Implementing POSIX Threads under UNIX*: Description of Work in Progress

Frank Mueller†
Department of Computer Science, B-173
Florida State University
Tallahassee, Florida 32306-4019
phone: (904) 644-3441, *e-mail: mueller@cs.fsu.edu*

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.4a threads extension [3], describes the interface for light-weight threads that rely on shared memory and have a smaller context frame than processes. This paper describes the design and implementation of a library of POSIX.4a calls that is solely based on UNIX. It shows that a library implementation is feasible. This work can also be used as as a prototyping, testing, and debugging system in the regular UNIX environment.

keywords: `Ada language, Ada runtime system, C language, concurrency, light-weight threads, operating system, POSIX standard`

1 Introduction

Some important factors in software engeneering are reuse and portability of software components. Software standards are trying to address these issues. Standards should

- define the minimal criteria for software interfaces which have to be met across different architectures to facilitate portability;
- provide generality to meet the needs of a wide audience;
- facilitate the software design by supporting systematic approaches to problem solving;
- support the process of application development, that is the transition from specification to design, verification, and implementation;
- provide a common base for software development tools.

1.1 POSIX Standards

The IEEE Technical Committee on Operating Systems has been engaged in an effort to specify an interface for portable operating systems (POSIX) with various extensions which is also being adopted by the national (ANSI) and international (ISO/IEC) standards.

The base for the POSIX standard, the "Portable Operating System Interface for Computer Environments" (POSIX.1) [1], defines a non-proprietary application program interface which is derived from UNIX but may as well be provided by any other operating system. POSIX.1 has been approved as a standard. Several extensions are still in work, among them the "Realtime Extension" (POSIX.4) [2] and the "Threads Extension" (POSIX.4a) [3]. The latest draft of the POSIX.4a "Threads Extension" (Pthreads) provides the interface specification for the work described hereafter.

1.2 The Notion of Threads

A thread is an independent flow of control within a process. Threads support the concept of concurrent programming: They may be created and terminated dynamically, such that multiple threads may exist within one process. Threads within the same process 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

*UNIX is a trademark of UNIX System Laborities, Inc.

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

context is smaller than the context of processes. Therefore, context switches between threads are considerably cheaper than context switches between processes. Furthermore, threads are an adequate model to implement Ada tasks. They also 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 synchronization primitives to allow mutual exclusion.

This work focuses on the design and implementation issues of POSIX threads (Pthreads) on a Sun SPARC architecture.[1] It describes a true library implementation with a minimal interface to SunOS 4.1 (UNIX 4.3 BSD).

This article is structured as follows: Section 2 introduces the reader briefly to programming with threads. Section 3 presents an overview of previous work in the area. Section 4 motivates the design decisions. Section 5 discusses the design and implementation in more detail. Section 6 contains directions of future work. Sections 7 summarizes this paper.

2 Programming with Threads

This section is intended for readers who desire to get a brief introduction on when and how to use threads. This discussion is followed by some examples on programming with threads using the C language.

2.1 Application Area

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

- express high-level parallelism within a programming language;
- handle slow I/O devices while other threads within the process remain unblocked;
- perform less important operations in background;
- implement each stage of a pipeline as a separate thread within the same process;
- design a multi-threaded operating system kernel which distributes different tasks among threads;
- exploit the power of shared memory multiprocessors (tightly coupled systems);
- support time-critical applications (realtime) by speeding up the response time for asynchronous events via signal handling.

The programming model of threads allows one to ignore the architectural details such as the number of processors; it rather concentrates on the algorithmic expression of concurrency. The use of concurrency in a programming language requires a certain programming discipline combined with new techniques. The concurrent components of a program may in fact be executed in parallel on a multi-processor.

2.2 Common Pitfalls

Some of the common pitfalls encountered in concurrent programming are discussed in the following:

1. Access to ressources (*e.g.* shared memory) has to be protected by mutual exclusion (*e.g.* by use of semaphores). Generally, it is the programmer's obligation to identify portions of the program which need to be protected by mutual exclusion. Pthreads provides a "mutex" which can be locked to guarantee mutual exclusion until it is unlocked. But a mutex should only be locked for the shortest possible time to minimize the contention of different threads for this mutex.

2. Deadlocks may occur if circular dependencies for resource requests exist. A common practice to avoid deadlocks which are caused by the use of mutual exclusion is to specify the order in which different mutexes may be locked if a thread needs to lock more than one mutex at a time.

3. Starvation may occur if a higher priority thread always succeeds in locking a mutex before a lower priority thread gets the chance to lock it.

4. The number of context switches between threads should be minimized. Although threads are lightweight, a context switch is far from free. By reordering synchronization primitives, the number of context switches may be reduced.

5. The utilization of the system needs to be scaled to maximize the throughput. Too many threads on too few processors may force the operating system into performing poorly, too few threads may not exploit the full power of the hardware.

These issues illustrate that a careful design of application programs is essential to benefit from concurrency and preserve the intended semantic of a program.

[1] Sun and SunOS are trademarks of Sun Microsystems, Inc.; SPARC is a trademark of SPARC International, Inc.

2.3 Programming Examples

A few examples may help to demonstrate the programming techniques using threads in the language C. First a minimal program with two threads is given in Figure 1. The `main` function is the initial thread by definition. The code of any other thread corresponds to the C function which is specified upon the creation operation, such as `new_thread` in the example. The main thread first creates a new thread while passing along an integer as an argument, waits for the new thread to terminate, and finally detaches the new thread to allow the memory of the terminated thread to be reclaimed. The new thread prints out its argument and terminates while the main thread is waiting on the join instruction. To be more precise, the new thread may execute any time after it is created and before the join call of the main thread returns since both threads share the same priority level.

```
#include <pthreads/pthread.h>

void new_thread(arg)
  int *arg;
{
  printf("new thread argument = %d\n", *arg);
}

main()
{
  pthread_t th;
  int i = 1;

  pthread_create(&th, NULL, new_thread, &i);
  pthread_join(th, NULL);
  pthread_detach(&th);
}
```

Figure 1: Process with two threads

Figure 2 shows how mutual exclusion is used to protect shared data structures. A list of items can be shared by many threads. Whenever a thread accesses the list, for example to add an item, it locks the mutex associated with the shared list. It then has exclusive access to the data structure and may modify the head of the list or any links until the mutex is unlocked.

Figure 3 shows how synchronization primitives can be used to implement counting (general) semaphores[4]. Access to a shared resources may be synchronized by semaphores. A thread may call `sem_p` to lock a resource, perform some critical operation, and call `sem_v` to unlock the resource. Since the critical operation may suspend the thread indefinitely, for example if a slow device if accessed, semaphores cannot be implemented by mutexes alone. Mutexes should only be locked for only a very short amount of time (see pitfalls item 1). Condition variables provide the means for synchronization when a thread may suspend indefinitely. Of course, condition variables introduce overhead: A mutex has to be locked before any conditional wait and unlocked afterwards. Furthermore, a condition wait must be embedded in a loop testing for a condition which is set by the signaling party. Spurious wake-ups may occur causing the condition to evaluate to false.

```
typedef struct list {
  struct list *next;
  item t;
} *list_t;

typedef struct shared_list {
  list_t head;
  pthread_mutex_t lock;
} *shared_list_t;

void shared_list_add(q, t)
  shared_list_t q;
  item t;
{
  list_t *new;

  new = malloc(sizeof(*new));
  pthread_mutex_lock(&q->lock);
  new->next = q->head;
  q->head = new;
  pthread_mutex_unlock(&q->lock);
}
```

Figure 2: Mutual exclusion to protect shared data

In Figure 3, the operation P (`sem_p`, to request a resource) has to gain exclusive access to decrement the counter by locking a mutex. If the counter reaches zero the thread is suspended until a resource becomes available. The operation V (`sem_v`, to release a resource) also locks the mutex, increments the counter and signals a thread if the counter was zero. The mutex is unlocked before signaling the condition variable to reduce unnecessary context switches as discussed before (see pitfalls item 4).

The last example in Figure 4 shows how semaphores can be used. Assume that two threads are created, one for the producer and the other for the consumer, both taking a pointer to the same initialized shared buffer as an argument. The producer/consumer pair then acts as a line-buffered pipeline on alternating buffers which are shared between the threads. The buffers are protected by counting semaphores rather than mutexes since the producer and consumer may suspend for a long time when accessing slow devices.

```
typedef struct sem {
  pthread_mutex_t lock;
  pthread_cond_t cond;
  int count;
} sem_t;

void sem_init(s, init)
  sem_t *s;
  int init;
{
  pthread_mutex_init(&s->lock, NULL);
  pthread_cond_init(&s->lock, NULL);
  s->count = (init < 0 ? 0 : init);
}

void sem_p(s)
  sem_t *s;
{
  pthread_mutex_lock(&s->lock);
  while (s->count == 0)
    pthread_cond_wait(&s->cond, &s->lock);
  s->count--;
  pthread_mutex_unlock(&s->lock);
}

void sem_v(s)
  sem_t *s;
{
  int contention;

  pthread_mutex_lock(&s->lock);
  contention = (s->count++ == 0);
  pthread_mutex_unlock(&s->lock);
  if (contention)
    pthread_cond_signal(&s->cond);
}
```

Figure 3: Counting semphores

The producer reads characters from an input stream until the end of a line is encountered and places the characters in the first buffer. It then signals the consumer (`sem_v`). The producer will wait (`sem_p`) if the consumer is still working on the next buffer; otherwise the producer will read more characters into the next buffer.

The consumer, initially suspended waiting for a full buffer, is woken up by the producer and writes the capitalized characters of the first buffer into an output stream. When reaching the end of a line, it signals

```
#define BNUM  2
typedef struct shared_buffer {
  FILE *fdin, *fdout;
  sem_t s_full, s_empty;
  int c[BNUM][BSIZE];
} shared_buffer_t;

void producer(sb)
  shared_buffer_t *sb;
{
  int i, k = 0;
  while (TRUE) {
    sem_p(&sb->s_empty);
    i = 0;
    do {
      if ((sb->c[k][i++] = getc(sb->fdin))
                                  == EOF) {
        sem_v(&sb->s_full);
        pthread_exit(NULL);
    } while (sb->c[k][i-1]!='\n' && i<BSIZE);
    sem_v(&sb->s_full);
    k = (k + 1) % BNUM;
  }
}

void consumer(sb)
  shared_buffer_t *sb;
{
  int i, k = 0;
  while (TRUE) {
    sem_p(&sb->s_full);
    i = 0;
    do {
      if (sb->c[k][i] == EOF)
        pthread_exit(NULL);
      putc(toupper(sb->c[k][i++]),sb->fdout);
    } while (sb->c[k][i-1]!='\n' && i<BSIZE);
    sem_v(&sb->s_empty);
    k = (k + 1) % BNUM;
  }
}

main()
{
  shared_buffer_t sb;
  ...
  sem_init(&sb.s_full, 0);
  sem_init(&sb.s_empty, BNUM);
  pthread_create(&th1, NULL, producer, &sb);
  pthread_create(&th2, NULL, consumer, &sb);
  pthread_join(th2, NULL);
  ...
}
```

Figure 4: Synchronization between threads

the producer (`sem_v`) and, if the next buffer has not been filled by the producer, goes back to sleep (`sem_p`). Upon reaching end-of-file, the producer thread signals the consumer one last time before it terminates (`pthread_exit`). The consumer handles the last buffer until it encounters end-of-file and terminates.

A multi-stage pipeline would create one shared buffer and one producer/consumer pair for each stage and inititialize the input/output streams such that the output of a previous stage is the input for the next stage's buffer. In addition, yet another thread for each pipeline stage could manipulate the buffer, *e.g.* the task of capitalizing each character could be performed concurrently by a third thread and a third buffer in the above example.

These examples should give the reader an idea how much more naturally threads can be embedded in a programming language as opposed to UNIX processes. From the programmer's point of view, a thread may simply be regarded as an instance of the routine specified at creation time. But the examples should also have illustrated the challenge of programming with concurrency. More examples and hints on programming with threads are given in [5, 9].

3 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 [6]. The first commercial operating system to support Cthreads was the Mach OS [13]. Cooper also provided an implementation of Cthreads based on Mach threads. The original notion of Cthreads lacked priorities, did not handle signals on a per-thread basis, and supported only non-preemptive scheduling. Lately, some commercial operating systems (*e.g.* LynxOS [7], SunOS [10, 12]) support Pthreads by using a mixture of library and kernel implementation. An earlier, partial implementation of Pthreads on the library level [11] was used as a base for this project.

4 Motivation

An implementation of Pthreads can be carried out as:

- a kernel implementation, where all functionality is part of 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.

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.

5 Pthreads Standard

The POSIX.4a standard [3] 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, asynchronous, synchronous, and disabled interruptibility.

The support is currently being extended to include other scheduling policies and process control.

6 Design and Implementation

The design of Pthreads has been strongly influenced by constraints of the POSIX.4a standard, limitations due to the approach of a library implementation, and to some extend by the use of SunOS 4.1 (UNIX 4.3 BSD) on a SPARC architecture. The machine-dependent part of the implementation consists of about 300 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 5 illustrates the different software layers of the design.

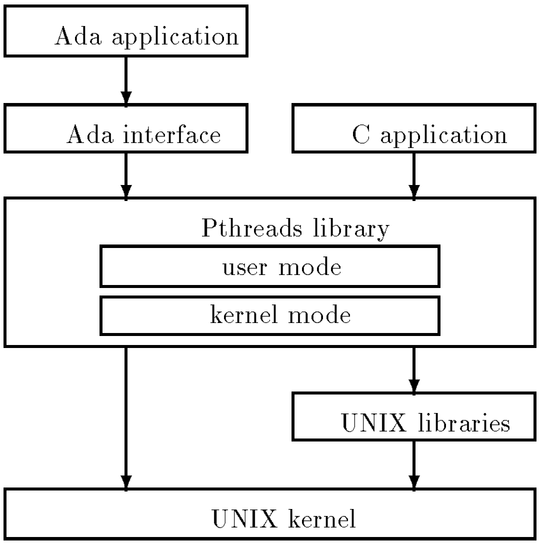

Figure 5: Software Layers

An application layer allows programs to use Pthreads services. In case of the programming language C the library routines of Pthreads are readily 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 POSIX.4a 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:

- *Preemptibility:* Scheduling policies such as round-robin, priorities, and asynchronous events (signals) 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. 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, specifically 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.

6.1 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.

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.

The only way to leave the kernel is to invoke the dispatcher 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.

6.2 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 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 again.

6.3 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

- 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 the program counter of the previous frame with the next instruction the thread has to execute,
- loading UNIX' global error number with the thread's error number, and
- executing a restore instruction to switch to the thread's frame.

Notice that 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. Before the control is transferred to the new thread, the kernel flag is cleared and it is checked if signals were caught while being 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 have changed 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 first disables all signals, switches to the thread's context without transferring control, clears the kernel flag, and then transfers control to the thread if no signals have been caught in the kernel mode. When the thread regains control it returns from the universal signal handler, enables all signals again, and returns to the UNIX interrupt frame which will restore the global state (global registers, floating point registers, and 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.

6.4 Signal Handling

Once a signal is caught at the process level, it will either be handled immediately if the kernel flag is not set, or its handling will be deferred until the dispatcher is called. Signal handling determines the receiving thread and the action to be taken for a caught signal. The recipient is determined according to the so-called *interest model* which describes how a thread can declare interest in receiving 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 any thread has declared interest in the signal, direct it at the thread with the highest priority; else
5. if the action defined on the signal is the default action, perform the default action on the process; else
6. if the action defined on the signal is to ignore the signal, take no action; else
7. pend the signal on the process level until a thread becomes eligible to receive it.

Notice that the choice of an arbitrary thread would have been sufficient in step four to comply with the

Pthreads standard. 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 disabled 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 becomes ready; else
3. if the thread suspended in a call to `sigwait`, the thread becomes ready; else
4. if a handler has been registered for the signal, a *fake call* will be installed for the selected thread and the thread becomes ready.

6.5 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.

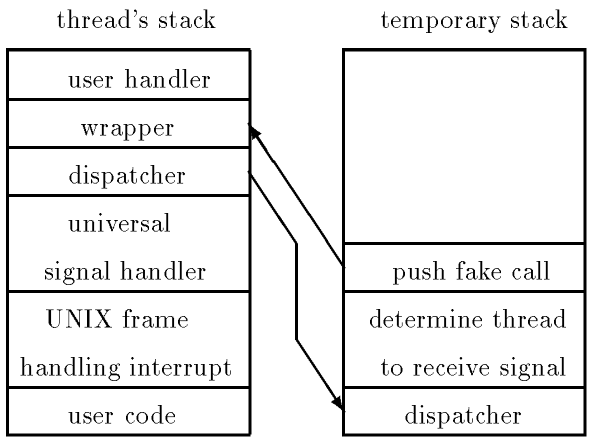

Figure 6: Fake Call onto same Thread

Figure 6 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:

- 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 signal mask is restored and pending signals are handled;
- 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 user-defined return address is a feature not required by the standard.)

6.6 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 12 UNIX services most of which are not time-critical with one major exception. It is most crucial to minimize the use of system 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.

6.7 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.* control is transferred from inside a user handler to an exception handler for signals delivered synchronously).

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[8].

7 Future Work

Previous drafts of the Pthreads document specified a routine `pthread_init` which had to be invoked prior to any thread activity. This operation has been dropped from the lastest draft [3]. For kernel implementations, the initialization could be performed as part of the process initialization but for a library implementation the lack of an initialization routine is problematic. One solution would be for each Pthreads routine to check if initialization has already been performed. This additional check degrades the performance unacceptably thereby making library implementations less feasible. Therefore, this implementation assumes that a call to `pthread_init` is made before other Pthreads routines are used.

Several optional features of Pthreads still need to be implemented, including round-robin scheduling, mutex priority inheritance, and mutex priority ceilings.

Furthermore, threads should also be supported in a debugging environment such that one can extract information about the status of threads and set breakpoints when contexts are switched.

8 Conclusion

It was shown that a true library implementation of Pthreads is possible and feasible. The discussed implementation supports preemptibility, 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 has been used successfully as a base to implement an Ada runtime system which passes validation tests.

The overhead of separate signal handling for each thread complicates the design and implementation considerably. Some of the advantages of light-weight processes may have been lost due to the requirements of Pthreads. It will remain to be seen if the Pthreads standard will gain wide acceptance under these circumstances.

Acknowledgement

I would like to thank Ted Giering and Pratit Santiprabhob for their valuable help and cooperation throughout the project as well as Ted Baker for his advice and support.

References

[1] *Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C Language].* IEEE, 1990. Std 1003.1-1990.

[2] *Realtime Extension for Portable Operating Systems (Draft 12).* IEEE, February 1992. P1003.4/D12.

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

[4] M. Ben-Ari. *Principles of Concurrent and Distributed Programming.* Prentice Hall, 1990.

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

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

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

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

[9] S. Kleiman, B. Smaalders, D. Stein, and D. Shah. Writing multithreaded code in solaris. In *COMPCON Conference Proceedings*, Spring 1992.

[10] M. L. Powell, S. R. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks. SunOS multi-thread architecture. In *USENIX Technical Conference*, Winter 1991.

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

[12] D. Stein and D. Shah. Implementing lightweight threads. In *USENIX Technical Conference*, Summer 1992.

[13] A. Tevanian, R. F. Rashid, D. B Golub, D. L. Black, E. Cooper, and M. W. Young. MACH threads and the UNIX kernel: The battle for control. In *USENIX Technical Conference*, Summer 1987.