An Implementation of Scheduler Activations on the NetBSD Operating System

Nathan J. Williams
Wasabi Systems, Inc.
nathanw@wasabisystems.com

Abstract

This paper presents the design and implementation of a two-level thread scheduling system on NetBSD. This system provides a foundation for efficient and flexible threads on both uniprocessor and multiprocessor machines. The work is based on the scheduler activations kernel interface proposed by Anderson et al. [1] for user-level control of parallelism in the presence of multiprogramming and multiprocessing.

1 Introduction

Thread programming has become a popular and important part of application development. Some programs want to improve performance by exploiting concurrency; others find threads a natural way to decompose their application structure. However, the two major types of thread implementation available – user threads and kernel threads – both have significant drawbacks in overhead and concurrency that can limit the performance of applications.

A potential solution to this problem exists in the form of a hybrid, two-level thread system known as *scheduler activations* that divides the work between the kernel and the user levels. Such a system has the potential to achieve the high performance of user threads while retaining the concurrency of kernel threads.

The purpose of this paper is to describe the design and implementation of such a two-level system and associated thread library in NetBSD, show that the speed of thread operations is competitive with user thread implementations, and demonstrate that it can be implemented without hurting the performance of unrelated parts of the system.

First, as motivation, Section 2 describes traditional thread implementations. Scheduler activations are explained in the context of two-level thread implementations, and then described in detail. Section 4 gives the interface that the scheduler activations system presents to programs, and Section 5 follows with details about the kernel implementation behind the interface. The thread library built on this interface is described in Section 6, and the performance of the system and the thread library is examined and compared to other libraries in Section 7. Finally, Section 8 concludes and considers directions for future work.

2 Thread Systems

Historically, there have been two major types of thread implementations on Unix-like systems, with the essential differences being the participation of the kernel in the thread management. Each type has significant drawbacks, and much work has gone into finding a compromise or third way.

The first type of thread system is implemented purely at user-level. In this system, all thread operations manipulate state that is private to the process, and the kernel is unaware of the presence of the threads. This type of thread system is often known as "N:1" thread system because the thread implementation maps all of the N application threads onto a single kernel resource.

Examples of this type of thread system include the GNU PTH thread library [6], FSU Pthreads, PTL2 [7], Provenzano's "MIT pthreads" library [8], and the original DCE threads package that formed so much of the basis for the POSIX thread effort [4]. Some large software packages contain their own thread system, especially those that were originally written to support platforms without native thread support (such as the "green threads" in Sun's original Java implementation). All of the *BSDs currently have one of these user-level packages as their primary thread system.

User-level threads can be implemented without kernel support, which is useful for platforms without native thread support, or applications where only a particular subset of thread operations is needed. The GNU PTH library, for example, does not support preemptive time-slicing among threads, and is simpler because of it. Thread creation, synchronization, and context switching can all be implemented with a cost comparable to an ordinary function call.

However, operations that conceptually block a single thread (blocking system calls such as read(), or page faults) instead block the entire process, since the kernel is oblivious to the presence of threads. This makes it difficult to use user-level threads to exploit concurrency or provide good interactive response. Many user-level thread packages partially work around this problem by intercepting system calls made by the application and replacing them with non-blocking variants and a call to the thread scheduler. These workarounds are not entirely effective and add complexity to the system.

Additionally, a purely user-level thread package can not make use of multiple CPUs in a system. The kernel is only aware of one entity that can be scheduled to run – the process – and hence only allocates a single processor. As a result, user-level thread packages are unsuitable for applications that are natural fits for shared-memory multiprocessors, such as large numerical simulations.

At the other end of the thread implementation spectrum, the operating system kernel is aware of the threaded nature of the application and the existence of each application thread. This model is known as the "1:1" model, since there is a direct correspondence between user threads and kernel resources. The kernel is responsible for most thread management tasks: creation, scheduling, synchronization, and disposal. These kernel entities share many of the resources traditionally associated with a process, such as address space and file descriptors, but each have their own running state or saved context.

This approach provides the kernel with awareness of the concurrency that exists within an application. Several benefits are realized over the user-thread model: one thread blocking does not impede the progress of another, and multiprocessor parallelism can be exploited. But there are problems here as well. One is that the overhead of thread operations is high: since they are managed by the kernel, operations must be performed by requesting services of the kernel (usually via system call), which is a relatively slow operation. [1]. Also, each thread consumes kernel memory, which is usually more scarce than user process memory. Thus, while kernel threads provide better concurrency than user threads, they are more expensive in time and space. They are relatively easy to implement, given operating system support for kernel execution entities that share resources (such as the `clone()` system call under Linux, the `sproc()` system call under IRIX, or the `_lwp_create()` system call in Solaris). Many operating systems, including Linux, IRIX, and Windows NT, use this model of thread system.

Since there are advantages and disadvantages of both the N:1 and 1:1 thread implementation models, it is natural to attempt to combine them to achieve a balance of the costs and benefits of each. These hybrids are collectively known as "N:M" systems, since they map some number N of application threads onto a (usually smaller) number M of kernel entities. They are also known as "two-level" thread systems, since there are two parties, the kernel and the user parts of the thread system, involved in thread operations and scheduling. There are quite a variety of different implementations of N:M thread systems, with different performance characteristics. N:M thread systems are more complicated than either of the other models, and can be more difficult to develop, debug, and use effectively. Both AIX and Solaris use N:M thread systems by default. [2]

In a N:M thread system, a key question is how to manage the mapping of user threads to kernel entities. One possibility is to associate groups of threads with single kernel entities; this permits concurrency across groups but not within groups, reaching a balance between the concurrency of N:1 and 1:1 systems.

The *scheduler activations* model put forward by Anderson et al. is a way of managing the N:M mapping while maintaining as much concurrency as a 1:1 thread system. In this model, the kernel provides the application with the abstraction of virtual processors: a guarantee to run a certain number of application threads simultaneously on CPUs. Kernel events that would affect the number of running threads are communicated directly to the application in a way that maintains the number of virtual processors. The message to the application informs it

[1] The DG/UX operating system prototyped an implementation that took advantage of the software state saving of their RISC processor to permit fast access to simple kernel operations. This technique has not been widely adopted.

[2] Sun Solaris now also ships with a 1:1 thread library; application developers are encouraged to evaluate both thread libraries for use by their application.

of the state that has changed and the context of the user threads that were involved, and lets the user-level scheduler decide how to proceed with the resources available.

This system has several advantages: like other M:N systems, kernel resource usages is kept small in comparison to the number of user-level threads; voluntary thread switching is cheap, similar to user-level threads, and like 1:1 systems, an application's concurrency is fully maintained. Scheduler activations have been implemented for research purposes in Taos [1], Mach 3.0 [2], and BSD/OS [9], and adopted commercially in Digital Unix [5] (now Compaq Tru64 Unix).

The scheduler activations system shares with other M:N systems all of the problems of increased complexity over 1:1 systems. Additionally, there is concern that the problems addressed by scheduler activations are not important problems in the space of threaded applications. For example, making thread context switches cheap is of little value if thread-to-thread switching is infrequent, or if thread switching occurs as a side effect of heavyweight I/O operations.

Implementing scheduler activations for NetBSD is attractive for two major reasons. First, NetBSD needs a native thread system which is preemptive and has the ability to exploit multiprocessor computer systems. Second, this work makes a scheduler activations interface and implementation available in an open-source operating system for continued research into the utility and viability of this intuitively appealing model.

3 Scheduler Activations

As described by Anderson et al., the scheduler activations kernel provides the application with a set of virtual processors, and then the application has complete control over what threads to run on each of the virtual processors. The number of virtual processors in the set is controlled by the kernel, in response to the competing demands of different processes in the system. For example, an application may express to the kernel that it has enough work to keep four processors busy, while a single-threaded application is also trying to run; the kernel could allocate three processors to the set of virtual processors for the first application, and give the fourth processor to the single-threaded program.

In order for the application to be able to consistently use these virtual processors, it must know when threads have blocked, stopped, or restarted. For user-level operations that cause threads to block, such as sleeping for a mutex or waiting on a condition variable, the thread that is blocking hands control to the thread library, which can schedule another thread to run in the usual manner. However, kernel-level events can also block threads: the `read()` and `select()` system calls, for example, or taking a fault on a page of memory that is on disk. When such an event occurs, the number of processors executing application code decreases. The scheduler activations kernel needs to tell the application what has happened and give it another virtual processor. The mechanism that does this is known as an *upcall*.

To perform an upcall, the kernel allocates a new virtual processor for the application and begins running a piece of application code in this new execution context. The application code is known as the *upcall handler*, and it is invoked similarly to a traditional signal handler. The upcall handler is passed information that identifies the virtual processor that stopped running and the reason that it stopped. The upcall handler can then perform any user-level thread bookkeeping and then switch to a runnable thread from the application's thread queue.

Eventually the thread that blocked in the kernel will unblock and be ready to return to the application. If the thread were to directly return, it would violate two constraints of scheduler activations: the number of virtual processors would increase, and the application would be unaware that the state of the thread had changed. Therefore, this event is also communicated with an upcall. In order to maintain the number of virtual processors, the thread currently executing on one of the application's processors is preempted. The upcall is then run in the context of that virtual processor, carrying notifications of both the first thread returning from the kernel and the second thread being preempted. The upcall handler, knowing all of this, can then decide which thread to run next, based on its own processing needs. Typically, this would choose the highest priority thread, which should be one of the two involved in the upcall notification.

There are other scheduling-related events that are communicated by upcall. A change in the size of the virtual processor set must be communicated to the application so that it can either reschedule the thread running on a removed processor, or schedule code to run on the new processor. Traditional POSIX signals, which would normally cause a control-flow change in the application, are communicated by upcall. Additionally, a mechanism is provided for an application to invoke an upcall on another processor, in order to bring that processor back under control of the thread engine (in case thread engine code running on processor 1 decides that a different, higher-priority thread should start running on processor 2).

4 Kernel Interface

The application interface to the scheduler activations system consists of system calls. First, the `sa_register()` call tells the kernel what entry point to use for a scheduler activations upcall, much like registering a signal handler. Next, `sa_setconcurrency()` informs the kernel of the level of concurrency available in the application, and thus the maximum number of processors that may be profitably allocated to it. The `sa_enable()` call starts the system by invoking an upcall on the current processor. While the application is running, the `sa_yield()` and `sa_preempt()` calls allow an application to manage itself by giving up processors and interrupting other processors in the application with an upcall.

4.1 Upcalls

Upcalls are the interface used by the scheduler activations system in the kernel to inform an application of a scheduling-related event. An application that makes use of scheduler activations registers a procedure to handle the upcall, much like registering a signal handler. When an event occurs, the kernel will take a processor allocated to the application (possibly preempting another part of the application), switch to user level, and call the registered procedure.

The signature of an upcall is:

```
void sa_upcall(int type,
        struct sa_t *sas[],
        int events,
        int interrupted,
        void *arg);
```

The `type` argument indicates the event which triggered the upcall. The types and their meanings are described below.

The `sas` field points to an array of pointers to `struct sa_t` which describe the activations involved in this event. The first element of this array, `sas[0]`, points to the `sa_t` of the running activation. The next elements of the array (`sas[1]` through `sas[events]`) describe the activations directly involved in an event. The value of `events` may be zero, in which case none of the array elements are used for this purpose. The remaining elements of the array (`sas[events+1]` through `sas[events+interrupted]`) describe the activations that were stopped in order to deliver this upcall; that is, the "innocent bystanders". The value of `interrupted` may be zero, in which case none of the elements are used for this purpose.

Upcalls are expected to switch to executing application code; hence, they do not return.

The set of events that generates upcalls is as follows:

- `SA_UPCALL_NEWPROC` This upcall notifies the process of a new processor allocation. The first upcall to a program, triggered by `sa_enable()`, will be of this type.
- `SA_UPCALL_PREEMPTED` This upcall notifies the process of a reduction in its processor allocation. There may be multiple "event" activations if the allocation was reduced by several processors.
- `SA_UPCALL_BLOCKED` This upcall notifies the process that an activation has blocked in the kernel. The `sa_context` field of the event should not be continued until a `SA_UPCALL_UNBLOCKED` event has been delivered for the same activation.
- `SA_UPCALL_UNBLOCKED` This upcall notifies the process that an activation which previously blocked (and for which a `SA_UPCALL_BLOCKED` upcall was delivered) is now ready to be continued.
- `SA_UPCALL_SIGNAL` This upcall is used to deliver a POSIX-style signal to the process. If the signal is a synchronous trap, then `event` is 1, and `sas[1]` points to the activation which triggered the trap. For asynchronous signals, `event` is 0. The `arg` parameter points to a `siginfo_t` structure that describes the signal being delivered.
- `SA_UPCALL_USER` This upcall is delivered when requested by the process itself with `sa_preempt()`. The `sas[1]` activation will be the activation specified in the call.

If the last processor allocated to a process is preempted, then the application can not be informed of this immediately. When it is again allocated a processor, an upcall is used to inform the application of the previous preemption and the new allocation, so that the user-level scheduler can reconsider the application's needs.

The low level upcall mechanism is similar to signal delivery. A piece of code, known as the upcall trampoline, is copied to user space at the start of program execution. To invoke an upcall in the application, the kernel copies out the upcall arguments into user memory and registers, switches to user level with the arguments to the upcall and the address of the upcall entry point available, and starts running at the trampoline code, which calls the upcall routine.

4.2 Stacks

Upcall code, like any application code written in C, needs a stack for storage of local variables, return addresses, and so on. Using the stack of a preempted thread is not always possible, as there is no preempted thread in the case of new processor allocations. Also, using the stack of a preempted thread makes thread management more difficult, because that thread can not be made runnable again until the upcall code is exiting, or another processor might start running it and overwrite the upcall handler's stack area. Therefore, upcalls must be allocated their own set of stacks. The `sa_stacks()` system call gives the kernel a set of addresses and sizes that can be used as stacks for upcalls. Since the kernel does not keep track of when an upcall handler has finished running, the application must keep track of which stacks have been used for upcalls, and periodically call `sa_stacks()` to recycle stacks that have been used and make them available again. By batching these stacks together, the cost of the `sa_stacks()` system call is amortized across a number of upcalls.

4.3 Signals

The POSIX thread specification has a relatively complicated signal model, with distinctions drawn between signals directed at a particular thread and signals directed at the process in general, per-thread signal blocking masks but process-level signal actions, and interfaces to wait for particular signals at both the process and thread level. The method of handling signals under scheduler activations must permit a thread package to implement the POSIX signal model.

Since the kernel does not know about specific threads, it can not maintain per-thread signal masks and affect per-thread signal delivery. Instead, signals are handed to the application via the upcall mechanism, with the `arg` parameter pointing to a `struct siginfo_t`. The user thread library can use this to invoke the signal handler in an appropriate thread context. In order to do this, though, the user thread code must intercept the application's calls to `sigaction()` and maintain the table of signal handlers itself.

5 Kernel Implementation

This section describes the changes needed to implement scheduler activations in the NetBSD kernel, including the separation of traditional process context from execution context, the mechanics of adapting the kernel execution mechanics to maintaining the invariants of scheduler activations, and the separation of machine-dependent and machine-independent code in the implementation.

5.1 LWPs

Most of the systems where scheduler activations has been implemented to date (Taos, Mach, and the Mach-inspired Digital Unix) have kernels where a user process is built out of a set of kernel entities that each represent an execution context. This fits well with scheduler activations, where a single process can have several running and blocked execution contexts. Unfortunately, the NetBSD kernel, like the rest of the BSD family, has a monolithic process structure that includes execution context.

The implementation of scheduler activations on BSD/OS by Seltzer and Small [9] dealt with this problem by using entire process structures for execution context. This had substantial problems. First, the amount of kernel memory that is used for each activation is larger than necessary. Second, using multiple processes for a single application causes a great deal of semantic difficulty for traditional process-based interfaces to the kernel. Applications like `ps` and `top` will show multiple processes, each apparently taking up the same amount of memory, which often confuses users attempting to understand the resource usage of their system. Sending POSIX signals is an action defined on process IDs, but targeting a process which is a sub-part of an application conflicts with the POSIX threads

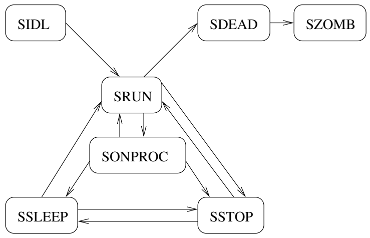

Figure 1: Original NetBSD Process States

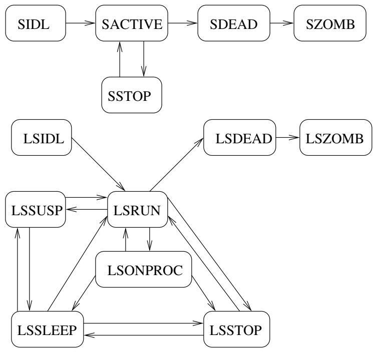

Figure 2: New NetBSD Process and LWP States

specification that an entire application should respond to signals, and that any thread may handle an incoming signal. [3] Finally, complexity must be introduced in the kernel to synchronize per-process data structures such as file descriptor lists, resource limits, and credentials.

Therefore, the first stage in implementing scheduler activations was separating process context from execution context in NetBSD. This was a slow but largely mechanical undertaking. The parts of the classic BSD `struct proc` that related to execution context were relocated to a new structure, `struct lwp` (LWP for "light-weight process", following Solaris and others). This included scheduling counters and priority, the run state and sleep fields, the kernel stack, and space for execution-context-specific machine-dependent fields. The process state values were reduced to those that represent the state of an entire process, and the execution-related process state values were changed to LWP state values.

Following this was an audit of every use of a variable of type `struct proc` within the kernel to determine whether it was being used for its process context or execution context. Execution context turned out to be the prevalent use of such variables, especially the global variable `curproc`. The conversion process consisted of replacing variables like "`struct proc *p`" with "`struct lwp *l`", and changing any code that actually referred to the process-level state to access it indirectly via a pointer in `struct lwp`. The scheduler was converted to handle scheduling LWPs rather than processes; the `fork()` system call was changed to create new LWPs for new processes, and the kernel "reaper" subsystem was adjusted to remove dead LWPs as well as dead processes.

Once this conversion was complete, the next stage was to permit the existence and concurrent execution of several LWPs within a single process. While the scheduler activations system will not have LWPs in a single process time sliced against each other, doing so was a good way of testing the LWP infrastructure, and may also be useful for binary compatibility with systems that use multiple LWPs per process. For the interface, several Solaris LWP functions were adopted: `_lwp_create()`, `_lwp_self()`, `_lwp_exit()`, `_lwp_suspend()`, and `_lwp_continue()`.

Two areas of the kernel were significantly affected by this: signal delivery and process exit. Previously, signal delivery was a straightforward switch on the state of the process. Now, with multiple LWPs and a large combination of possible states, the signal delivery code must iterate over the LWPs in order to find one that can accept the signal. Signals with actions that affect the state of all LWPs, such as `SIGSTOP` and `SIGCONT`, must also iterate over the LWPs and stop or continue each one as appropriate.

Process exit is complicated by the need to clean up all LWPs, not just the one that invokes exit(). The first LWP to attempt to exit must wait for all other LWPs to clean themselves up before cleaning up the process context. The kernel `_lwp_wait()` primitive requires some help to do this, especially in the presence of other LWPs that may be in similar sleep loops in the kernel. They must be coerced to quit their sleep loops while permitting the exiting lwp to continue in its loop. This is done by making tsleep() exit when a process is trying to exit (as noted by the `P_WEXIT` flag in `struct proc`), but providing a flag `PNOEXITERR` that makes the exiting LWP's wait loop ignore `P_WEXIT`.

The machine-dependent parts of the NetBSD kernel each require some porting work to make them work with LWPs. Some of this is straightforward: implementing the machine-dependent back ends for the `getcontext()` and `setcontext` system

[3] The LinuxThreads pthread implementation, while not based on scheduler activations, uses entire processes as threads and has all of these problems. "Why doesn't `kill` work on my threaded application?" is a frequently heard question in Linux and thread programming forums.

calls and changing some flags from P_FLAGNAME to L_FLAGNAME. More involved is splitting the machine-dependent parts of the old struct proc into the new struct proc and struct lwp. For example, on the i386, the TSS selector needs to be LWP-specific, but the pointer to the system call entry point needs to be proc-specific. On some architectures, such as the PowerPC, the machine-dependent part of struct proc becomes empty.

Finally, some delicate work is required in the code that implements the process context switch, which is usually written in assembler. The existing cpu_switch() function, which picks another process to run from the run queue, must be modified to return a flag indicating whether or not it switched to another process. This is used by the scheduler activations code to determine if a preemption upcall needs to be sent. A variant of this routine called cpu_preempt() must be implemented, which takes a new LWP to switch to, instead of picking one from the run queue. This is used by the scheduler activations code to continue executing within the same process when one LWP is blocked.

5.2 Scheduler Activations

The kernel implementation of the actual scheduler activations system is centered on a routine, sa_upcall(), which registers the need for an upcall to be delivered to a process, but does not actually modify the user state. This routine is used, for example, by system calls that directly generate upcalls, such as sa_preempt() and sa_enable().

The most interesting work occurs when a process running with scheduler activations enabled is in the kernel and calls the tsleep() function, which is intended to block the execution context and let the operating system select another process to run. Under the scheduler activations philosophy, this is the moment to send an upcall to the process on a new virtual processor so that it can continue running; this is handled by having tsleep() call a function called sa_switch() instead of the conventional mi_switch(). The mechanics of this are complicated by resource allocation issues; allocating a new activation LWP could block on a memory shortage, and since blocking means calling tsleep() again, recursion inside tsleep would result. To avoid this problem, a spare LWP is pre-allocated and cached when scheduler activations are enabled, and each such LWP allocates another as its first action when it runs, thus ensuring that no double-sleep recursion occurs.

The sa_switch() code sets the LWP to return with a "blocked" upcall, and switches to it. The new LWP exits the kernel and starts execution in the application's registered upcall handler.

When the LWP that had called tsleep() is woken up, it wakes up in the middle of sa_switch(). The switch code sets the current LWP to return with an "unblocked" upcall, potentially including the previously running LWP as an "interrupted" activation if it belongs to the same process.

The other important upcall is the "preempted" upcall. The scheduling code in NetBSD periodically calls a routine to select a new process, if another one exists of the same or higher priority. That routine calls sa_upcall() if it preempts a scheduler activations LWP.

The upcall delivery is done just before crossing the protection boundary back into user space. The arguments are copied out by machine-dependent code, and the process's trap frame is adjusted to cause it to run the upcall-handling code rather than what was previously active.

5.3 Machine dependence

The architecture-dependent code needed to support scheduler activations in NetBSD amounts to about 4000 lines of changed code per architecture. The bulk of that is the mechanical replacement of struct proc references with struct lwp references. About 500 lines of new code is necessary to implement getcontext() and setcontext, the machine-dependent upcall code, and the cpu_preempt() function. To date, the work to make an architecture support scheduler activations has been done by the author on two architectures, and by other people on five others. None of the volunteers who did this porting work reported any significant problems in the interface between machine-dependent and machine-independent scheduler activations code.

6 Thread Implementation

The principal motivation for the scheduler activations system is to support user-level concurrency, and threads are currently the dominant interface for expressing concurrency in imperative languages. Therefore, part of this project is the implementation of an application thread library that utilizes the scheduler activations interface. The library is intended to become the supported POSIX-compatible ("pthreads") library for NetBSD.

The thread library uses the scheduler activations interface described previously to implement POSIX threads. The threads are completely preemptable; the kernel may interrupt a thread and transfer control to the upcall handler at any time. In practice, this occurs most often when a system call blocks a thread or returns from having been blocked, or after another process on the system has preempted the threaded process and then returned. On a uniprocessor system, for example, logging in remotely via `ssh` and running a threaded process that produces terminal output causes frequent preemptions of the threaded process, as the kernel frequently allocates time to the `sshd` process to send terminal output back to the user. Periodic timers are used to generate regular upcalls to implement round-robin scheduling, if requested by the application.

The complete preemptability of scheduler activations threads is a problem in that it violates the atomicity of critical sections of code. For example, the thread library maintains a run queue; if an upcall happens while the run queue is being operated on, havoc will result, as the upcall handler will itself want to manipulate the run queue, but it will be in an invalid state. When spin locks are used to protect critical sections, this problem can lead to deadlock, if the upcall handler attempts to acquire the same lock that was in use by the interrupted thread.

Both the original scheduler activations work and the Mach implementation faced this problem, and both adopted a strategy of recovery, rather than prevention. That is, rather than violate the semantics of scheduler activations by providing a mechanism to prevent interruption during a critical section, they devised ways to detect when a critical section had been interrupted, and recover from that situation. [4] In both implementations, the upcall handler examines the state of each interrupted thread to determine if it was running in a critical section. Each such thread is permitted to run its critical section to completion before the upcall handler enters any critical section of its own.

The implementation of critical-section recovery in this thread library closely follows the Mach implementation. Critical sections are protected with spin locks, and the spin lock acquisition routine increments a spin lock counter in the acquiring thread's descriptor. When an upcall occurs, the upcall handler checks the spin lock count of all interrupted threads. If it finds an interrupted thread that holds spin locks, it sets a flag in the descriptor indicating that the thread is being continued to finish its critical section, and then switches into the context of that thread. When the critical section finishes, the spin lock release routine sees the continuation flag set in the descriptor and context switches back to the upcall handler, which can then proceed, knowing that no critical sections are left unfinished. This mechanism also continues the execution of any upcalls that are themselves preempted, and that may have been continuing other preempted critical sections.

While this system is effective in preventing problems with preempted critical sections, the need to manipulate and examine the thread's descriptor in every spin lock operation undesirably adds overhead in the common case, where a critical section is not preempted. An area for future exploration would be a mechanism more similar to that of Anderson's original implementation, which uses knowledge of which program addresses are critical sections to shift all of the costs of preemption recovery to the uncommon case of a critical section being preempted.

The thread implementation also has a machine-dependent component, although it is much smaller than the kernel component. Short routines that switch from one thread to another (saving and restoring necessary register context), and variants of these routines that are needed for the preemption detection described above, are needed for each CPU type supported by NetBSD.

7 Performance Analysis

There are two goals of examining the performance of the scheduler activations system. The first is determining whether the added complexity of having scheduler activations in the kernel hurts the performance of ordinary applications. The second is comparing the performance of the resulting thread system with existing thread systems to demonstrate the merits of the scheduler activations approach.

The first measurements were done with the HBench-OS package from Harvard University [3]. HBench-OS focuses on getting many individual measurements to explore the performance of a system in detail. Five of the

[4] Adding to the terminological confusion, some Sun Solaris documentation refers to the `schedctl_start()` and `schedctl_stop()` functions, which temporarily inhibit preemption of a LWP, as "scheduler activations".

tests measure system call latency; the results from the NetBSD kernel before and after the implementation of scheduler activations are shown here, as measured on a 500 MHz Digital Alpha 21164 system.

	before SA	after SA
getpid	0.631	0.601
getrusage	4.053	4.332
timeofday	1.627	1.911
sbrk	0.722	0.687
sigaction	1.345	1.315

The results are mixed; some tests are faster than before; others are slower. The larger set of HBench-OS tests test process-switch latency for a variety of process memory footprints and number of processes being switched; that set showed similarly mixed results. So while the performance of the system changed slightly, it did not conclusively get faster or slower. This result makes the work as a whole more attractive to the NetBSD community, since it does not require accepting a performance trade-off in order to get a better thread system.

For measuring thread operation costs, three different micro-operations were measured: The time to create, start, and destroy a thread that does nothing; the time to lock and unlock a mutex (under varying degrees of contention), and the time to switch context between exiting threads.

The operations were measured on an Apple iBook, with a 500MHz G3 processor with 256k of L2 cache. The tests were conducted with the scheduler activations thread library on NetBSD, the GNU PTH library on NetBSD, and LinuxThreads on Linuxppc 2.4.19.

	SA	PTH	Linux
Thread	$15\mu s$	$96\mu s$	$90\mu s$
Mutex	$0.4\mu s$	$0.3\mu s$	$0.6\mu s$
Context	$225\mu s$	$166\mu s$	$82\mu s$

LinuxThreads exhibited roughly linear scaling of lock time with the number of threads contending for the lock; neither the SA library nor PTH had noticeable increases in lock time, demonstrating interesting differences in the scheduler involved. The PTH null thread creation time is also surprisingly large, for a pure user-space thread library that should have low overhead. Linux does quite well at the context-switch test, and it will be worth investigating how to get that speed in NetBSD.

In all three benchmarks, The scheduler activations threads demonstrated that basic operations are competitive with both pure user threads and 1:1 kernel threads.

8 Conclusions and future work

This paper has presented the design and implementation of a two-level thread scheduling system based on the scheduler activations model, including the kernel interface, kernel implementation, and user implementation. Measurements were taken that demonstrate both competitive thread performance and no sacrifice in non-threaded application performance. The implementation is sufficiently well-divided between machine-dependent and machine-independent parts that porting to another architecture is only a few days' work. As was initially hypothesized, scheduler activations is a viable model for a thread system for NetBSD. The existence of a scheduler activations implementation in a portable, open source operating system will enable further research into the properties of this appealing system.

This project continues to evolve, but several future goals are clear: integration with the main NetBSD source tree; cooperation with the fledgling support for symmetric multiprocessing in NetBSD; implementation of better critical section preemption as described, implementation of optional POSIX threads features such as realtime scheduling, and as always, performance tuning.

Availability

The kernel and user code described here is available under a BSD license from the NetBSD Project's source servers [5], currently on the CVS branch called `nathanw_sa`. Machine-dependent code has been written for the Alpha, ARM, i386, MIPS, Motorola 68k, PowerPC, and VAX architectures, with more on the way. Integration into the trunk of NetBSD-current is expected after the release of NetBSD 1.6.

Acknowledgments

Thanks to Frans Kaashoek of the MIT Laboratory for Computer Science for supervising early stages of this project, and to Wasabi Systems for sponsoring continued development. Thanks to Klaus Klein for providing a Single Unix Specification-compliant implementation of `ucontext_t` and its associated system calls. Thanks

[5] Please see http://www.netbsd.org/Documentation/current/ for ways of getting NetBSD

to Bill Sommerfeld and Jason Thorpe for review of the interface and many useful suggestions throughout the work on this project. Finally, thanks to Chris Small and Margo Seltzer for their implementation of scheduler activations on BSD/OS; I am indebted to them for their demonstration of feasibility.

References

[1] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proc. 19th ACM Symposium on Operating System Principles*, pages 95–109, 1991.

[2] Paul Barton-Davis, Dylan McNamee, Raj Vaswani, and Edward D. Lazowska. Adding scheduler activations to Mach 3.0. Technical Report 3, Department of Computer Science and Engineering, University of Washington, August 1992.

[3] A. Brown and M. Seltzer. Operating system benchmarking in the wake of lmbench: A case study of the performance of NetBSD on the Intel x86 architecture. In *Proceedings of the 1997 ACM SIGETRICS Conference on Measurement and Modeling of Computer Systems*, pages 214–224, 1997.

[4] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997. ISBN 0-201-63392-2.

[5] Digital Equipment Corporatiom. *Guide to DECthreads*. Digital Equipment Corporation, 1996. Part number AA-Q2DPD-TK.

[6] Ralf S. Engelschall. Gnu portable threads. http://www.gnu.org/software/pth/pth.html.

[7] Portable threads library. http://www.media.osaka-cu.ac.jp/ k-abe/PTL/.

[8] Christopher Provenzano. Portable thread library. ftp://sipb.mit.edu/pub/pthreads/.

[9] Christopher Small and Margo Seltzer. Scheduler acvivations on BSD: Sharing thread management between kernel and application. Technical Report 31, Harvard University, 1995.