Using POSIX Threads to Implement Ada Tasking:
Description of Work in Progress

E.W. Giering III T.P. Baker

Department of Computer Science
Florida State University
Tallahassee, FL 32306-4019
Phone: (904)644-3441
Internet: giering@nu.cs.fsu.edu

Abstract

The draft standard Threads Extension to the POSIX Portable Operating System Interface provides an application program interface to services supporting the creation and execution of multiple threads of control sharing the address space and file descriptors of a single POSIX process. This paper reports on the progress of an effort to test the compatibility of POSIX threads and Ada by implementing Ada tasking using the proposed standard POSIX threads operations.

1 Introduction

The POSIX/Ada Real-Time (PART)[1] project has two main objectives: to accelerate development of standard Ada language bindings for the POSIX Realtime Extension [3] and Threads Extension [7]; to determine whether these interfaces are a suitable basis for the implementation of Ada tasking, by building a prototype Ada runtime system (RTS) using them.

The second of these goals is intended to provide feedback to the IEEE Pthreads working group. Rationale contained in Draft 5 of Pthreads makes it clear that it is intended to allow the efficient implementation of Ada tasking. However, it is difficult to account for all possible interactions with the Pthreads interface in the absence of an actual implementation. This paper reports on the progress toward such an implementation.

In Section 2 we provide some details about POSIX threads and some rationale for using them to implement Ada tasking. Section 3 outlines the scope of the problems addressed, and Section 4 outlines our approach to implementing an Ada RTS based on POSIX threads. Section 5 contains a discussion of the design of the RTS. Section 6 reports on the current status of the project, and presents some performance results. Finally, in Section 7 we outline future research and draw some conclusions from the work so far.

2 Background

POSIX is a family of standards for an application program interface to an operating system. The POSIX standards are being developed by the IEEE Technical Committee on Operating Systems (TCOS), and are being promoted as American and international standards. POSIX has its roots in an effort to promote application program portability, by establishing a non-proprietary standard interface to the many variants of the UNIX[2] operating system. Though POSIX still retains a UNIX flavor, its scope has been widened to provide a common application program interface that can be supported by virtually any operating system. The intent remains that an application that strictly conforms to the POSIX standards should be able to be compiled and executed on any operating system that conforms to the same standards.

The IEEE identifies POSIX standards by designations of the form 1003.x. For instance, 1003.1 designates the C language application program interface for core operating systems services, such as creating files, doing input and output, creating processes, and inter-process communication. It is the base POSIX standard, and has been approved by the ISO as ISO/IEC 9945-1: 1990. Other POSIX standards are extensions to this base.

[1]PART is funded by the Ada Joint Program Office under the Ada Technology Insertion Program, through the U.S. Army CECOM.

[2]UNIX is a trademark of UNIX System Laboratories, Inc.

Two other POSIX standards on which this project depends are 1003.4 and 1003.4a. Both of these exist only as drafts, which are currently in the IEEE ballot process. The 1003.4 interface (Realtime Extension) [3] is an extension to 1003.1 that provides services commonly needed in real-time applications. Examples of these services include binary semaphores, process memory locking, and timers. The 1003.4a interface (Threads Extension, or Pthreads for short) [7] is an extension to 1003.4 that supports multiple threads of control within a single POSIX process. Examples of services provided by 1003.4a include thread creation, mutual exclusion, and thread suspension. Both 1003.4 and 1003.4a are expressed as C language interfaces.

Where an operating system supports Pthreads, there are good reasons why an Ada implementation would want to take advantage of Pthreads as a basis for Ada tasks. The use of Pthreads to support tasking makes it easier to port the Ada implementation between systems which support the POSIX interfaces. It also has the potential of providing a user with more options, allowing the choice of Ada compiler and Pthreads implementation (and therefore task scheduling support) to be separated. Performance may improve, if Pthreads is supported by the OS kernel. System calls need only block the calling task, rather than the whole process. If global thread scheduling is provided, there may be better response to asynchronous events. If the operating system supports multiprocessing, throughput may also be better. Finally, it might be possible for the Ada application to make use of multi-threaded library services that are implemented in C, such as window and database interfaces.

3 Problem Statement

The rationale in Draft 6 of the Pthreads interface indicates that it is intended to support the implementation of Ada tasks [7]. Section 9.5.1 states that "It is fully intended that a one-to-one correspondence between Ada tasks and threads be possible, and that the capabilities provided here support efficient implementation of Ada tasking." It goes on to outline how various features of Ada tasking can be implemented using Pthreads, including task creation, activation, rendezvous, and abortion.

The idea of implementing Ada tasks as POSIX threads is part of the solution to the larger problem of providing an Ada binding to POSIX services. There is a standard Ada binding for 1003.1, namely 1003.5 [2]. This interface is defined as a set of packages, which provide access to the facilities of POSIX.1 via Ada data types, subprograms, and generics. Any attempt to extend this interface to include POSIX threads must address the functional overlap between POSIX threads and Ada tasks, and the potential for conflict between the Pthreads and Ada implementations in processing signals and scheduling the use of processing resources. A natural way to resolve these such semantic and implementation issues is to map Ada tasks onto POSIX threads.

Another issue that can be clarified by an implementation is the degree of interoperability between Ada tasking and Pthreads. Is it practical to allow an Ada application to use Pthreads primitive operations directly to affect the execution of Ada tasks, or should the RTS completely hide the low-level Pthreads interface from the application programmer? (This issue is discussed at some length in [11].)

Despite the fact that the Pthreads working group has apparently given some thought to making Pthreads suitable for the implementation of Ada tasking, it is not certain whether they have succeeded. Ada tasking semantics are fairly complex. It would be useful to the Pthreads working group to know the results of an implementation before the interface is standardized. This would aid in determining what changes, if any, are indicated to make Ada support practical.

4 Approach

4.1 Overview

Each Ada task corresponds to a POSIX thread. The Ada RTS exists as a set of library subprograms which use the Pthreads operations in such a way that the threads' behavior complies with the Ada semantics for tasks. The scheduling of the threads is directly under the control of the Pthreads scheduler, as is the state of each thread. Runtime stack allocation is also under the control of the Pthread implementation.

In many cases the mapping between an Ada task and the corresponding thread is straightforward. For example, Pthreads defines an optional standard for priority-based scheduling, derived from the process scheduling of the 1003.4 Realtime Extensions draft. This is required to support SCHED_RR and SCHED_FIFO scheduling policies, which are priority scheduling with and without time slicing, respectively. Both policies are consistent with Ada requirements: Ada priorities are static except during rendezvous, when a server task is raised to the priority of its client; Pthreads priorities are fully dynamic, allowing the Ada RTS to make this priority adjustment in the code implementing the **accept** statement. Other Ada features, such as the distinction between task creation and activation and the rules for task termination, are very different from their Pthreads counterparts, and must be implemented almost entirely by the Ada RTS.

4.2 Logistics

We had originally hoped to obtain an operating system which supported multithreaded processes, to which we would add enough C-language glue-code to present a Pthreads-compliant interface to the Ada RTS. We were unable to obtain such an operating system by project start, so we built our own user-level implementation of Pthreads [11,13]. This runs on a SPARCstation under the SunOS 4.0.3 operating system. [3]

In order to have a basis for comparison, we wanted to develop a Pthreads-based replacement for an existing commercial Ada implementation. The Verdix Corporation[4] made this possible by loaning us the source code of the 1.0 Sun Ada runtime system.

We considered two approaches to development of the Pthreads-based RTS. One was to modify the existing tasking code, and the other was to replace the tasking portion of the RTS completely. This latter course is made possible by the fact that the compiler does not directly access the kernel data structures used in implementing tasks, but always manipulates them by calling an RTS subprogram. In the end we decided on the latter approach.

All packages used to implement tasking in the original RTS have been replaced by new packages. The package interfaces are based on the Common Ada Run-Time System (CARTS) interface specification [8], with some modifications to match the RTS calls generated by the compiler. A discussion of the interface used by the Verdix compiler can be found in Section 2.5.4 of the Sun Ada Programmer's Guide [5].

[3] Sun, SunOS, and Sun Ada are trademarks of Sun Microsystems, Inc.; SPARC and SPARCstation are trademarks of SPARC International, Inc.

[4] Verdix is a trademark of Verdix Corporation.

5 Design of the Runtime System

In this section we report on the current status of the design of the Pthreads Ada tasking system. This design continues to evolve with our understanding of Pthreads and the execution environment provided by compiler-generated code. This description reflects changes to the design which we previously reported [11].

5.1 Pthreads Operations

To make the following exposition more understandable to readers with a limited knowledge of POSIX threads, we give a brief summary of some of the Pthreads operations used in our RTS. For details, refer to the Pthreads standard [7].

A POSIX thread is created in C using the *pthread_create()* function. The application passes a pointer to a function, called the *start* function, to *pthread_create()*; the Pthreads implementation then calls this function in a separate thread of control. Thus, unlike Ada task creation, the creation of a thread implies its becoming eligible to execute.

Mutual exclusion is provided through *mutexes*. When a thread wants exclusive access to some shared resource, it locks the associated mutex, via *pthread_mutex_lock()*; if some other thread has already locked that mutex, the requesting thread is suspended until the thread holding the mutex unlocks it, via *pthread_mutex_unlock()*. Any number of tasks can be suspended on the same mutex; one of them is granted the mutex and permitted to continue execution when the holder unlocks the mutex. Mutexes are similar to binary semaphores; the principal difference is that the thread which holds the mutex must be the one to unlock it. This makes mutexes difficult to use for general communication between threads; an arbitrary thread cannot signal to other threads that something has occurred by unlocking a locked mutex. For this kind of synchronization, *condition variables* are used.

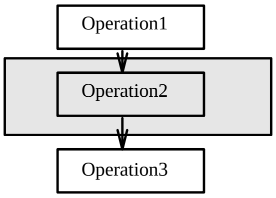
A thread waits for a condition to become true by calling *pthread_cond_wait()* on a condition variable. Another thread can signal that the condition has become true by signaling the condition variable, via *pthread_cond_signal()* (this is not to be confused with operations on POSIX signals [1]). A mutex is associated with the condition variable by the *pthread_cond_wait()* call; this mutex must be locked before the call, is unlocked (atomically) by the call, and is locked again when the call returns. This is to protect the condition for which the thread is waiting. A *pthread_cond_signal()* call is guaranteed to wake up at least one waiting thread, but it turns out to be more efficient (particularly on multiprocessors) to allow more than one waiting thread to return. Since the first thread to reacquire the associated mutex might make the condition false again, each thread needs to check that the condition is true when *pthread_cond_wait()* returns. This is usually done in a while loop (see example below).

POSIX threads are designed to be analogous to POSIX processes. For example, signals are delivered to individual threads in a multithreaded process using much of the same semantics as for delivery to a single-threaded process.

In the following sections, algorithms used to implement Ada tasking are illustrated using flow charts. For clarity, special graphics are used to illustrate some pthreads operations. The *pthread_mutex_lock()* and *pthread_mutex_unlock()* operations are not shown explicitly; instead, operations executed by a thread which has a mutex locked are enclosed in shaded polygons. For example, the code fragment

```
Operation1;
pthread_mutex_lock(mutex);
Operation2;
pthread_mutex_unlock(mutex);
Operation3;
```

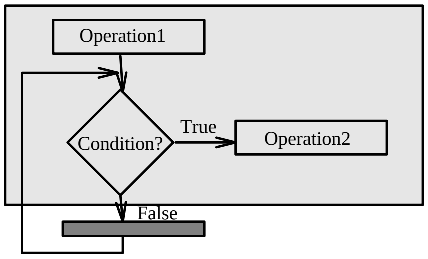
would be illustrated as



The *pthread_cond_wait()* operation is represented graphically by a small shaded rectangle. This rectangle is placed outside of the polygon representing points where the associated mutex is locked; the relationship to that mutex is evident from the fact that flow to and from the condition wait rectangle is from points where that mutex is locked. For example, the fragment

```
pthread_mutex_lock(mutex);
Operation1;
while not condition loop
  pthread_cond_wait(cond, mutex);
end loop;
Operation2;
```

would be illustrated as



5.2 Thread Specific Data and the TCB

Since there are Ada task features which are not directly supported by Pthreads, it is necessary to attach additional information to a thread implementing an Ada task. This information is kept in a record called the Ada Task Control Block (ATCB); a pointer to this block serves as the task identifier. This can be a problem, since the compiler expects a task to identify itself, but the corresponding thread has no direct means of identifying the task that it is supposed to represent.

The thread-specific data feature of Pthreads can be used to associate an ATCB pointer with a task. The *pthread_setspecific()* function allows each thread to associate a value with a key. For a given key, a different value can be associated by each thread. The thread can retrieve this value at any time by calling *pthread_getspecific()*. The ATCB pointer of an Ada task can be stored and retrieved as thread-specific data of the corresponding thread. However, since many tasking operations need the ATCB of the current task, *pthread_getspecific()* may be called very frequently. This can be potentially time-consuming.

To avoid frequent calls to *pthread_getspecific()*, we stored a copy of the ATCB pointer of a task at a fixed location in its stack, specifically just below the last usable stack location (see Figure 5.2). Since the stack limit must be available to allow checking for stack overflow, this allows a task to access its own ATCB in just a few machine instructions.

5.3 Task Creation and Activation

Unlike a POSIX thread, an Ada task may be created before it is activated. For an Ada task, the thread can either be created at the time of task creation, in which case it must suspend itself until the task is activated, or it can be created at task activation time, with task creation only allocating and initializing the ATCB [7].

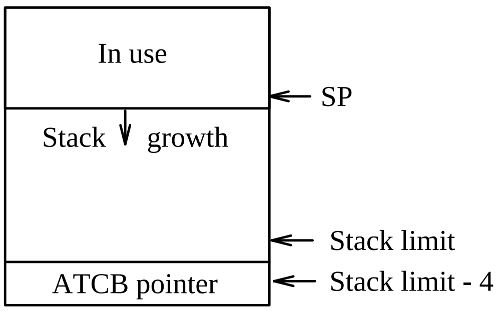


Figure 1: Ada task stack layout

Table 1: Partial ATCB layout

Task body address
Stack Size
Thread ID
Stack Limit
Mutex
Condition Variable
Activation list link

Thread creation at the time of activation appears to be easier to implement. However, this would mean that a task could be created when there was actually insufficient memory for the creation of a thread. This should be detected and raise STORAGE_ERROR at the point of task elaboration, but it would not actually be detected until *pthread_create()* was called at activation. To prevent this, we decided that a thread should be created when a task is created. The thread is then suspended on a Pthreads condition variable, until the creator activates the task.

The need to bind the ATCB to the thread constrained the design of task activation. The original plan for starting a thread was to pass the address of a "wrapper" function to *pthread_create()*, with a pointer to the procedure generated by the compiler for the task body as the argument to this wrapper. The wrapper calls the task body using this pointer. This satisfies the Pthreads requirement that the body of a thread be a function that takes a single pointer argument.

We ran into a problem binding the ATCB to the thread. A thread's thread-specific data and runtime stack are only accessible by the thread itself, so the creator of the thread would not be able to use either of these to bind the ATCB pointer to the new thread. The thread itself must bind the ATCB. The logical place for this is in the wrapper. However, the wrapper does not have access to the ATCB until it is bound; it can allocate the ATCB, but lacks certain values that are needed to initialize it.

We decided to use the ATCB to pass the needed information into the wrapper, since much of the information needs to be there anyway. The task creator allocates the ATCB and initializes it, passing the ATCB pointer to the wrapper via *pthread_create()*. It also returns this pointer as the task ID, something that would be difficult if the wrapper did the allocation. The wrapper then binds the pointer to the thread. This requires the address of the task body code and any other machine dependent information needed to run the task body to be placed in the ATCB; these fields are not used after thread initialization. This inefficiency in the use of ATCB space was considered preferable to the dynamic storage management requirements for an initialization structure, which we considered as an alternative.

A partial layout of the ATCB, showing those fields used to activate a task, is shown in Figure 1. A flow chart showing the basic algorithms used to implement task activation is shown in Figure 2. In some places, the mutex of both the creator and created tasks are locked simultaneously to keep the fields used for activation in them synchronized. To prevent deadlock, the creator's mutex is always locked first. The ATCB is allocated and its task body address and stack size are filled in by the creator (Create task). Pthreads passes a pointer to the ATCB to Wrapper, which fills in the thread ID and stack limit by inspecting its environment. The thread then suspends itself on the condition variable. At activation time, the creator traverses the list of tasks to be activated, counts them, marks them as ready to activate, and signals their condition variables. Each task being activated then proceeds to elaborate the declarative part of the corresponding task body. As each task completes its activation, it informs the creator by decrementing the count of activating tasks and signaling the creator. The creator waits on its own condition variable for this count to become zero, preventing it from continuing until the last activation has been completed.

5.4 Rendezvous

The basic algorithm used to implement rendezvous in our RTS is shown in Figure 3; this algorithm is the same as that outlined in a note to the Pthreads draft [9]. The figure illustrates a rendezvous between a parameterless entry call and an `accept` statement with a body. To keep the ATCB fields in both tasks synchronized they are protected by the mutex of the accepting task.

The calling task checks whether the target task is al-

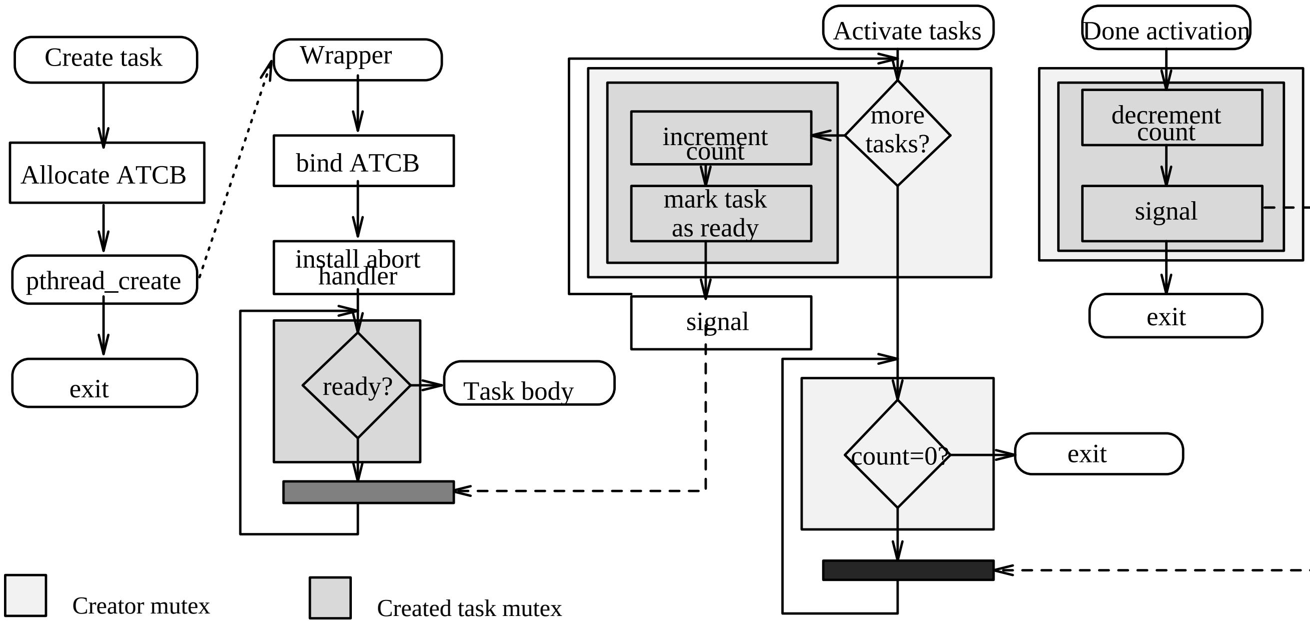


Figure 2: Task activation

ready waiting on the call using the `Accepting` flag in the acceptor's ATCB. If so, it sets the flag to false to block other calls to this entry and signals the acceptor's condition variable, allowing the rendezvous to proceed. Otherwise, it queues itself on the entry queue (in the acceptor's ATCB). In either case, it then waits on its own condition variable for the rendezvous to end. For its part, the acceptor checks to see if there is a call waiting on the entry queue. If there is, it proceeds to execute the body of the accept statement immediately; otherwise, it waits to be signaled by the caller before doing so. When the accept body is complete, the acceptor marks the rendezvous as completed in the caller's ATCB and signals the caller. Both tasks can then proceed. (The logic on the accepting side has been simplified in this diagram. In actuality, there is no accept procedure; instead, the compiler generates separate calls before and after the `accept` body.)

5.5 Interrupt Entries

POSIX signals act as a form of software interrupt, allowing a process to be informed of external events. These events can cause asynchronous termination or asynchronous transfer of control in the target process.

A process can choose to ignore or *handle* a signal. In addition, it can *mask* one or more signals, specifying that a signal of that type is to be held *pending* until the process *unmasks* it. Only one signal of each type is held pending; further signals delivered while the signal is masked are lost. The delivered signal, and other signals specified by the user, are masked before calling the handler.

In a multithreaded process, the handler specified for a signal applies to all threads, but signal occurrences are handled in the contexts of individual threads. There are fairly complex rules specifying which thread should be chosen. For the present discussion, it is only important that each thread has its own signal mask, and that a thread must have a signal unmasked to be chosen for delivery. An application can control which thread receives a signal by arranging for only one thread in the process to have the given signal unmasked at any one time.

In the C language specification of POSIX.1, a process can specify a function to be called on delivery of a signal. The POSIX.5 Ada bindings specifies the signal handling interface in terms of *interrupt entries*. This is based on the analogy of signals to interrupts. Ada only allows one interrupt entry to be active at a time for a given signal.

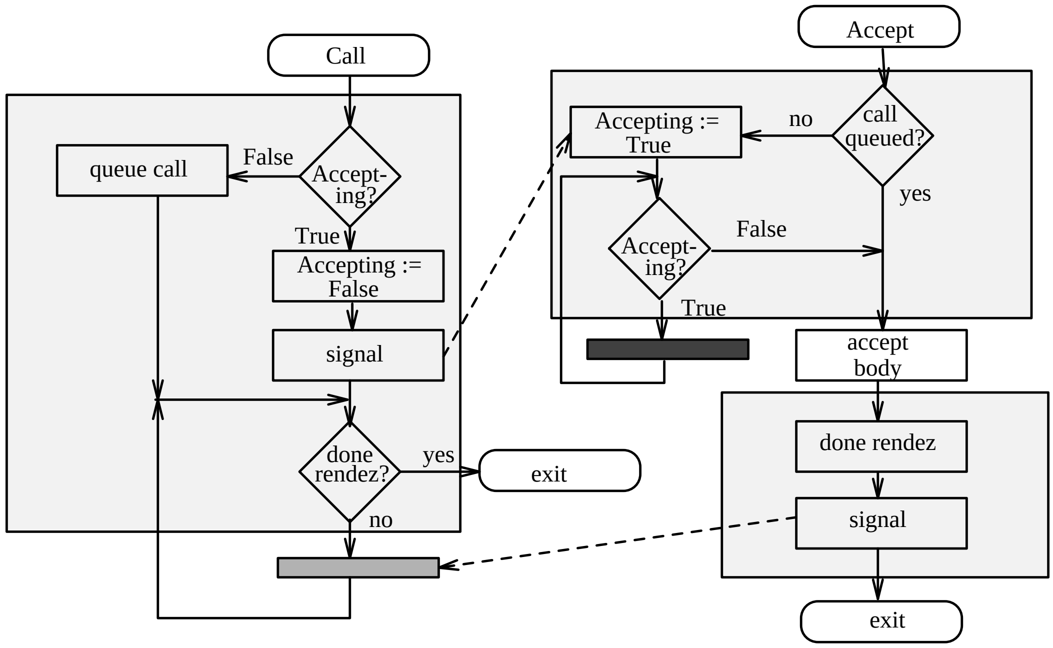


Figure 3: Simple rendezvous

POSIX.5 further specifies that a signal whose entry is not awaited is held pending, as if it were masked. This is consistent with a policy of keeping the signal masked until a call on the associated entry is can be accepted, and then only unmasking the signal within the thread of the associated task.

One naïve implementation of interrupt entries is to install a signal handler; this handler would then call the Pthreads RTS to make the entry call. However, such a handler is executed in the context of the thread to which the signal is delivered, and that thread cannot continue until the handler returns. When the handler suspended on the entry call, the task would deadlock.

One solution is for the RTS to create (internally) a separate task to serve as signal handler. This task waits for the signal using the Pthreads *sigwait()* function. This function takes a set of signals as an argument and atomically unmasks those signals and suspends the calling task. When one of the specified signals is delivered, all of the signals are masked, and *sigwait()* returns the value of the signal received. No signal handler is involved. The internal RTS task can then make an entry call to the user-specified interrupt entry without deadlocking. The control structure of the body of the internal RTS task might be:

```
loop
  sig := sigwait(signal_set);
  Call_Simple(Defining_Task,
      Entry_ID, Null_Parameter);
end loop;
```

Here, `Defining_Task` is the task that defines the interrupt entry, and `Null_Parameter` indicates that the entry has no parameters.

This implementation involves considerable overhead. A task is a fairly heavyweight object to use as a signal handler. It requires a separate stack and at least a partial ATCB. It also involves context switches that could be avoided if the accepting task could wait on the signal directly.

Another possibility is to use *sigwait()* in place of *pthread_cond_wait()* to suspend a task in an accept statement on an interrupt entry. When the signal is delivered to the thread, the *sigwait()* call would return, allowing the accepting task to proceed. Since an interrupt entry can appear in a `select` statement with normal entries or be called as a normal entry, there must be a way for other tasks to wake up a task waiting on such a *sigwait()*. They can do this by sending a signal to the task using *pthread_kill()*. A simplified version of the *accept()*

statement under this scheme might look like:

```
pthread_mutex_lock(Acceptor.Mutex)
Acceptor.Accepting := True;
pthread_mutex_unlock(Acceptor.Mutex);
sigwait(Signal_Set);
pthread_mutex_lock(Acceptor.Mutex);
if Acceptor.Accepting then
  -- code for regular entry call
else
  -- code for interrupt entry call
end if;
```

If the *sigwait()* returns because of a signal sent by another task, that task must set `Accepting` to `False` and put its task ID into the acceptor's ATCB so that the acceptor can signal the caller when it is done. This requires the acceptor's mutex to be unlocked. However, unlike *pthread_cond_wait()*, *sigwait()* does not atomically unlock a mutex when suspending a thread, so the `accept` statement code must do it before calling *sigwait()*. The caller can then lock the mutex, manipulate the acceptor's ATCB, signal the thread, and unlock the mutex.

A signal does none of this; it just causes the *sigwait()* to return. In the interval between the return of *sigwait()* and the *pthread_mutex_lock()*, there is nothing to stop a normal call from succeeding. The mutex is unlocked, and the `Accepting` flag is `True`. If a caller preempted at this point, it could set `Accepting` flag to `False`, signal the acceptor, and suspend. The acceptor would then see that a normal entry call had taken place and continue; the signal would have been lost.

The *pthread_cond_wait()* operation can also be used to wait for a signal. A side effect of handling a signal in the context of a thread waiting on a condition variable is to wake up the thread. The mutex is reacquired before calling the handler, so that the handler code is also executed under its protection. It remains locked after returning from the handler, and the thread is eligible to run. The code for the accept statement would look like:

```
pthread_mutex_lock(Acceptor.Mutex);
Accepting := True;
set_sigprocmask(SIG_UNBLOCK, Sigset);
while Accepting loop
  pthread_cond_wait(Acceptor.Cond,
      Acceptor.Mutex);
end loop;
set_sigprocmask(SIG_BLOCK, Sigset);
```

However, this implementation has a problem symmetric with that using *sigwait()*: *pthread_cond_wait()* atomically locks the mutex when suspending, but does not mask signals. If a signal is delivered in the interval between the time that the `Accepting` flag is checked and the time that the task is suspended, the wait will not return, since it was not in effect when the signal was delivered.

The current implementation uses this last approach, solving the lost signal problem using the C *setjmp()* and *longjmp()* functions. This is illustrated below:

```
Setjmp(Acceptor.Jmp_Env, Ret);
if Ret = 0 then
  while Accepting loop
    set_sigprocmask(SIG_UNBLOCK, Sigset);
    pthread_cond_wait(Acceptor.Cond,
        Acceptor.Mutex);
    set_sigprocmask(SIG_BLOCK, Sigset);
  end loop;
end if;
```

The signal handler looks like:

```
Accepting := False;
Longjmp(Acceptor.Jmp_Env);
```

The *longjmp()* will cause control to be returned to the point of the *setjmp()*, except that the value of `Ret` will not be zero. This will avoid calling the *pthread_cond_wait()* and allow the rendezvous to continue, eliminating the problem of receiving the signal before the task is suspended.

This is an implementation dependent solution; the C-language *setjmp()* and *longjmp()* functions work with the Ada compiler we used. For another compiler or operating system, machine code might need to be written to achieve a similar effect.

5.6 Termination

The termination of Ada tasks is complicated by a number of features, which are intimately connected with Ada's nested scope structure. In order to ensure that the task's environment persists so long as the task can execute, the concept of task *master* is introduced. The scope corresponding to a master is required to persist until all the tasks that depend on that master have terminated.

Termination is further complicated by selective wait statements. If a selective wait contains a terminate alternative, then a task waiting on that selective wait can terminate if its master is otherwise ready to terminate. Thus, a master can terminate when all of its dependents are either terminated or waiting on a terminate alternative. Note that up to the point of termination the selective wait can still accept an entry call, which would make the task no longer eligible for termination.

Pthreads lacks support for the Ada concept of task dependence hierarchy. Therefore, all operations involving task masters have to be handled by the Ada RTS, outside of the Pthreads implementation. Pthreads operations are only used as primitives, to suspend masters waiting for dependents to terminate.

Since the dependency hierarchy is distinct from other Ada hierarchies, it is usually implemented as a separate data structure created at run time. Our implementation uses an alternate algorithm, originally used in an implementation of the Model RunTime System Interface (MRTSI) [10].

The termination algorithm is based on the concept of a task's *parent*, which is defined as the task that elaborates the task's master. The parent task ID is recorded in the ATCB. Of course, the parent is not necessarily the master. However, the master must correspond to one of a sequence of nested scopes under the parent. At the point of task creation, there can be only one such master at each level of nesting, so the nesting level under the parent task uniquely identifies the master. For a master to be replaced with another master at the same level, it has to exit, which means that all of its dependent tasks have to terminate. The combination of the parent task and the nesting level of the master within the parent will therefore uniquely define a task's master for as long as the task exists. When a master completes, its parent task counts all of the "awake" dependents and then waits on a condition variable for this count to become zero. The dependents decrement this count as they terminate or begin waiting on selective wait statements with terminate alternatives, signaling the parent task. The parent task then counts all dependents not yet terminated (i.e. waiting on a terminate alternative), signals them, and waits for them to terminate in a similar manner.

5.7 Abortion

The Ada **abort** statement allows a task to cause other tasks (or itself) to terminate, subject to Ada termination processing. An aborted task becomes abnormal, but it can continue to run until it reaches one of the synchronization points defined in the ARM [4], which are roughly the places where the task could suspend. However, we have chosen to abort a task executing user code immediately, when permitted by Ada. Otherwise, a runaway task could not be aborted, if it is caught in an infinite loop that does not contain any of the operations specified as abortion synchronization points.

The operation provided by Pthreads to allow one thread to terminate another is *pthread_cancel()*. It does not provide for Ada task termination processing, but it does provide for *cleanup handlers* which a thread can specify to be executed when it is canceled. Cleanup handlers are installed by *pthread_cleanup_push()* and removed by *pthread_cleanup_pop()*; execution and removal is in LIFO order.

Our implementation places much of the **abort** processing in cleanup handlers. Unfortunately, cleanup handlers alone are not sufficient. When thread cancellation takes effect, the thread becomes eligible to execute its cleanup handler; it does not actually execute the handler until the thread is next scheduled. Some parts of Ada abortion must take place immediately, before the **abort** statement finishes execution. In particular, the target tasks must become abnormal and, if they are suspended waiting for a rendezvous, they must become completed and removed from the entry queue they are waiting on (if any). Our **abort** implementation performs these actions before cancelling the thread; the remainder of abortion processing is done by the thread itself in its cleanup handler.

There are situations in which abortion of a task must be deferred. Ada requires deferral of abortion while a task is in rendezvous. There are also critical sections within the Ada RTS, where RTS data structures are inconsistent, and it is not practical to restore consistency via a local cleanup handler. Finally, there are some Pthreads functions that are not *interrupt safe*; that is, if the thread is canceled while performing one of these operations the effect is undefined.

Deferral of thread cancellation can be achieved via the thread *cancellability state*[5]. A POSIX thread is always in one of three cancellability states: disabled, controlled, or asynchronous. In the asynchronous cancellability state, a thread may be canceled at any time. In the controlled cancellability state, cancellation is held pending until the thread's execution reaches one of certain specified cancellation points, which are generally points where the thread could suspend[6]. In the disabled cancellation state, cancellation is held pending until the

[5] The term "cancellability" is substituted here for the Pthreads term "interruptibility", to reduce confusion; these states refer only to cancellation and do not affect the response to hardware interrupts or signals.

[6] These are very similar to the Ada abortion synchronization points.

thread's cancellability state changes.

In our implementation, controlled cancellability is used to defer abortion. Ada tasks are in the controlled cancellability state while they are executing in the RTS, and are in the asynchronous cancellability state at all other times. There is no need to ever disable cancellability entirely, since the only time abortion is required to be deferred by Ada is when a task has made an entry call and the rendezvous is under way. The Ada RTS detects this case, and does not cancel the thread; instead, the thread checks itself for abnormality when the rendezvous completes.

Disallowing asynchronous cancellation within the RTS allows the RTS to be simpler. It can be argued that a runaway task is still assured of being aborted, since the RTS should not have any unbounded loops. It is only important that a task executing in the RTS be abortable at suspension points, which should be the only places where an RTS call can experience unbounded delay; these correspond to controlled cancellation points.

Termination of a task, including termination from cleanup handlers, involves locking the task's mutex. If the mutex is unlocked, or some other thread is holding it, there is no problem. However, it is erroneous for a thread to lock a mutex that it already holds. Thus, the cleanup handler of a terminating task must ensure that that thread has not already locked the mutex, before it starts termination processing. This is a problem, since Pthreads provides no way for a thread to test whether it already holds a mutex. The solution is to maintain the invariant that a thread never has its own mutex locked at a point where the cleanup handler can execute.

Since interruptibility is controlled throughout the RTS, and since *pthread_mutex_lock()* is not an interruption point, the task can lock and unlock its mutex without fear of cancellation. The one exception to this is *pthread_cond_wait()*, which is an interruption point, and which always leaves the mutex locked when canceled. The solution to this is outlined in Chapter 9 of the Pthreads draft [7]; that is, an additional cleanup handler is pushed before each *pthread_cond_wait()*, after the task's mutex has been locked, and is popped after the wait returns but before the mutex is unlocked. This handler will run before the abortion cleanup handler (in LIFO order), and, since it knows that the mutex is locked, it can unlock it unconditionally.

6 Results

The RTS implementation is complete enough to demonstrate the feasibility of implementing Ada tasks using Pthreads, and to expose some of the programming techniques and design tradeoffs involved. We have run the Chapter 9 portion of the ACVC 1.11 validation suite with our Pthreads-based tasking system, and all the required tests passed. This provides some degree of confidence that the essential semantics of Ada tasks have been implemented.

A further goal of this project is to measure how the use of the proposed standard Pthreads interface affects tasking performance. This cannot be done in absolute terms, since we cannot ensure that our implementation is the fastest possible using Pthreads, nor can we find a fastest-possible non-Pthreads implementation to use as a basis for comparison. However, comparing the performance of our RTS with that of the RTS originally provided with the compiler can give some indication of the relative cost of implementation Ada tasks as POSIX threads.

Some of the timings obtained from tasking related tests of the PIWG benchmark suite [12] are given in Table 2. These tests were compiled using `-O9` and `AUTO_INLINE` optimizations [6], and timeslicing was disabled. The first two columns represent the averages of 10 runs, run as the only user activity on a diskless (networked) SPARCstation SLC running SunOS 4.0.3. The times we obtained using our Pthreads-based RTS are shown under the heading "POSIX RTS". For comparison, the column labeled "Sun Ada RTS 1.0" contains times we obtained on the same machine using the original Sun Ada 1.0 runtime system. The column labeled "Sun Ada RTS 1.1" contains times provided to us by Verdix Corporation for Sun Ada 1.1 (which has an improved runtime system and code optimizer); these give an indication of the performance possible. Since we did not have access to Sun Ada 1.1, we have not been able to compile and test our Pthreads-based RTS against this new Verdix RTS.

In general, the performance with the non-Pthreads RTS is better than with the Pthreads-based RTS. We believe that this is due primarily to overhead imposed by going through the Pthreads interface, and the distributed overhead of the Pthreads implementation, which must support more general capabilities (such as per-thread signal delivery) than are needed to support Ada tasking.

A simple example of the overhead due to going through

Table 2: PIWG Benchmark Results — All times in μ seconds (see text for benchmark conditions).

Name	Description	POSIX RTS	s.d.	Sun Ada RTS 1.0	s.d.	1.1
C000001	Create & Terminate Task — Task Type used in Procedure	1816	3.8	1362	6.7	616
C000002	Create & Terminate Task — Task Declared in Procedure	1861	19.0	1354	5.5	634
C000003	Create & Terminate Task — Task in Declare Block of Main	1760	2.8	1382	4.6	645
T000001	Minimum Rendezvous - Entry call and return time	536	0.7	420	1.0	234
T000002	Rendezvous — Single library-level task, one entry	517	0.5	420	1.5	232
T000003	Rendezvous — Two library-level tasks, one entry/task	425	1.1	405	1.6	240
T000004	Rendezvous — Library-level task, two entries, in select	517	4.2	428	1.4	251
T000005	Rendezvous — Ten library-level tasks, one entry/task	422	0.8	375	1.5	241
T000006	Rendezvous — One library-level task, ten entries	685	7.0	456	1.8	281
T000007	Minimum Rendezvous — No accept body	380	0.4	283	0.9	127
T000008	Time to pass an integer through a buffer	1055	1.9	836	2.2	532

the Pthreads interface is the problem of boosting the priority of an accepting task to that of the calling task. With an Ada-only RTS, the priorities would most likely reside in the ATCBs of the respective tasks; one possible implementation might look like:

```
if Caller.Priority > Acceptor.Priority then
  Caller.Acceptor_Prev_Priority :=
      Acceptor.Priority;
  Acceptor.Priority := Caller.Priority;
  Requeue(Acceptor);
  Reschedule; -- or implied by return to user code.
end if;
```

In contrast, going through the Pthreads interface requires the use of an abstract data structure to communicate scheduling parameters to and from the application:

```
pthread_attr_init(Attributes, Result);
pthread_getschedattr(Caller.Thread,
    Attributes, Result);
pthread_attr_getprio(Attributes,
    Caller_Prio, Result);
pthread_getschedattr(Acceptor.Thread,
    Attributes, Result);
pthread_attr_getprio(Attributes,
    Acceptor_Prio, Result);
if Caller_Prio > Acceptor_Prio then
  Caller.Acceptor_Prev_Priority :=
      Acceptor_Prio;
  pthread_attr_setprio(Attributes,
      Caller_Prio, Result);
  pthread_setschedattr(Acceptor.Thread,
      Attributes, Result);
end if;
```

Even with in-line expansion of code for all the Pthreads calls (which our RTS does not do), using the Pthreads interface involves much movement of data between the internal Pthreads data structures, the abstract attributes structure, and the ATCB. There are numerous examples of this kind of mismatch between the services required by Ada and those provided by Pthreads.

Despite such mismatches, execution times with the Pthreads RTS are close to the same order of magnitude as with the non-Pthread RTS. Moreover, it may be possible to close this gap in performance slightly, by tuning the Ada RTS and the Pthreads implementation.

7 Conclusions and Future Work

We have so far encountered no problems with Pthreads that would make implementation of Ada impractical. We have found reasonable algorithms for all Ada task operations using the Pthreads primitives and some machine code. While performance may never equal that of an RTS based on an implementation of multiple threads tailored for Ada, the penalty is sufficiently small to make a Pthreads RTS attractive where other benefits are expected. However, to balance these optimistic findings, we feel compelled to inject some caution.

Regarding interoperability between Ada and C-language subsystems using Pthreads, it is important to understand that implementing Ada tasks as POSIX threads does not solve all the problems. No one should expect an Ada application to interface to off-the-shelf libraries written in C using Pthreads, blindly, without regard for what those C-language libraries are doing internally. There remain many ways to get into trouble. For example, there may be conflicts in the usage of signals between the Ada RTS and the C-language subsystem. For another example, problems are likely to arise if a C-language thread (that does not correspond to any Ada task) calls an Ada subprogram which then invokes the Ada RTS.

Regarding the reliability of our test results in predicting the performance penalty due to using Pthreads, it is important to understand that our Pthreads implementation is a pure "library" one. That is, it executes entirely as user code, without any special kernel support from the operating system. To reap some of the benefits of Pthreads, such as per-thread blocking of system calls and global scheduling of threads, kernel support would be necessary. However, a kernel implementation is likely to impose significantly greater overhead, due to the cost of switching in out of the kernel[7]. There is reason for concern that this overhead could be high; word-of-mouth reports suggest the slowdown might be as bad as a factor of three.

In the time remaining to this project, we intend to further tune the RTS to try to improve our estimate of its potential performance. We also intend to do some experiments to estimate the effect on performance if Pthread operations were implemented as OS kernel calls. We further plan to develop an application level binding to other Pthreads functions. This binding will contain all useful Pthreads functions which do not interfere with Ada tasking. While we will use our implementation as a guide, we hope to provide a binding that can be made safe in any "reasonable" implementation of Ada tasks in the presence of Pthreads.

In future work we hope to explore the suitability of Pthreads for implementing the proposed Ada 9X tasking extensions.

Acknowledgements

The preparation of this paper was funded by the Ada Joint Program Office under the Ada Technology Insertion Program, through the U.S. Army Communications Electronics Command, Software Engineering Directorate, subcontracted through the Telos Corporation. It was also partially supported by the Florida State University. Verdix Corporation provided the loan of their Ada compiler and the source code to its RTS, with the cooperation of Sun Microsystems.

Ideas expressed in this paper owe much to discussions with the members of the IEEE POSIX Ada binding working group, Offer Pazy, Pratit Santiprabhob, and Frank Mueller.

[7] These issues are explored at greater length in [11].

References

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

[2] *Portable Operating System Interface for Computer Environments, Ada Binding (Draft 8).* IEEE, March 1992. P1003.5/D8.

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

[4] *Reference Manual for the Ada Programming Language.* United States Department of Defense, Ada Joint Program Office, February 1983. ANSI/MIL-STD-1815A-1983.

[5] *Sun Ada Programmer's Guide.* Verdix Corporation, April 1991.

[6] *Sun Ada User's Guide.* Verdix Corporation, April 1991.

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

[8] Ted Baker. Requirements specification for the Common Ada Run-Time System. February 1991. Prepared for Westinghouse Electric Corporation.

[9] Ted Baker, Bob Conti, Offer Pazy, and Harold Seigel. An approach to implementing Ada rendezvous using Pthreads. October 1990. POSIX note P1003.4-N0261.

[10] T.P. Baker. A model runtime system interface for Ada. *ACM Ada Letters*, IX(1):84–132, January/February 1989.

[11] 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.

[12] Performance Issues Working Group. Piwg test suite. March 1992. Available through anonymous ftp from ajpo.sei.cmu.edu.

[13] Frank Mueller. A library implementation of POSIX threads under UNIX. In *Proceedings of the Second Software Engineering Research Forum*, November 1992. To appear.