A Library Implementation of POSIX Threads under UNIX

Frank Mueller[1] – Florida State University

ABSTRACT

Recently, there has been an effort to specify an IEEE standard for portable operating systems for open systems, called POSIX. One part of it, the POSIX 1003.4a threads extension (Pthreads for short) [12], describes the interface for light-weight threads that rely on shared memory and have a smaller context frame than processes.

This paper describes and evaluates the design and implementation of a library of Pthreads calls that is solely based on UNIX. It shows that a library implementation is feasible and can result in good performance. This work can also be used as a comparison of the performance of other implementations, or as a prototyping, testing, and debugging system in the regular UNIX environment. Finally, some problems with the Pthreads standard are identified.

Introduction

Light-weight threads are independent threads of control within a regular process that share global data (global variables, files, etc.) but maintain their own stack, local variables, and program counter. Threads are referred to as light-weight because their context is smaller than the context of processes. Therefore, context switches between threads may be cheaper than context switches between processes. Furthermore, threads are an adequate model to implement Ada tasks and provide a simple but powerful model for exploiting parallelism in a shared-memory multiprocessor environment. The POSIX threads extension specifies a priority-driven thread model with preemptive scheduling policies, signal handling, and primitives to provide mutual exclusion as well as synchronized waiting. Although the Pthreads draft is not yet a standard and is still being changed through a balloting process, we will refer to the document [12] as the ''Pthreads standard''. More background on programming with threads is given in [5, 11] as well as in a previous paper describing the early stages of this implementation [17].

This work focuses on the design and implementation issues of POSIX threads (Pthreads) on the Sun SPARC architecture. It describes a true library implementation with a minimal interface to Sun UNIX 4.3 BSD and evaluates its performance.

This article is structured as follows: An overview of previous work in the area precedes the design decisions and their motivations. Then, a brief overview of the Pthreads standard is followed by a more detailed discussion of the design and implementation. Finally, measurements and their evaluations, unresolved problems with the standard, future work, and summary follow.

[1]This work was partially funded by the Ada Joint Program Office, through the U.S. Army CECOM and Telos Corp.

Related Work

Cthreads, an early implementation of threads, is a coroutine-like extension of the language C. A library implementation was used as a teaching tool by Cooper [7]. This original notion of Cthreads lacked priorities, did not handle signals on a per-thread basis, and supported only non-preemptive scheduling. The first commercial operating system to support threads was the Mach OS [23]. Cooper also provided an implementation of Cthreads based on Mach threads thereby supporting preemption. An early library implementation of prioritized preemptive threads at Brown University [14] supports various architectures including a multiprocessor and handled signals asynchronously. Lately, some commercial operating systems (e.g., LynxOS [9], SunOS [18, 22]) support Pthreads by using a mixture of library and kernel implementation, while others such as Chorus [1] provide more functionality as part of the kernel. An earlier, partial implementation of Pthreads on the library level [19] was used as a base for this project.

Motivation

The Pthreads standard provides a uniform base for multiprocessor shared-memory applications, real-time system environments, and a cheap model for multi-threaded programs on a single processor. The notion of threads can be used to implement Ada tasks or to express parallelism within applications at the level of programming languages. An implementation of Pthreads can be carried out as:

- a kernel implementation, where all functionality is part the the operating system kernel;
- a library implementation, where all functionality is part of the user program and can be linked in; or
- a mixture of the above.

A kernel implementation simplifies control over thread operations and signal handling but adds the overhead of entering and leaving the kernel at each call. A library implementation can be more efficient since it does not have to enter the operating system kernel but it complicates signal handling and some thread operations, and it also has to deal with two different scheduler, one for processes (kernel level) and one for threads (library level).

This study discusses the issues of a true library implementation which can be used on a SPARC architecture without specific operating system support for threads. It has been used successfully in an effort to implement an Ada runtime system on top of Pthreads to make the Ada runtime system more portable and to show that the overhead of layering a runtime system on top of Pthreads is not prohibitive.

Pthreads Standard

The Pthreads standard specifies various services that can be provided to support multi-threaded applications. Most of the interface specifications leave many details to the implementation. For example, support for certain functions and the detection of some errors is optional. Therefore, Pthreads-compliant implementations may vary considerably. This implementation supports the following functionality:

- *thread management:* initializing, creating, joining, exiting, and destroying threads;
- *synchronization:* mutual exclusion, condition variables;
- *thread-specific data*;
- *thread priority scheduling:* priority management, preemptive priority scheduling;
- *signals:* signal handlers, asynchronous wait, masking of signals, long jumps;
- *cancellation:* cleanup handlers, different interruptibility states.

The support is currently being extended to include process control.

Design and Implementation

The design of Pthreads has been strongly influenced by constraints of the Pthreads standard, limitations due to the approach of a library implementation, and to some extend by the use of SunOS (UNIX 4.3 BSD) on a SPARC architecture. The machine-dependent part of the implementation consists of about 400 lines of predominantly assembly code. The interface consists of a C library with linkable entry points and can optionally be compiled to generate a language-independent interface. A language interface for Ada has already been designed and tested. Figure 1 illustrates the different software layers of the design.

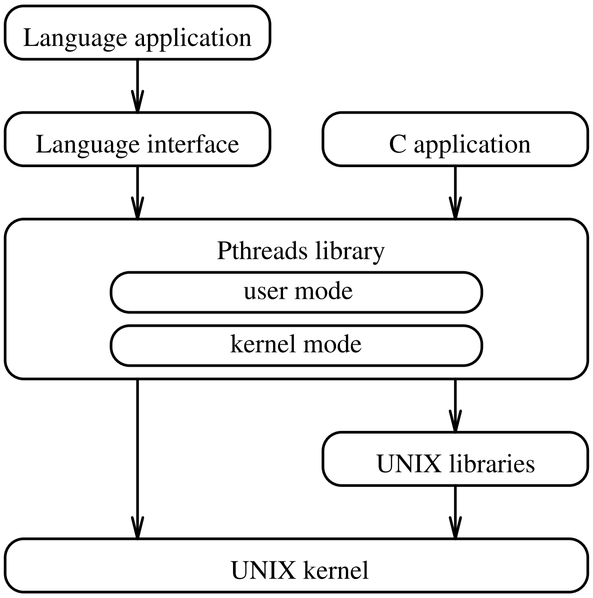


Figure 1: Software Layers

An interface allows programs to use Pthreads services. In case of the programming language C the library routines of Pthreads are immediately available. Any other programming language needs a language interface to the Pthreads library to pass parameters correctly, perform type conversion and other language or compiler-dependent adjustments. The Pthreads library contains a set of routines whose interface and functionality are defined by the Pthreads standard. The code of Pthreads routines partially executes as user code and, within critical sections, operates in the Pthreads kernel mode which guarantees mutual exclusion between threads. The implementation uses a number of UNIX standard library routines and UNIX kernel calls. The design was driven by the following objectives:

- *Preemptability:* Scheduling policies such as round-robin scheduling and asynchronous events (signals) together with priorities can only be supported by a preemptive kernel design.
- *Fast Context Switches:* The context switch is the only means by which control is transferred from one thread to another. A thread's light weight should reduce the context switch overhead.
- *Small Critical Sections:* The time spent in critical sections should be as short as possible. The overhead of entering and leaving critical sections should also be small.
- *No unlimited Stack Growth:* If an asynchronous event arrives while executing an interrupt handler, another handler may be pushed onto the stack and so on *ad infinitum.* A

scheme for handling signals that avoids unlimited stack growth is described below.

- *Few Operating System Calls:* Since calls to the operating systems are time-consuming operations, the use of them should be minimized, especially in time-critical places such as signal handling and context switches.
- *Language-Independent Interface:* The implementation should support the design of an interface to Pthreads with a minimum of dynamic overhead for programming languages other than C.

Pthreads Kernel

Structures allocated by Pthreads must be protected from being modified inconsistently during the handling of asynchronous events (signals). To provide such a protection the library implementation guarantees that critical sections of the library code can only be executed by one thread at a time. The technique which was used to provide mutual exclusion for this implementation is commonly known as a *monolithic monitor* and will be referred to as the library kernel or simply *kernel* in the following.

An alternative to using coarse-gained locking, such as a monolithic monitor, would be to perform fine-grained locking where a different semaphore is associated with each global data structure. The latter approach allows for more concurrency in a multiprocessor environment but more operations need to be performed to guarantee mutual exclusion for each data structure individually. Since this implementation is dedicated to a uniprocessor environment, it was decided to implement a monolithic monitor.

The Pthreads kernel can be entered by setting the kernel flag. Thereafter, any operations are protected so that modifications to thread-internal data structures are guaranteed to be performed in mutual exclusion with other threads. Another flag, the dispatcher flag, indicates whether the dispatcher will be invoked when leaving the Pthreads kernel. The flag is set when a new thread is scheduled or when a signals is received while executing in the Pthreads kernel. To leave the Pthreads kernel, the kernel flag is simply reset if the dispatcher flag was not set; otherwise the dispatcher is invoked which might result in a context switch to another thread. This also allows the implementation to handle signals received from within the kernel as explained below.

Signal Delivery

The delivery of process-level signals to threads is closely coupled with the dispatcher. In particular, signals received while in the kernel are handled differently than signals received while executing instructions outside the kernel although they share a universal signal handler on the process level.

During the initialization of Pthreads a universal signal handler is installed for all maskable UNIX signals. When a signal is caught by the universal handler and the kernel flag is not set, the kernel is entered by setting the kernel flag, all signals are enabled, and a routine is called which first directs the signal at the appropriate thread and then calls the dispatcher. The control might not immediately be transferred back to the same thread if the signal made a higher priority thread eligible to run.

When a signal is caught while in the kernel, the received signal is logged and its handling is deferred until the dispatcher is called. The control is then immediately transferred back to the interruption point by returning from the universal signal handler which also enables signals at the process level again.

Thread States

A thread may be blocked waiting for some event, ready to execute (but not chosen yet by the scheduling policy to be dispatched), running (dispatched), or terminated (cannot be scheduled anymore). Furthermore, a thread may be detached in conjunction with any of the above states.

After a detached thread terminates or after a terminated thread is detached, any memory associated with the thread can be reclaimed and the thread may not be referenced any longer.

The Dispatcher

Under normal circumstances, a call to the dispatcher will select the next thread eligible to run from the set of ready threads according to the scheduling policy. If the selected thread differs from the thread currently running a context switch has to be performed. A thread context switch on the SPARC consists of

- saving non-scratch registers of the current thread which is accomplished on the Sun SPARC by a trap into the UNIX kernel to flush the set of active register windows onto the stack (`ST_FLUSH_WINDOWS`),
- loading the frame pointer with the top of the thread's stack,
- loading UNIX' global error number with the thread's error number,
- loading non-scratch registers and switching to the frame of the new thread by executing a restore instruction, and
- transferring control to the new thread.

On the SPARC, the only registers changed during a thread's context switch are those describing the local state contained in the register windows (ins/outs and locals). Any global state such as global registers, floating point registers, and the status word never need to be updated during a context switch because these registers are either considered to be scratch registers (across explicit calls to Pthreads routines) or are saved by the UNIX (when a signal is delivered). When a thread which was not interrupted by a signal is dispatched, the context is logically switched to the local state of the new thread (see Figure 2). Before the control is transferred

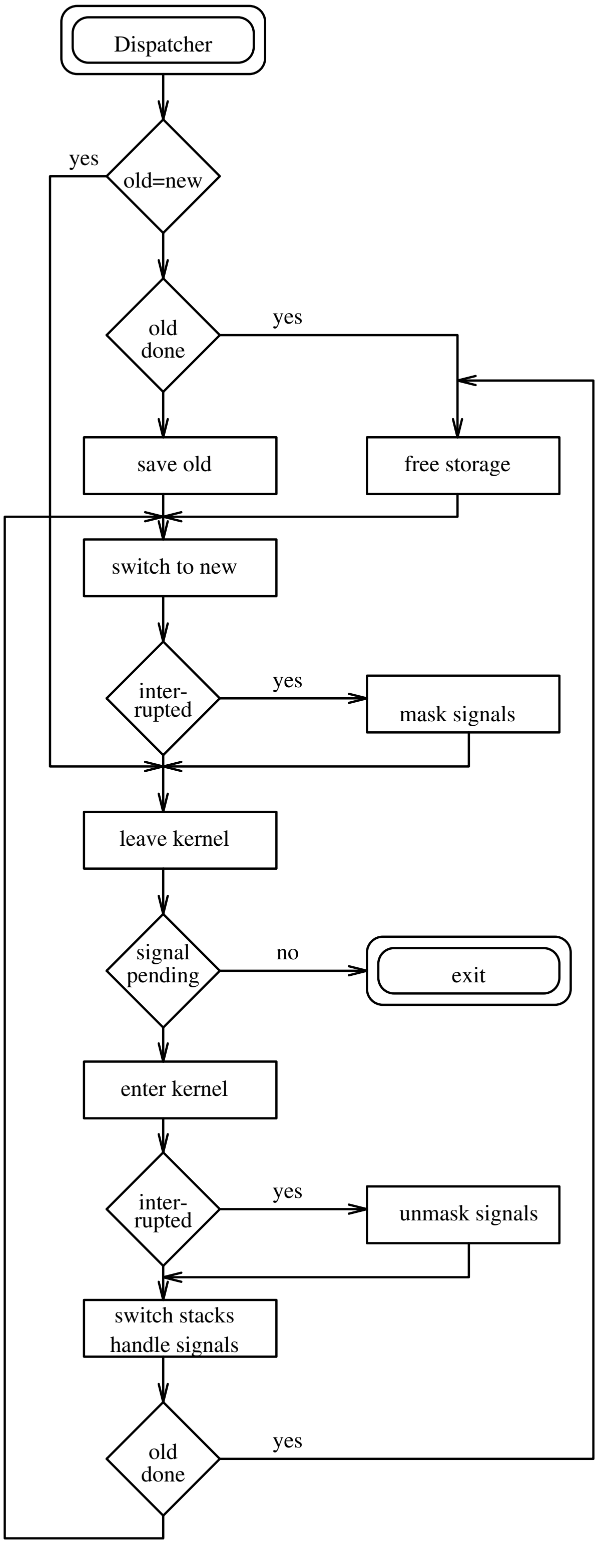


Figure 2: Flowchart of the Dispatcher: Switch Context from old to new Thread

to the new thread, the kernel and dispatcher flags are cleared and it is checked whether signals were caught while in the kernel. If no signals were caught the control can be transferred to the new thread; otherwise the signals will be handled as explained below and another attempt to dispatch a thread will follow. Since the handling of signals might change the thread to be dispatched next, the context switch has to be restarted.

When an interrupted thread is chosen to be dispatched, the universal signal handler will still be pending on top of the thread's stack. Therefore, the dispatcher disables all signals before initiating the context switch. When the thread regains control it will return from the universal signal handler, enable all signals again, and return to the UNIX interrupt frame which will restore the global state (global registers, floating point registers, and the the status word). It is essential to disable signals before switching to the context of an interrupted thread to avoid unbounded stack growth. Otherwise the universal signal handler could be interrupted by yet another instance of the universal signal handler (and so on) before the thread can return from the first instance of the handler.

Signal Handling

Process level signals are deferred until the dispatcher is called if they were caught while in the Pthreads kernel. Otherwise, they are handled immediately. Signal handling determines the receiving thread and the action to be taken for the signal. The recipient is determined according to the so-called *signal delivery model* which describes when a thread receives a signal and how conflicts between multiple threads are resolved. This implementation uses the following conflict resolution (beginning with the highest precedence):

1. If the signal is specifically directed at a thread, this thread is the recipient; else
2. if the signal is delivered synchronously, direct it at the thread which caused it; else
3. if the signal was caused by a timer expiration, direct it at the thread which armed the timer; else
4. if the signal was caused by an I/O completion, direct it at the thread which requested I/O; else
5. if any thread has the signal unmasked, direct it at such a thread; else
6. pend the signal on the process level until a thread becomes eligible to receive it.

The choice of an arbitrary thread is sufficient in step 5 to comply with the Pthreads standard. This implementation performs a linear search of a list of all threads until either all threads are exhausted in the search or a thread is found which has the signal unmasked. (The routine `sigwait` is just another case where the signal is unmasked).

If a thread is selected as the recipient of a signal an action will be selected as follows (beginning with the highest precedence):

1. If the thread masked the signal, pend the signal on the thread; else

2. if the signal is the alarm signal and was caused by a timer expiration, the selected thread either becomes ready if it was suspended or it is position at the tail of the ready queue if the timer expiration was furthermore caused by time-slicing; else
3. if the thread suspended in a call to `sigwait`, the thread becomes ready and signals specified in the call to `sigwait` are masked for the thread; else
4. if a handler has been registered for the signal, a *fake call* will be installed for the selected thread, signals are masked according to the mask specified in `sigaction`, and the thread becomes ready; else
5. if the signal is the cancellation signal (see section ‘‘Thread Cancellation’’), a *fake call* to `pthread_exit` is pushed onto the threads stack and the thread becomes ready; else
6. if the action defined on the signal is to ignore the signal, take no action and discard the signal; else
7. if the action defined on the signal is the default action, perform the default action on the process.

Fake Calls

Thread signal handlers (user handlers) installed by a call to `sigaction` are invoked through a mechanism called *fake call*. A fake call pushes a frame on top of a thread's stack and sets up the frame to act as if a function had been called explicitly by the thread.

The use of fake calls as a mechanism to invoke user handlers is motivated by the constraint that user handlers have to execute at the priority level of the corresponding thread. Rather than making an explicit call to the user handler when a process signal is received, the execution of the user handler has to be deferred until the receiving thread is dispatched. This is enforced through the use of fake calls.

Figure 3 illustrates the mechanism of fake calls. User code is interrupted by a signal causing the operating system to create a new frame which saves the state at the interruption point and invoke Pthreads' universal signal handler which calls the dispatcher. The dispatcher changes to the temporary stack (as indicated by an arrow) but remains active. While executing on the temporary stack the signal is directed to a thread (in this case the interrupted thread), and a new frame, a wrapper, is created on top of the thread's stack (indicated by another arrow). The program counter and stack pointer of the thread have to be updated to reflect the new frame of the fake call, which will execute the wrapper when the thread regains control. The wrapper takes the following actions:

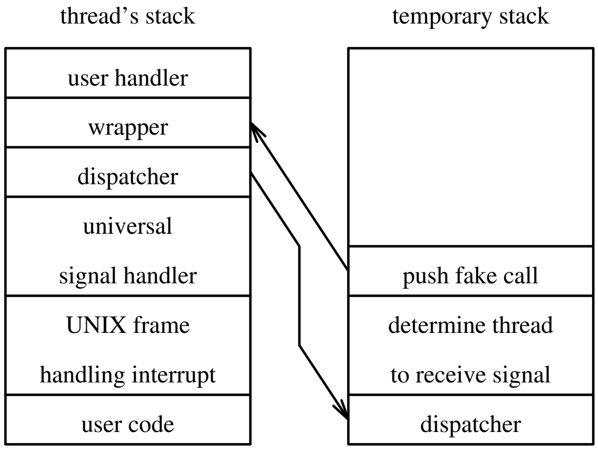


Figure 3: Fake Call onto same Thread

- If the user handler interrupted a conditional wait, the mutex is reacquired and the conditional wait terminated;
- the thread's error number is saved;
- the user handler is called;
- the thread's error number is restored;
- the requested per-thread signal mask is restored and pending signals on the thread and process are handled if now enabled;
- the control is either transferred back to the interruption point or to an instruction whose address can optionally be specified by the user handler.

The ability to redirect control to some specified address is a feature not required by the Pthreads standard but rather left open as implementation defined. Nevertheless, this feature is essential for the Ada runtime system: When a synchronous signal is received, one needs to return from the user handler and restore the previous frame before propagating the exception corresponding to the signal. The Ada runtime system also makes use of the signal code which, in some cases, distinguishes

Interruptibility		Action
State	Type	
disabled	any	SIGCANCEL pends on thread until cancellation is enabled
enabled	controlled	SIGCANCEL pends on thread until interruption point is reached
enabled	asynchronous	Cancellation is acted upon immediately

Table 1: Action taken upon Cancellation Request

between different causes of the same synchronous signal.

Thread Cancellation

A thread may be cancelled by calling `pthread_cancel`. The cancellation is handled as a request for sending a special (internal) signal SIGCANCEL to a thread. Depending on the interruptibility state of the receiving thread, an action will be taken upon a cancellation request according to Table 1.

Interruption points are functions defined in the Pthreads interface which may suspend a thread indefinitely (e.g., conditional waits) with the exception of locking a mutex. Locking a mutex should not be an interruption point to allow for efficient implementations. An interruption point can also be created by calling `pthread_testintr`.

If a cancellation request is acted upon, the interruptibility state of the receiving thread is changed to *disabled*, all other signals are disabled for this thread, and a fake call to `pthread_exit` is pushed onto the thread's stack.

UNIX Interface

To maximize the performance of a true library implementation, calls to the operating system kernel need to be minimized. The overhead associated with entering and leaving the UNIX kernel makes kernel calls expensive operations. This implementation makes use of about 20 UNIX services most of which are used for initialization of the Pthreads library and a few other non-time-critical stages. However, there are a few exceptions.

- When a context switch is performed on the SPARC the register windows of the current thread are flushed via a system trap instruction. The register windows of the new thread will be loaded when the restore instruction is executed which causes a window underflow trap. These two traps consume most of the time required for a context switch and are inherent to any context switch on the SPARC.
- Thread creation/termination involves allocation/deallocation of heap space which sporadically may result in kernel calls to `sbrk`. This could be avoided in most cases by preallocating a pool of thread control blocks and stacks. Thus, dynamic memory allocation would only be performed when the pool space is exhausted at creation time.
- It is most crucial to minimize the use of kernel calls when signals are caught or handled; in particular, calls to `sigsetmask` need to be minimized and signals should be blocked for the shortest interval possible to avoid the loss of signals at the UNIX process level. This implementation uses two calls to `sigsetmask` for each signal received by the process.

Synchronization

The Pthreads standard specifies a ''mutex'' object, a data structure for mutually exclusive access of shared data structures and condition variables for synchronization between threads. Other synchronization methods such as counting semaphores [3] can be easily implemented on top of these primitives [17].

A thread may acquire (lock) an unlocked mutex. Thereafter, mutual exclusion between threads in the same process is guaranteed, until the mutex is unlocked provided that other threads guard critical sections by the same mutex. If a thread tries to lock a mutex which is already locked, the thread suspends. If a thread unlocks a mutex and other threads are waiting on the mutex, the waiting thread with the highest priority will acquire the mutex. To simplify implementations, a thread cannot be cancelled while in controlled interruptibility when it suspends due to mutex contention to guarantee a deterministic state of the mutex in cleanup handlers.

A mutex should only be locked for the shortest possible time to minimize contention. For example, one should protect access to data structures shared between threads by mutexes. But one should not lock a mutex, perform an action which might cause suspension of the thread, and then unlock the mutex, since contention is likely to occur while the thread holding the mutex suspends.

To allow synchronization between threads and suspension over a longer (possibly unbounded) time interval, the standard introduces condition variables. A mutex and a predicate based on shared data are associated with a condition variable. When a thread wants to synchronize with another thread, it locks the mutex, tests the predicate and, if the predicate evaluates to false, suspends on a conditional wait. When the thread is reactivated, it reevaluates the predicate and so on until the predicate becomes true.

The reevaluation of the predicate is essential since spurious wakeups on a multiprocessor and wakeups due to asynchronous events may cause the thread to resume execution while the predicate still evaluates to false.

When a thread enters a conditional wait with the associated mutex locked, the mutex is unlocked atomically with the suspension of the thread. Similarly, the mutex is atomically relocked when thread resumes execution. Thus a mutex is always in a known state even when signals interrupt a conditional wait since the mutex will be reacquired before any interrupt handler starts executing.

A condition variable is typically signaled by a thread after the thread changes the state of some shared data allowing the associated predicate to evaluate to true. When a condition variable is signaled, at least one of the threads blocked on it become ready. If more than one thread is blocked on a condition variable, the thread with the highest priority will become ready. In particular on multiprocessors, an implementation that allows multiple threads become unblocked on signaling a condition variable may be more efficient.

Mutexes should be implemented to provide mutual exclusion in the most efficient way possible. Ideally, a simple test-and-set instruction should be sufficient. Unfortunately, this would result in several deficiencies:

- The standard also specifies more complex optional protocols such as priority inheritance to avoid priority inversion (see section ‘‘Priority Inversion: Inheritance and Ceilings’’). Priority inheritance requires that the ownership of the mutex be recorded atomically with the locking operation.
- Hardware implementations of test-and-set instructions often perform worse than restartable atomic instruction sequences for mutual exclusion on a uniprocessor [4].

The priority inheritance protocol requires that if a high priority thread suspends on a mutex due to contention with a low priority thread which holds the mutex, the low priority thread inherits the high priority until it unlocks the mutex. Thus, the ownership association of a mutex allows the high priority thread to boost the priority of the thread holding the mutex. This will be discussed in more detail later. But first, several implementation options for recording the ownership of a mutex atomically with locking the mutex shall be considered.

A restartable atomic sequence is guaranteed to be atomic by augmenting the signal handler. If such a sequence was interrupted by the signal handler, the atomic sequence is restarted in the signal handler; otherwise no action in taken. For the implementation of the mutex lock, it is thereby guaranteed that there be an owner associated with every locked mutex at any given time.

This scheme does not extend to multiprocessors. Rather, test-and-set instructions become essential as they are the only means to guarantee atomic updates of memory. But restarable atomic sequences can be used to record ownership in conjunction with a test-and-set instruction on multiprocessors by letting the contending thread spin until the bounded interval between locking the mutex and setting the owner must have passed for the acquiring thread [15].

On the SPARC, the test-and-set instruction is about as fast as a restartable atomic sequence [4]. It was therefore decided to use the test-and-set instruction for mutual exclusion but executed it inside a restartable atomic sequence which also included the recording of the ownership. Such a sequence consists of 7 instructions in our implementation (see Figure 4), two instruction more than required by SunOS 5.0 [15]. Sun reserves a hardware register to contain the current thread ID at any time which saves an address calculation and a load required in our implementation. Since this hardware register is reserved for internal use by the SPARC Application Binary Interface [21], such properties cannot be guaranteed by our implementation for any register without changing the operating system kernel.

```
ldstub  [%o0+mutex_lock],%o1
tst     %o1
bne     mutex_locked
sethi   %hi(_kern),%o1
or      %o1,%lo(_kern),%o1
ld      [%o1+pthread_self],%o1
st      %o1,[%o0+mutex_owner]
```

Figure 4: Atomic Sequence to Lock a Mutex and Record the Owner

An additional atomic instruction besides the test-and-set instruction would have avoided these problems: Consider a *compare-and-swap*) which atomically tests some memory word and sets it to the value of a specified register if the memory location contained zero. Let the condition flags also be set by the testing. Then this instruction could be used to record ownership instead of the restartable atomic sequence. Such an approach removes the overhead induced on signal handlers by atomic sequences. But the compare-and-swap instruction would need two more cycles to execute than the test-and-set to perform the comparison and decide whether to update the memory word. This does not seem critical though since a test-and-set instruction will always be followed by a test and a conditional branch instruction to check on the success of the operation. The encoding in 32 bits is the same for both the test-and-set and compare-and-swap instruction since a memory location and a register will be specified in both cases. Therefore, a compare-and-swap instruction should be provided in the instruction set of any processor.

Ada Interface and Binding

A system-level interface between the Pthreads C library and the language Ada has been implemented. This interface has been used to implement an Ada runtime system which is able to map Ada tasks onto threads due to the similarity of their properties. The runtime system can be easily ported to other systems that support Pthreads except for a few implementation-dependent features of Pthreads (e.g., use of signal context record). An Ada binding for Pthreads (user-level interface), on the other hand, is more complex than a bare language interface. Several services provided by Pthreads interfere with the Ada language definition, in particular the handling of signals. We are currently engaged in an effort to define a suitable subset of Pthreads operations as a safe Ada binding [10].

It is suggested by the Pthreads standard that several Pthreads routines be implemented as C macros. Unfortunately, this is a severe limitation to the language-independent approach taken otherwise.

Most notably, cleanup handlers are suggested to be implemented as a pair of macros: `pthread_cleanup_push` opens a new lexical scope, declares a cleanup structure automatically, and links it to a thread-specific cleanup stack. `pthread_cleanup_pop` restores the previous state of the cleanup stack and closes the lexical scope. Since this implementation depends on the creation of lexical scopes it cannot be incorporated as a function call into a language interface. Another layer would have to be included to embed the macro call into a regular C function. Furthermore, the current cleanup structure could no longer be allocated as a local variable within the new lexical scope but would have to become a global variable. Finally, to guarantee that the two operations occur as a pair in the same lexical compiler support would be needed.

It was decided to avoid C macros for interface implementations in general. This trades the overhead of function calls otherwise not needed by C applications for the generality and language-independence of the interface. Such an approach seemed favorable over implementation-specific solutions.

Measurements and Evaluation

The Pthreads standard suggests a set of performance metrics based on the set of routines defined in the interface. Table 2 shows selected measurements used in previous studies. The measurements for our implementation were taken on a Sun SPARC 1+ (column 3) and on a Sun SPARC IPX (column 4) under SunOS 4.1 using dual loop timing analysis. Some measurements are compared to those reported for SunOS [18] (column 2) taken on a Sun SPARC 1+. Others are compared to the results reported for a pre-release of LynxOS (column 5) taken on a Sun SPARC IPX.

The benefit of a library implementation is indicated by the fact that to enter and exit the Pthreads kernel is considerably faster than to enter and exit the UNIX kernel. (The latter metric was obtained by timing a `getpid` call.) This is still true for the comparison with Lynx although their performance shows some improvement over traditional UNIX kernels.

The metrics included a pair of mutex lock and unlock operations, first under the assumption the a mutex is requested while unlocked, and second the interval between an unlock by thread A and the return from a lock operation by thread B (which was suspended while A held the mutex). Mutexes are a mechanism designed for fine-grain locking and should consequently only be held for a short time. A thread should therefore seldom suspend on a mutex lock. Thus, it should be attempted to maximize the performance of mutex operations without contention. Semaphore synchronization refers to one Dijkstra *P* operation plus one *V* operation and were implemented on top of mutexes and condition variables [17]. Neither Lynx operations for synchronization nor Sun's ''unbound thread synchronization'' via semaphores is reported to perform as well as ours.

Further measurements were taken for creating a new thread (excluding the context switch time). The thread control block and stack were pre-cached in a memory pool to avoid dynamic memory allocation.

Performance Metric	Time[μsec]			
	Sparc 1+		Sparc IPX	
	Sun	Ours		Lynx
enter and exit Pthreads kernel			0.4	7.5
enter and exit UNIX kernel			18	
mutex lock/unlock, no contention			1	5
mutex lock/unlock, contention			51	
semaphore synchronization	158	101	55	75
thread create, no context switch	56	25	12	
setjmp/longjmp pair	59	49	29	
thread context switch (yield)			37	38
UNIX process context switch			123	41
thread signal handler (internal)			52	
thread signal handler (external)			250	
UNIX signal handler			154	

Table 2: Performance Metrics

Sun's ''unbounded thread creation'' corresponds to this test as it makes the same assumptions. Comparing the measurements, thread creation of this implementation seems to be faster than Sun's.

The performance of a pair of `setjmp` and `longjmp` operations gives a lower bound on the overhead of a context switch but a true context switch involves some additional overhead as indicated by the measurements. Again, this implementation exceeds the speed reported by Sun but matches Lynx'. Little tuning is possible for the context switch on the SPARC since most of the time is spent in the kernel traps to save and restore registers. Also notice that UNIX process context switches are considerably slower than thread context switches. For LynxOS the performance of context switches hardly differs between processes and threads. (The UNIX process context switch time was measured by timing the execution of two alternating processes which activate each other by exchanging signals minus the time required for process signal delivery.)

The measurements taken for signal handling reflect the time it takes from sending a signal until the signal is received. Since this implementation is build on top of the somewhat slow signal handlers provided by UNIX, external signal handling for threads (i.e., signals directed at the process and demultiplexed to threads) is a time-consuming operation. The performance of internal signal handling (i.e., signals directed at a thread for within the process) indicates that a faster implementation might be possible if the operating system kernel was redesigned. This suggests either that threads be implemented as part of the operating system such that signals can be handled within the kernel or that the kernel/user interface allows the kernel to send the signal to the correct user thread directly [16].

Overall, this implementation seems to match and in some cases exceed the performance reported by commercial implementations.

Perverted Scheduling: Testing and Debugging

Debugging on multiprocessors is typically more complex, more expensive (since a whole set of multiprocessors might be blocked), and errors might not always be reproducible. This library implementation can be helpful to detect and analyze errors in a uniprocessor environment before an application is tested on a multiprocessor. Two types of errors can be distinguished, *serial* errors which occur in uniprocessor environments and *parallel* errors which are inherent to parallel execution. Debugging the former type of errors is well understood. But errors of the latter type are often hard to detect. This implementation of Pthreads has been extended for debugging purposes to optionally provide *perverted scheduling*, a set of scheduling policies which simulate parallel execution on multiprocessors. The following set of perverted scheduling polices has been implemented:

- *Mutex Switch:* On each successful locking of a mutex, a context switch is forced by repositioning the current thread at the tail of its priority queue. The thread at the head of the ready queue executes next.
- *Round-Robin Ordered Switch:* On leaving the Pthreads kernel, a context switch is forced by repositioning the current thread at the tail of the lowest priority queue. The thread at the head of the ready queue executes next.
- *Random Switch:* On leaving the Pthreads kernel, a context switch is forced if the next binary random number produced by some pseudo random-number generator is true. In this case, the current thread is repositioned at the tail of the lowest priority queue and the next thread to execute is selected by randomly choosing a thread from the ready queue.

The above policies may not always conform with priority scheduling as defined in the Pthreads standard. In fact, for the latter two a lower priority thread may execute while a higher priority thread is ready. But on a multiprocessor, the execution of high and low priority threads might occur in parallel. By alternately executing high and low priority threads, this implementation tries to simulate parallel execution using concurrency.

Perverted scheduling policies are easier to deal with than time-sliced round-robin scheduling since in the time-slicing case context switches are caused by timer expirations. In multiprogramming environments timer expirations may vary according to a processor load and in a debugging environment timer expirations may further vary depending on debugging actions. Thus, errors which occur during time-sliced round-robin scheduling may not be reproducible.

The perverted scheduling policies have been used to test the robustness of our implementation of an Ada runtime system based on Pthreads. Several errors were detected which did not show up under the FIFO scheduling policy. But none of the errors were inherent to multiprocessors, they could have occurred on a uniprocessor as well with a different (and legal) ordering of execution of threads. Varying the initialization of random number generators for the random switch policy also proved to be a simple but powerful way to influence the ordering of threads during execution. Still, more experience has to be gained to understand all the benefits of perverted scheduling.

Open Problems

Several problems regarding the POSIX standard have to be resolved. One problem, the use of C macros as discussed in a previous section, suggests that greater emphasis should be placed on language-independence.

Non-Blocking Kernel Calls

UNIX does not provide non-blocking equivalents of some blocking system calls, for example for the interface to directories in the file system. Other non-blocking interfaces for I/O, for example, do not provide the correct semantics with regard to POSIX when interrupted by signals.

Marsh and Scott [16] have made suggestions to overcome some problems associated with user-level threads by defining a generic interface between operating system kernel and user level. This interface provides fast communication between the kernel and user-level activity. For example, when issuing non-blocking I/O request the kernel associates the request with a user-provided datum (the calling thread) such that the user-level thread scheduler can be notified of the I/O completion in conjunction with this datum. This obviates signal demultiplexing at the user level which should increase the response to asynchronous events considerably without unduly complicating the operating system kernel.

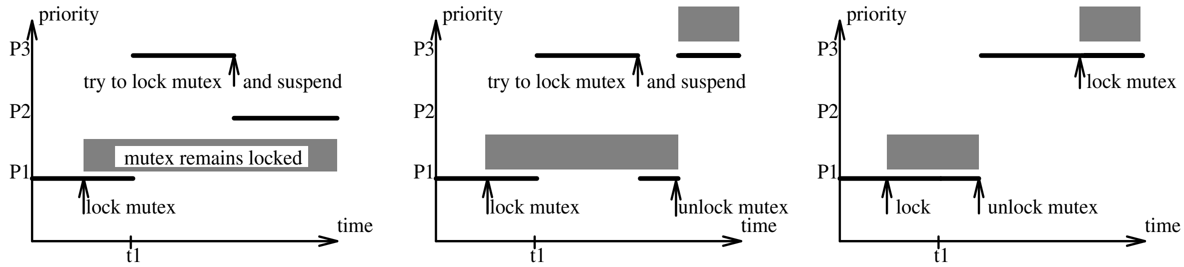
Priority Inversion: Inheritance and Ceilings

Combining priorities and critical sections may cause priority inversion, a situation where a higher priority thread cannot preempt the lower priority thread executing a critical section. Priority inversion may result in unacceptably long delays within multithreaded operating system kernels (microkernels) [8] and user applications. Furthermore, it might not be possible to guarantee timing constraints of real-time systems.

Consider the example in Figure 5(a). A solid line indicates that a thread is executing and a grey box over a thread shows that the thread holds a mutex. A low priority thread P1 locks a mutex and is preempted at *t1* by a high priority thread P3. Thread P3 then tries to lock the same mutex and blocks since the mutex is held by P1. At *t1*, a medium priority thread P2 also has become ready and starts to execute when P3 suspends. Thread P2 does not contribute to the progress of P3 since P3 will not resume its execution until P1 releases the

	Inheritance Protocol	Ceiling Protocol (via SRP)
Boost Priority	of thread holding mutex when a high priority thread suspends on the mutex lock	of the current thread when the mutex is acquired
Boost Prio Level	set to max(own prio, prio of contending threads) by the contending thread	set to prio ceiling of mutex by locking thread
Lower Priority	on unlocking mutex	on unlocking mutex
Lower Prio Level	set to max(own prio, prio of contending threads of other mutexes remaining locked)	set to level before acquiring the mutex
Implementation	linear search of locked mutexes (unlock)	push/pop of ceiling values (stack)
Adaptability	adjusts dynamically to prio level of threads at lock static, prio ceiling set to at least	max(prio of threads locking mutex) at initialization
Bound on Inversion	sum of longest critical section of lower prio threads	tighter: longest critical section of lower prio threads

Table 3: Properties of Synchronization Protocols



(a) no protocol (b) inheritance protocol (c) ceiling protocol

Figure 5: Dealing with Priority Inversion

mutex. Thus, the priorities of P2 and P3 are inverted, the lower priority thread P2 continues its execution without giving the higher priority thread P3 any chance to regain control.

Several protocols have been suggested to overcome priority inversion. The Pthreads standard specifies two protocols, priority inheritance [20] and priority ceiling emulation which can be implemented efficiently using SRP (stack resource policy [2]). A short comparison between the two protocols is given in Table 3. While the implementation of ceilings via SRP is considerably more efficient, priority inheritance can adjust to dynamic changes of priorities which cannot be anticipated and may perform better when contention is rare. For the ceiling protocol, the priority ceiling of a mutex has to be initialized at compile time to a at least the maximum priority of all threads that may lock this mutex. Thus, priority ceilings are associated with the synchronization object (mutex) while priority inheritance is concerned with the priority of threads.

Consider the example in Figure 5(b) with the inheritance protocol. P1 inherits P3's priority when P3 tries to lock the mutex. Thus, P1 runs until it unlocks the mutex and lowers its priority to the original level. P3 then continues to execute since it has the highest priority and can now acquire the mutex. Priority inversion is avoided since P2 does not get to run.

With the ceiling protocol in Figure 5(c), the priority ceiling of the mutex matches (or even exceeds) P3's priority since P3 is the highest priority thread locking the mutex. Thus, when P1 locks the mutex, its priority is raised to the ceiling level. When P1 unlocks the mutex, its priority is lowered to the original level. Although P3 has become ready at *t1* it can only preempt P1 when the mutex becomes unlocked due to the priority ceiling. Later on, P3 locks the mutex and its priority is boosted to the ceiling value. Priority inversion is avoided since P2 never runs. Notice that this protocol tends to require fewer context switches than the inheritance protocol and mutexes are locked for a shorter time.

Several observations regarding the Pthreads standard were made when trying to implement the forementioned protocols:

- The inheritance protocol and the ceiling protocol can be implemented independently. But the Pthreads standard allows ceilings only in the presence of inheritance, which seems to be too restrictive and should be changed.
- The ceiling protocol can be implemented much more efficiently (using a stack [2]) if critical sections are nested properly. Thus, if mutexes are unlocked in a different order than they were locked, the behavior should be undefined for at least the ceiling protocol. Also, if the ceiling of a mutex is not set to the level of the highest priority thread which may lock it, the effect should be undefined. The Standard does not specify such restrictions.
- The implementation of different protocols compromises efficiency. There is only one routine for locking and one for unlocking a mutex defined in the interface. The different protocols are identified by attributes. A simple mutex lock (no protocol) could have been implemented with a test-and-set instruction. But it now requires an additional check of the attributes. It seems preferable to provide different interfaces for each protocol since the actions taken in each case vary considerably.
- The relation of priority scheduling and lowering a thread's priority on unlocking the mutex is ambiguous. It is not clear if a thread will be placed at the tail of its priority level queue as required by the priority scheduling policy or at the head.
  The latter approach seems preferable since a priority boost effects a thread only temporarily. It is not the choice of the thread to change its priority. Rather, the thread is forced into a higher priority. Consequently, neither should any other thread at the same priority level be scheduled instead of the current thread when the priority is reset to the initial level due to the ceiling protocol, nor should the effected thread by penalized for boosting its priority. Furthermore, context switches can potentially be avoided.
- The protocols for inheritance and ceiling do not mix well. In particular, if critical sections with different protocols were nested, the implementation of the ceiling protocol would degrade to that of the inheritance protocol since priority levels would not follow the stack principle (LIFO) anymore. If the ceiling

#	Action	*Pi*	*Pc*	Comment
1	lock(inht)	0	0	no contention for *inht*
2	lock(ceil)	1	1	*ceil* has prio ceiling 1
3	...	2	2	contention for *inht*, inherit prio 2
4	unlock(ceil)	2	0	protocol divergence
5	unlock(inht)	0	0	

Table 4: Mixing Inheritance and Ceiling Protocol

protocol is to be implemented using a stack, the nesting of critical sections using the different protocols for ceiling and inheritance has to be prohibited.

The example in Table 4 illustrates the last point. Consider mutex *inht* with inheritance protocol and mutex *ceil* with ceiling protocol. The priority of the thread using inheritance protocol *Pi* differs from the usage of the ceiling protocol *Pc* in step 4. If the ceiling protocol was implemented as a stack, it would restore the priority prior to locking mutex *ceil*. But this leads to priority inversion for mutex *inht*. If, on the other hand, a linear search was performed on an unlock regardless of protocols, the priority would remain boosted until step 5 and unbounded inversion could be avoided. Thus, the linear search of the inheritance protocol would have to be performed for the ceiling protocol as well if the protocols were mixed.

Future Work

The current status of the implementation still lacks shared mutexes and condition variables which can be used across processes. Such objects could either be implemented on top of existing interprocess communication primitives or by allocating a mutex object in a shared data space. The latter approach should achieve better performance. Nevertheless, enforcing mutex protocols across process boundaries, for example to inherit priorities, seems inefficient in a library implementation since the libraries of the two processes would have to communicate somehow.

It may sometimes be useful to create a new thread but defer its activation, also referred to as lazy thread creation. If threads were used within medium and fine-grain models of parallelism, thousands of threads might be in existence at the same time. Clearly, system resources such as stack space will not suffice for all threads. It may therefore be desirable to create a thread but delay its activation including resource allocation until the thread is ''needed'' by some other thread, for example due to synchronization. An attribute passed at creation time could indicate that the activation is to be deferred.

The current implementation allocates heap space for the stack and thread control block (TCB) at creation time. This accounts for about 70% of the thread creation time. Thus, thread creation could be sped up considerably if a memory pool for TCB and stack was established as was done by other thread implementors.

A major obstacle to the use of threads is to make C libraries reentrant for threads. Several library calls use global state information, some interfaces are non-reentrant, macros have to be modified, and interruption due to signals has to be considered without sacrificing much performance [13]. This issue has not been addressed yet to supplement our implementation with a thread-safe C library among others.

A programming environment for threads should also provide debugging facilities with support for multi-threading [6]. Information could be extracted from the thread control block and made available to the user. Context switches could become visible to the user. For example, when a context switch is about to occur, the user could choose whether to continue debugging after the suspension point of a thread or whether to change into the context of another thread. In addition, separate debugging windows could be allocated for each thread within a process.

The implementation could be extended to support multiprocessors. Several changes would have to be made to the Pthreads kernel. Most notably, the monolithic monitor would have to be replaced by fine-grain locking of shared data structures to minimize contention between different processors while operating in the kernel mode. Otherwise, the advantages of a multiprocessor could not be fully exploited.

Conclusion

It was shown that a true library implementation of Pthreads is possible and feasible. The discussed implementation supports preemptability, fast context switches between threads, small critical sections, avoids unlimited stack growth, uses few operating system calls, and provides a language-independent interface. The implementation seems to exceed the performance of other, kernel-supported implementations, and has been used successfully as a base to implement the tasking portion of an Ada runtime system which passes validation tests for tasking.

The overhead of separate signal handling for each thread complicates the the design and implementation considerably. Some of the advantages of light-weight threads may have been lost due to the requirements of Pthreads. Furthermore, several problems related to language-independence and mutex protocols in particular have to be resolved in future drafts of the standard. It remains to be seen if the Pthreads standard will gain wide acceptance under these circumstances.

Acknowledgments

I would like to thank the following people: Ted Giering and Pratit Santiprabhob for their valuable help and cooperation during the design. Ted Baker for his suggestion for early locking of mutexes for condition variables and his comments about mutex protocols. Viresh Rustagi for the implementation of asynchronous I/O and round-robin scheduling. Bill Gallmeister for providing comparative metrics for LynxOS.

Availability

The source code of Pthreads is available for non-commercial use via anonymous ftp from `ftp.cs.fsu.edu` (128.186.121.27) in the file `/pub/PART/pthreads.tar.Z` – other material such as related publications can be found in the same directory.

Bibliography

[1] Francois Armand, Frederic Herrmann, Jim Lipkis, and Mark Rozier. Multi-threaded processes in CHORUS/MIX. In *Proceedings of EEUG Conference*, pages 1-13, Spring 1990.

[2] T.P. Baker. Stack-based scheduling of realtime processes. *Real-Time Systems*, 3(1):67-99, March 1991.

[3] M. Ben-Ari. *Principles of Concurrent and Distributed Programming*

[4] Brian N. Bershad, David D. Rerdell, and John R. Ellis. Fast mutual exclusion for uniprocessors. In *Proceedings of the Annual Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 223-233, October 1992.

[5] A. D. Birrell. An introduction to programming with threads. Research Report 35, DEC Systems Research Center, 1989.

[6] Deborah Caswell and David Black. Implementing a mach debugger for multithreaded applications. In *Proceedings of the USENIX Conference*, pages 25-40, Winter 1990.

[7] E. Cooper and R. Draves. C threads. TR CMU-CS-88-154, Carnegie Mellon University, Dept. of CS, 1988.

[8] J. Eykholt, S. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. Williams. Beyond multiprocessing ... multithreading the SunOSkernel. In *Proceedings of the USENIX Conference*, pages 11-18, Summer 1992.

[9] Bill O. Gallmeister and Chris Lanier. Early experience with POSIX 1003.4and POSIX 1003.4a. In *Proceedings of the IEEE Symposium on Real-Time Systems*, pages 190-198, 1991.

[10] E.W. Giering and T.P. Baker. POSIX/Adareal-time bindings: Description of work in progress. In *Proceedings of the Ninth Annual Washington Ada Symposium*. ACM, July 1992.

[11] E.W. Giering and T.P. Baker. Using POSIX-threads to implement Adatasking: Description of work in progress. In *TRI-Ada '92 Proceedings*, pages 218-529. ACM, 1992.

[12] IEEE. *Threads Extension for Portable Operating Systems (Draft 6)*, February 1992. P1003.4a/D6.

[13] Michael B. Jones. Bringing the c libraries with us into a multi-threaded future. In *Proceedings of the USENIX Conference*, pages 81-91, Winter 1991.

[14] Thomas W. Doeppner Jr. Threads – a system for the support of concurrent programming. TR CS-87-11, Brown University, Dept. of CS, 1987.

[15] S. Khanna, M. Sebree, and J. Zolnowsky. Realtime scheduling in SunOS 5.0. In *Proceedings of the USENIX Conference*, pages 375-390, Winter 1992.

[16] Brian D. Marsh, Michael L Scott, Thomas J LeBlanc, and Evangelos P. Markatos. First-calss user-level threads. In *Symposium on Operating Systems Principles*, pages 110-121, October 1991.

[17] Frank Mueller. Implementing POSIXthreads under UNIX: Description of work in progress. In *Proceedings of the Second Software Engineering Research Forum*, pages 253-261, November 1992.

[18] M. L. Powell, S. R. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks. SunOSmulti-thread architecture. In *Proceedings of the USENIX Conference*, pages 65-80, Winter 1991.

[19] Ganesh Rangarajan. A library implementation of POSIXthreads. Master's Project Report, Florida State University Department of Computer Science, July 1991.

[20] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175-1185, September 1990.

[21] Inc. SPARC International. *The SPARC Architecture Manual: Version 8*. Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[22] D. Stein and D. Shah. Implementing lightweight threads. In *Proceedings of the USENIX Conference*, pages 1-10, Summer 1992.

[23] A. Tevanian, R. F. Rashid, D. B Golub, D. L. Black, E. Cooper, and M. W. Young. MACHthreads and the UNIXkernel: The battle for control. In *Proceedings of the USENIX Conference*, pages 185-197, Summer 1987.

Author Information

Frank Mueller is currently pursuing his Ph.D. in Computer Science at Florida State University. His interests as a student and research assistant include compilers, computer architecture, Ada, real-time systems, and concurrent programming. His U.S. mail address is Florida State University; Department of Computer Science, B173; Tallahassee, FL 32306-4019. His e-mail address is mueller@cs.fsu.edu.