a `blocked` signal to another idle virtual processor (if one exists). When the worker thread's I/O is complete, the `ready` signal is sent by the kernel to the worker's virtual processor; the worker is rescheduled, completes its work, and exits. The virtual processor then yields to another thread.

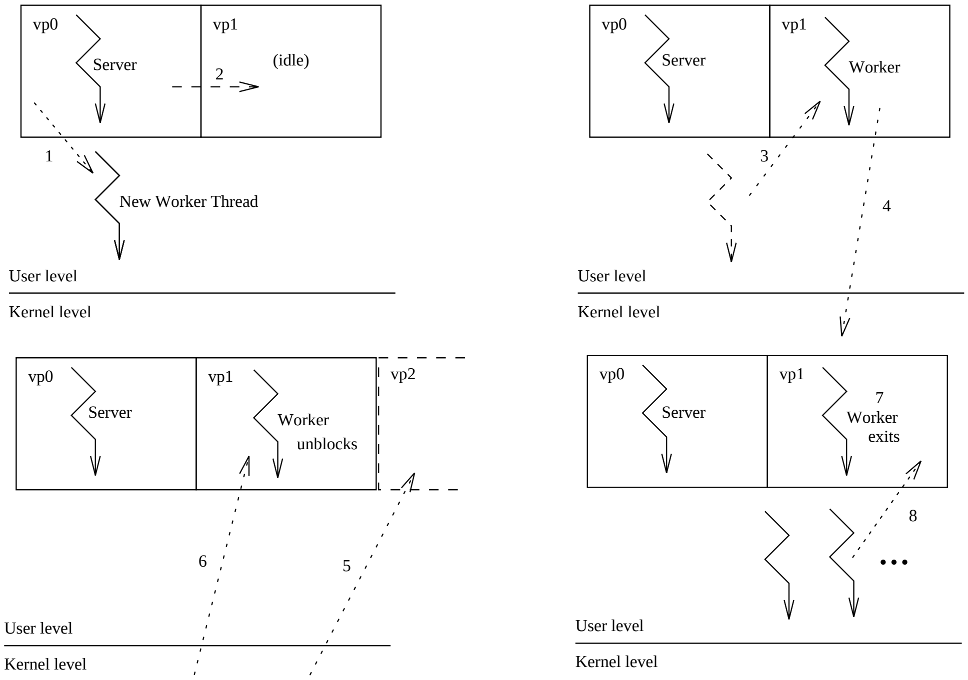
To compute the cost of an application that follows this model, we need to know:

- *NF (number of forks):* the number of threads forked by the application while running.

If we rewrote this type of application using Myth, we would see a time savings of:

$$min(AT, CT) - (RB \times 375\mu s + NF \times 103\mu s)$$

Note that if the child thread issues multiple blocking I/O requests, the thread fork cost can be amortized across the



1. Server forks worker thread (103μs).
2. Server sends signal to idle virtual processor (127μs).
3. Virtual processor yields to worker (8μs).
4. Worker issues I/O, which blocks in kernel.
5. Kernel upcalls to idle virtual processor (116μs).
6. I/O completes, kernel upcalls to worker's virtual processor (116μs).
7. Worker is rescheduled, completes, and exits.
8. Virtual processor yielded to another thread (8μs).

Figure 2: Multi-threaded server starting a thread

I/O requests.

If the time spent performing asynchronous I/O (AT) is smaller than the overhead caused by additional context switches and forks, the multi-threaded application will run more slowly than the single threaded version.

5.2 Database Server

Consider an alternative architecture for our prototypical database server. When it receives a client request that can be satisfied from the cache, it does so. A request that requires disk access is handled by forking a new thread to service the request. If we make some assumptions about the application and system performance we can compute the cost or benefit of rewriting the server using our system.

We will assume that a disk access takes at least 10ms, and that the application is evenly split between time spent servicing these asynchronous requests (AT) and time spent servicing requests from the cache (CT). There is no synchronous I/O, so $ST = 0$. With a 95% hit ratio, a request that hits in the cache takes about 526μs to service. Over the course of one second, we receive fifty disk requests ($RB = 50$) which take 500ms, and 950 cache requests that take 500ms. For each request we fork a new thread ($NF = 50$), so the benefit is:

$$min(AT, CT) - RB \times 375\mu s - NF \times 103\mu s$$

or

$$min(500ms, 500ms) - 50 \times 375\mu s - 50 \times 103\mu s$$

or 476ms. The maximum possible savings, $min(AT, CT)$, is 500ms, so the actual gain is 95% of the possible gain.

Although this application could be implemented using standard system facilities, the benefit is smaller. If the server forked a child using `sfork` instead of `th_fork` the cost per fork is raised from 103μs to 2076μs . However, there would be no requirement to send signals to wake waiting processes, so the cost-benefit equation becomes:

$$min(AT, CT) - NF \times 2076\mu s$$

which, in this case, is

$$min(500ms, 500ms) - 50 \times 2076\mu s$$

or 396ms. The difference is small (80ms), but significant:

it is sufficient to service 152 client requests that hit in the cache.

If a regular `fork` were used, at a per-`fork` cost of 8470μs, the benefit equation is:

$$min(AT, CT) - NF \times 8470\mu s$$

which is

$$min(500ms, 500ms) - 50 \times 8470\mu s$$

or 76ms, only 15% of the maximum possible benefit. The 500ms is almost entirely wiped out by using `fork`.

In some instances, the server may not be able to determine *a priori* whether a call will block or not. For example, the database might be accessed by mapping it into memory through the use of `mmap`, rather than through explicit `read` requests. Any access to the mapped region may result in a disk access (page fault). If this were the case, to be safe the server might fork a new thread to handle each request as it came in. The cost of forking and servicing a non-blocking thread would be the fork cost, the signal cost, and the cost of two yields ($103 + 126 + 2 \times 8 = 245\mu$s). If we continue to assume that the cache hit ratio is 95%, and that each disk request takes 10ms, the cost of forking a thread on each request (5ms for twenty requests) is half the benefit gained by not blocking the application (one 10ms block out of twenty requests).

Locking costs are ignored in this example; depending on the architecture of the server, a substantial portion of its time may be spent in synchronization code. As was mentioned above, performance tests of the LIBTP transaction processing package [SELT92] showed that 25% of the total time spent updating database metadata was accounted for by kernel locking calls. In that study, most lock requests were uncontested. Assuming that the performance of the kernel locks in that study had roughly the performance of the simple kernel locks in this work, user level locks, which take 9% of the time of kernel locks, would decrease this percentage to 2.25% of total time.

5.3 Other Examples

The benefit of being able to overlap computation and I/O is not limited to database servers. For example, a program that computes the checksum of a file can spend a considerable amount of time computing. If the checksum computation can be performed piecewise, the application could fork multiple threads, each working on a separate section of the file. While one thread was waiting for I/O, others could be computing.

Programs that do not perform explicit I/O can also benefit from overlapping I/O and computation through multi-threading. Ray tracing, a computer graphics rendering scheme, involves tracing the path followed by a ray of light through the scene for each pixel on the display. Because a scene description can be quite large, it may end up being paged in and out of physical memory. Each page in is an I/O operation on which the process needs to wait. If a ray-tracing program were decomposed into multiple threads, when one was blocked waiting for a portion of the scene description to be paged in, others could continue to compute on the portions in physical memory.

6 Conclusions

The performance characteristics of the Scheduler Activations thread model is well-suited for uniprocessors. Assuming the existence of an `sfork`-like system call, it is straightforward to implement Scheduler Activations with few changes to the kernel, by reusing existing facilities. The benefit seen matches the expected, intuitive "one to two orders of magnitude".

The division of responsibility between the kernel, which schedules virtual processors, and the application, which multiplexes threads on those virtual processors, allows an application to manage its threads at the user level without paying the overhead of entering the kernel.

When used to allow an application to overlap computation with I/O, the overhead of using Scheduler Activations is less than 10% of the cost of an I/O operation, and the remainder of the I/O time can be overlapped with computation.

7 Status and Availability

The system runs on BSD/386 version 1.0, which is not a current release of that system. A port to the current release, BSD/OS 2.0, is underway. The `sfork` changes have been ported to release 2.0 by BSDI, but are not part of the official release. Our kernel changes were applied to files that are not BSDI-proprietary, and are publicly available, as is our user-level library code and our test suite. Given a 4.4BSD-based kernel, and `sfork`, integration of our changes should be fairly easy.

In the current release, floating-point state is not saved and restored when a thread yields. Saving and restoring state is fairly expensive, and, in fact, is not done by BSD/386 unless the floating-point unit was used since the previous context switch. Implementing floating-point save and restore is not complicated. Addition of the code would not affect either the Myth or BSD/386 microbenchmarks, as neither perform floating-point operations.

References

[ACET86] Acetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., Young, M., "Mach: a New Kernel Foundation for UNIX Development", *1986 Summer USENIX Conference* (July 1986).

[ANDE91] Anderson, T., Bershad, B., Lazowska, E., Levy, H., "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism", *Proceedings of the 13th SOSP*, pp. 95–109 (1991).

[ARAL89] Aral, Z., Bloom, J., Doeppner, T., Gertner, I., Langerman, A., Shaffer, G., "Variable Weight Processes with Flexible Shared Resources", *1989 Winter USENIX Conference*, pp. 405-412 (January 1989).

[BART92] Barton-Davis, P., McNamee, D., Vaswani, R., Lazowska, E., "Adding Scheduler Activations to Mach 3.0", University of Washington TR 92-08-03 (August 1992).

[BSDI93] *BSD/386 Version 1.0*, Berkeley Software Design, Inc. (1993).

[CORM90] Cormen, T., Leiserson. C., Rivest, R., *Introduction to Algorithms*, MIT Press, Cambridge, MA, pp. 290-295 (1990).

[CSRG94] Computer Systems Research Group, University of California at Berkeley, *4.4BSD-Lite Source CD-ROM*, USENIX Association and O'Reilly & Associates (April 1994).

[INTEL92] *Intel486 TM Microprocessor Family Programmer's Reference Manual*, Intel Corporation, Mt. Prospect, IL (1992).

[43BSD] Leffler, S., McKusick, M. K, Karels, M., Quarterman, J., *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley, Reading, MA (1989).

[MCVOY94] McVoy, L., "lmbench: Portable Tools for Performance Analysis", unpublished manuscript, posted to `comp.benchmarks` (1994).

[MUEL93] Mueller, F., "A Library Implementation of POSIX Threads under UNIX", *1993 Winter USENIX Conference*, pp. 29–41 (January 1993).

[PIKE90] Pike, R., Presotto, D., Thompson, K., Trickey, H., "Plan 9 from Bell Labs" *UKUUG Proc. of the Summer 1990 Conference*, London, England, pp. 1–9 (July 1990).

[POSIX93] *P1003.4a, draft 7: POSIX System Application Program Interface Amendment 2: Threads Extension (C Language)*, IEEE, New York, NY (April 1993).

[PROV94] The latest release of Provenzano's package is available via anonymous ftp from `sipb.mit.edu` in `/pub/pthreads`.

[SELT92] Seltzer, M., Olson, M, "LIBTP: Portable, Modular Transactions for UNIX", *1992 Winter USENIX Conference*, pp. 9–25 (January 1992).

[SELT95] Seltzer, M., Small, C., Smith, K., "The Case for Extensible Operating Systems", Harvard University Center for Research in Computing Technology TR-16-95 (July 1995).