Experience with a Prototype of the POSIX "Minimal Realtime System Profile"

T.P. Baker, Frank Mueller, Viresh Rustagi*
Department of Computer Science
Florida State University
Tallahassee, FL 32304-4019

Abstract

This paper describes experience prototyping the proposed IEEE standard "minimal realtime system profile", whose primary component is support for realtime threads. It provides some background, describes the implementation, and reports preliminary performance measurements.

1 Introduction

A *thread* is an independent sequential flow of control. Threads differ from processes by sharing a common virtual address space with other threads. Threads are widely accepted as a computational building block for both uniprocessor and multiprocessor environments. In uniprocessor environments, the thread model simplifies the programming of asynchronous operations. In multiprocessor environments, threads may also allow higher throughput, by utilizing more than one processor.

The idea of cheap concurrency or "lightweight processes" has been around in various forms for a long time, including support for coroutines in the Mesa programming language[13], and multitasking in the Ada programming language[18]. The Pthreads (POSIX Threads) proposal is intended to provide similar functionality for programs in the C language. It is based on considerable experience, including C-threads [2], Mach threads[16, 17], and Brown University threads [3]. Several commercial operating systems support multithreaded processes, including the Lynx[4], Sun[12, 14], and Chorus[1] operating systems.

The POSIX 1003.4a project[8] represents an attempt to achieve some degree of application portability for C programs, across operating systems that support threads. This is an extension of the POSIX application program interface, which generally follows the UNIX process model.

Threads are considered a "real time" extension to POSIX. IEEE draft standard P1003.13[9] proposes a set of realtime application profiles, i.e. subsets of the POSIX standard that are suitable for certain classes of realtime applications. Threads are a key feature of these profiles. In particular, the "Minimal Realtime System Profile" assumes a single process, with threads being the only form of concurrency within the system. The underlying hypothesis is that by not requiring support for the more complex POSIX features, the profile permits an implementation that will be satisfactory for realtime applications with very tight efficiency and timing predictability requirements.

The POSIX proposals are likely to have an impact on future realtime applications development, since they are being promoted as both U.S. Government and international (ISO/IEC) standards.

The draft Pthreads standard specifies the following services:

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

2 Relationship to Ada Tasks

The Ada programming language[18] defines *tasks* as the only form of concurrent threads of control within a program. If the underlying operating system provides direct support for the POSIX (C-language) threads interface, it may be desirable to implement Ada tasks using this interface, by mapping Ada tasks to POSIX threads. Due to differences between the Ada and POSIX/C models, this mapping is not entirely straightforward.

The PART (POSIX Ada Real Time) project, at the Florida State University, is investigating the prac-

*This work was supported in part by the Ada Joint Program Office, via the HQ U.S. Army CECOM Software Engineering Directorate. However, the views reported here do not necessarily reflect a position of the sponsoring organization. Authors may be reached as "baker@cs.fsu.edu".

ticality of using POSIX threads to implement Ada tasks, especially in realtime applications. So far, a complete tasking implementation has been produced for the Ada 83 standard, using an implementation of P1003.4a Draft 6 layered over the Sun UNIX operating system[5]. Work is under way to extend this to the proposed new Ada 9X language standard [6, 19].

3 A "Bare Machine" Implementation

To evaluate the suitability of a Pthreads-based Ada implementation for realtime applications, one must start out with a suitable realtime implementation of Pthreads. Experience with the FSU layered implementation, and other Ada tasking implementations, makes it clear that acceptable realtime performance is not achievable for an implementation layered over a conventional UNIX operating system. Efficiency is certainly an issue, but the main problem is predictability.

Most UNIX implementations impose unpredictable delays on user processes, due to preemptions by interrupt handlers and operating system processes. For an operating system to provide predictable timing of user threads, it must be designed with this objective in mind, from the hardware up. Some commercial realtime operating systems, such as LynxOS and Chorus, apparently have been designed in this way.

As a basis for performance testing of our Ada 9X implementation, we chose to port our existing layered implementation of Pthreads to a "bare" SPARCengine 1E[15]. We chose to do this rather than using an existing commercial realtime OS, for many reasons. Chief among these is that we needed source code, to tune the threads implementation to better support Ada (if necessary), and to take control over interrupts. We also were concerned that the commercial implementations of threads might be too full-blown to take advantage of the restrictions of the POSIX minimal profile, since they support multiple processes, file systems, and a variety of hardware devices. (Finally, there was concern that licensing restrictions would stand in the way of publication.)

The SPARCengine port of the FSU Pthreads library is intended to fit the POSIX minimal realtime systems profile. It runs on a "bare" machine, without any other operating system. It does not support multiple processes, and so operates in single virtual address space. It does not have a file system. At present the only devices supported are a serial port and a timer. These simplifications eliminate unpredictable time delays due to page faults, waiting for completion of I/O, and I/O completion interrupt processing.

The scope of this prototype implementation is limited to a subset of the proposed POSIX minimal realtime system profile. The criteria that governed the choice of this subset are:

- The implementation should be powerful enough to allow testing in a realtime context. This requires the following functionalities:
  - Dynamic creation and termination of threads
  - Synchronization primitives
  - A readable realtime clock
  - Timer support sufficient for periodic task scheduling
  - Output routines to print results
- The implementation should provide sufficient functionality to implement Ada tasking.

The design of the implementation is divisible into three main components:

1. Pthreads support. This implements the detailed functionality of the Pthreads standard, including the dynamic creation and termination of threads, the synchronization primitives, and thread scheduling. It is the largest component, but can be identical to a library implementation.
2. Machine-specific support. This includes code to save and restore register windows for context-switches, boot up the kernel, and provide time-keeping services. The boot code involves initializing memory mapping hardware and installing trap handlers. With the library implementation, all of these functions are performed by the underlying operating system. A bare-machine implementation must perform these functions for itself.
3. C language support, including basic I/O and memory allocation. These are functions provided by the standard C libraries, but the standard Sun Microsystems implementation of the C libraries makes calls to the operating system. Without the support of the operating system, these libraries need to be reimplemented.

The design of the Pthreads functionality was constrained to be *async safe*. A function is *async safe* if calling the function asynchronously will not cause any invariants to be violated, even if it is called from the handler of an interrupt that may be delivered at any time [7]. Even though POSIX does not require the Pthreads functions to be async safe, we chose to require it as a matter of quality. Async safety allows a user to build more responsive realtime systems. Furthermore, it is required to support Ada 9X.

A *single-threaded kernel* approach was used. Once a thread has entered the kernel, no other thread can enter the kernel until that thread has left it. The alternative, a *multi-threaded kernel*, where separate locks are associated with different kernel data structures, would allow more concurrency in a multiprocessor environment. For this to pay off, the cost of interprocessor locking must be low, relative to the time typically spent in kernel. Since we have only a single processor, the choice was clear; the overhead of fine-grained locking would result in poorer performance.

The source code of our bare-machine Pthreads kernel consists of approximately 3300 lines of C-code, of which approximately 1000 lines are new for the bare-machine version and the rest is reused from the library level implementation. The core image of the kernel is 49 kilobytes, as compared to 984 kilobytes for the full Sun UNIX kernel.

Reuse of most of the code from the layered FSU Pthreads library permits direct performance comparisons of the two implementations. Differences can be attributed to running on a bare machine, versus as a layer over the UNIX operating system.

This code was tested for both functionality and performance.

Functional Testing Functional testing was done using a set of 25 tests, derived from tests originally developed to test the layered version of the FSU Pthreads library. The features tested by these bare-machine tests include:

- Thread management - creation, termination, join, detach
- Priority scheduling
- Mutexes - with and without priority ceilings
- Creating and destroying condition variables
- Timed conditional wait
- Thread specific data
- Setjmp/longjmp
- Signal handlers
- Cancellation and cleanup handlers

It was verified that the implementation could pass these tests, before performance testing began.

4 Absolute Performance Results

The performance tests were also derived from tests developed earlier for the layered version of the Pthreads library. These tests attempt to measure the specific performance metrics called out by Draft 6 of Pthreads.

Table 1 shows selected measurements of some of these metrics. The test programs use a dual loop timing analysis technique. The times reported are averages taken over 100,000 iterations. These measurements are compared to measurements taken earlier with the version of the Pthreads library layered over UNIX (SunOS 4.1.3e), on the same machine (VME SPARCengine 1E at 20MHz).

For the layered implementation, the time taken for 100,000 iterations of an operation ranges from about 100 milliseconds to 1 second. Though there was only one user process active, this is long enough that a system process might preempt, so the numbers shown here may be a bit high.

The metrics include:

- *Enter and exit Pthreads kernel.* This is the time taken to enter and immediately exit the kernel.
- *Mutex lock/unlock, no contention.* This is the time to perform a pair of mutex lock and unlock operations, under the assumption that a mutex is requested while unlocked.
- *mutex lock/unlock, contention.* This is the interval between an unlock by one thread and the return from a lock operation by another thread, which was suspended waiting for the mutex.
- *Semaphore synchronization.* This is one Dijkstra P operation plus one V operation. These are implemented on top of mutexes and condition variables.
- *Thread create, no context switch.* This measures the time taken to create a thread, excluding the context switch time.
- *setjmp/longjmp pair.* This is the time taken by a `setjmp` followed by a `longjmp`.
  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.
- *Thread context switch.* This is the time taken for a context switch.
- *Yield (1 thread).* This is the time taken by the *yield* operation when there is only one thread in the system.
- *Yield (2 threads).* In this case, there are two threads in the system.
- *Thread signal handler.* The measurements taken for signal handling reflect the time from sending a signal, by `pthread_kill`, till the signal is received.

Table 1: Performance of some Pthreads Operations

Pthreads Operations	Timings (μsecs)	
	Bare Machine	Layered over UNIX
enter and exit Pthreads kernel	1	1
mutex lock/unlock, no contention	3	3
mutex lock/unlock, contention	44	114
semaphore synchronization	60	103
thread create, no context switch	37	104
setjmp/longjmp pair	16	49
thread context switch	17	95
yield 1 operation	1	1
yield 2 operations	33	70
thread signal handler	55	92

5 Time Predictability Results

From the design of the implementation, we expect our bare-machine implementation to achieve predictable execution timing. The main cause of large deviations from the priority preemptive scheduling model has been eliminated, namely preemption of user threads due to scheduling of other processes, including operating system processes. The precision of timed wakeup events has also been improved, from ten down to one millisecond. With these improvements, we believe that the implementation of priority scheduling is strict enough that actual schedulable utilization will be very close to the theoretical predictions of schedulability analysis.

During debugging, we have already observed that the timing is remarkably consistent. This was evident in the reproducibility of failures due to race-condition problems between (earlier, incorrect versions of) the timer interrupt handler and the rest of the system.

We are currently working on benchmarks to measure the predictability of scheduling for actual task sets, to compare these against theoretical schedulability models, and to estimate the amount of overhead introduced by the Pthreads implementation.

The first test is based on a benchmark developed earlier for a preliminary design of a minesweeper trainer system for the U.S. Naval Coastal Systems Center. This consists of a set of six periodic threads, comprising a realtime simulation. Each thread has three phases. In the first phase, it reads the simulated state of other simulated subsystems from a global database. In the second phase, it computes its own next state. In the third phase, it updates the global database. The read and update phases require locking the database, which is done via a single Pthread mutex. This is shown in pseudo-code in Figure 1. The thread periods, and the execution times of the three phases, are shown in Table 2.

```
for(;;) {
  pthread_mutex_lock(&shared_memory);
  input_data();
  pthread_mutex_unlock(&shared_memory);

  execute();

  pthread_mutex_lock(&shared_memory);
  output_data();
  pthread_mutex_unlock(&shared_memory);

  next_request[task] += period[task];
  if (next_request[task] >= simulation_time)
    break;

  /* suspend until next period */
  pthread_mutex_lock(&mutex[self]);
  do {
    pthread_cond_timedwait(&cond[self],
      &mutex[self], &next_request[self]);
    clock_gettime(CLOCK_REALTIME, &current_time);
  } while (next_request[task] > current_time);
}
```

Figure 1: Task Simulation Algorithm

Task	Period [ms]	Input [μs]	Execute [ms]	Output [μs]	Util.
1	62.5	2.0	44.80	2.0	71%
2	125.0	0.3	0.05	0.3	0%
3	166.7	1.6	27.80	1.6	16%
4	250.0	8.0	0.11	8.0	0%
5	500.0	3.2	5.02	3.2	1%
6	1,000.0	24.0	10.36	24.0	1%

Table 2: Task Set

Such a task system should be suitable for schedulability analysis, based on the Rate-Monotonic model. The objective of our benchmark is to determine how close the actual performance comes to this model.

In the benchmark, a bisection method is used to compute the *breakdown utilization*, at which the tasks can just barely be scheduled without missing any deadlines. This is done by varying a linear scaling factor, called *load factor*, which applies to the execution times of all phases of all the tasks.

The benchmark was run repeatedly over both UNIX and the bare-machine implementation with an initial target utilization of 90%. The results are shown in Figure 2.

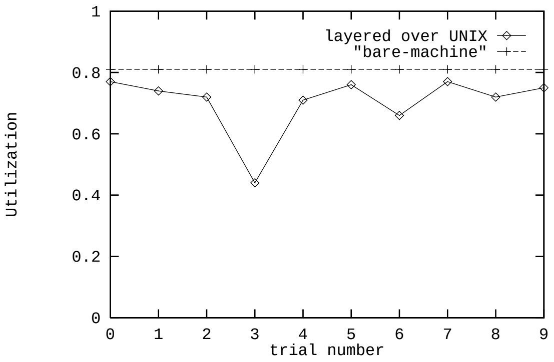

Figure 2: System Utilization for repeated Trails

It was observed that the timing of the benchmark over UNIX varies considerably at times. The bisection sometimes failed on its first iteration, thereby indicating that the breakdown utilization of 90% must reduced below 45%. The bisection would then proceed to terminate at a utilization around 44%. At other times, the bisection succeeded for a trial of a certain load factor. Upon termination of the bisection, the same load factor was tried again but resulted in a failure. We adapted our algorithm to restart the bisection with the current load factor as the upper bounds upon these sporadic failures.

The bare-machine implementation produced very predictable results without any variation. The utilization of the benchmark was measured at 81%. The remaining 19% can be interpreted as the time consumed by the bare-machine implementation of Pthreads. Under UNIX, the benchmark utilization had its peak at 77% with a remaining 23% overhead due to the operating system and the layered Pthreads implementation. The smaller overhead of the bare-machine implementation can be attributed to the performance improvements discussed in the last section.

The occasionally large variations in the utilization under UNIX and the sporadic failures of the bisection algorithm seem to be due to operating system activities which occur at unpredictable times. These activities include process scheduling, CPU time accounting, and the processing of ethernet messages[1]. The unpredictability of the UNIX operating system limits its applicability for hard real-time systems. Hard real-time applications may not be able to safely achieve a high utilization under UNIX. A bare-machine implementation seems to permit a higher utilization for hard real-time applications, providing both predictability and an efficient use of the hardware.

6 Conclusions

We have implemented a sufficient subset of the Minimal Realtime System Profile to permit performance testing. The implementation supports preemp-

[1] There was no local hard disk attached to the SPARCengine. The only asynchronous activities were due to clock and ethernet interrupts.

tive priority scheduling, with a restricted form of priority ceiling emulation for mutexes. It supports a realtime clock with microsecond precision, and timed events with millisecond precision, including the timeout for the wait operation on a condition variable. Experience with this implementation suggests that Pthreads can be implemented in a form that is suitable for realtime applications with hard timing constraints.

The absolute performance figures are encouraging. The performance of the bare-machine implementation is much better than that of the version layered over a full UNIX system. Part of this improvement is due to our algorithm for saving register windows to memory, which is different from that used by the commercial UNIX operating system. The other big contribution to the performance improvement is that our implementation avoids most of the overhead of UNIX system calls. User code executes in the same virtual address space as the kernel. This means kernel service calls can be ordinary subprogram calls, or even in-line macro calls, rather than traps. We also eliminate the overhead of demultiplexing service requests in the UNIX system call trap handler. This improvement seems specific to the minimal realtime systems profile. Running kernel and user processes in the same virtual address space would be unacceptable for a full POSIX implementation.

The experiments performed support the hypothesis that a bare-machine implementation can achieve excellent predictability. This provides the ability of this system to support *a priori* schedulability analysis, much in contrast to unpredictable systems such as UNIX.

Next, we plan to port the PART Ada runtime system implementation to the bare-processor Pthreads implementation, and test both the absolute speed and the timing predictability of Ada. This may require extending the functionality of the present implementation in some respects. Handlers need to be written for some traps that generate synchronous signals. For example, a mem_address_not_aligned trap should be processed to generate a SIGBUS for the current thread. The current C library support also needs some extensions.

Efforts will be made to flesh out the implementation in other respects, including support for timer-driven round-robin scheduling, and some debugging support.

References

[1] F. Armand, F. Herrmann, J. Lipkis, and M. Rozier, "Multi-threaded Processes in CHORUS/MIX", *Proceedings of EEUG Conference* (Spring 1990) 1-13.

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

[3] T. Doeppner Jr., *A threads tutorial*, TR CS-87-06, Brown University, Dept. of CS (1987).

[4] Bill O. Gallmeister and Chris Lanier. "Early experience with POSIX 1003.4 and POSIX 1003.4a", *IEEE Symposium on Real-Time Systems*, IEEE Computer Society (1991) 190–198.

[5] E.W. Giering and T.P. Baker, "Using POSIX threads to implement Ada tasking: Description of work in progress", *TRI-Ada '92 Proceedings* (Nov 1992) 518–529.

[6] E.W. Giering, Frank Mueller, and T.P. Baker, "Implementing Ada 9x features using POSIX threads: Design issues", *TRI-Ada '93 Proceedings*, ACM (Sep 1993) 214–228.

[7] IEEE Portable Applications Standards Committee, *P1003.4a: Threads Extension for Portable Operating Systems (Draft 6)*, IEEE (Feb 1992).

[8] IEEE Portable Applications Standards Committee, *P1003.4a: Threads Extension for Portable Operating Systems (Draft 8)*, IEEE (Oct 1993).

[9] IEEE Portable Applications Standards Committee, *P1003.13: Information Technology - Standardized Applications Environment Profile - POSIX Realtime Application Support (AEP)* (Draft 5) (Feb 1992).

[10] Frank Mueller, "Implementing POSIX threads under UNIX: Description of work in progress", *Proceedings of the Second Software Engineering Research Forum* (Nov 1992) 253–261.

[11] Frank Mueller, "A library implementation of POSIX threads under UNIX", *Proceedings of the USENIX Conference* (Jan 1993) 29–41.

[12] M.L. Powell, S.R. Kleiman, S.Barton, D. Shah, D. Stein, and M. Weeks, "SunOS Multi-thread Architecture", *USENIX* (Winter 1991) 65-80.

[13] D. D. Redell et al., "Pilot: An operating system for a personal computer", *Communications of the ACM, Vol. 23, No. 2* (Feb 1980).

[14] D. Stein and D. Shah, "Implementing lightweight threads", *Proceedings of the USENIX Conference* (Summer 1992) 1–10.

[15] SUN Microsystems, Inc., *The SPARCengine 1E Card Family User's Manuals* Part No: 800-8137-02 (Apr 1990)

[16] 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", *Proceedings of the USENIX Conference* (Summer 1987) 185–197.

[17] Hideyuki Tokuda, Tatsuo Nakajima, and Prithvi Rao, "Real-Time MACH: towards a predictable real-time system", *USENIX MACH Workshop* (Oct 1990).

[18] U.S. Department of Defense. *Military Standard Ada Programming Language* ANSI/MIL-STD-1815A, Ada Joint Program Office (Jan 1983).

[19] Ada 9X Mapping/Revision Team, *Ada 9X Reference Manual: Draft Version 4.0*, Intermetrics, Inc., 733 Concord Avenue, Cambridge, Massachusetts 02138 *(available by anonymous FTP from ajpo.sei.cmu.edu)* (Sep 1993).

Availability of Source Code

The source code of the version of the Pthreads library layered over UNIX is available via anonymous ftp from `ftp.cs.fsu.edu` (128.186.121.27), in the file `/pub/PART/pthreads.tar.Z`. Other material (related publications) can be found in the same directory.