Implementing Ada 9X features using POSIX Threads: Design Issues

Draft

E.W. Giering[1]
Computer Science Department
Languages Research Lab
New York University
New York, NY 10003

Frank Mueller T.P. Baker[2]
Department of Computer Science
Florida State University
Tallahassee, FL 32306-4019

Internet: giering@cs.fsu.edu Phone: (904)644-3441

[1]Partially funded by Air Force contract F08630-92-C-0019 through the GNAT project.
[2]Partially funded by the Ada Joint Program Office under the Ada Technology Insertion Program, through the U.S. Army CECOM.

Implementing Ada 9X features using POSIX Threads:
Design Issues

Draft

Abstract

The draft standard POSIX Threads Extension provides an application program interface to operating system services supporting the creation and execution of multiple threads of control within a single process. Where Ada is implemented over an operating system that supports POSIX-like thread services, there are good reasons to implement the Ada tasks as POSIX threads. This paper explores some of the design issues involved in implementing Ada 9X tasking constructs on a system supporting POSIX threads.

1 Introduction

We are developing a runtime system (RTS) library to support Ada 9X tasking. To enhance portability, this RTS is being designed to be able to use standard interfaces for operating system services. It will use a set of low-level primitives that can either be implemented directly on a bare machine, or mapped onto the services of an operating system that supports multiple threads of control within a single address space. The first such OS interface to be targeted is the POSIX 1003.4a Threads Extension. This paper outlines some of the technical issues involved in implementing the new tasking features of Ada 9X using the POSIX threads primitives, and some strategies being developed to address them.

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[3] 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 P1003.4a interface (Threads Extension, or *Pthreads* for short) [6] is an extension to the POSIX base standard that supports multiple threads of control within a single POSIX process. Examples of services provided by Pthreads include thread creation, mutual exclusion, and thread suspension. Pthreads is expressed as a C language interface. Design and implementation issues regarding Pthreads are discussed elsewhere [7, 8].

Ada 9X is a revision of the Ada language standard, which is currently in the form of a draft specification[1]. The existing Ada language standard, which Ada 9X will replace, was approved by the International Standard Organization in 1987, but has not changed significantly since 1983. We therefore call it *Ada 83*. Ada 9X retains the features of Ada 83 and adds some new features.

The use of Pthreads operations to implement the tasking operations of Ada 83, including task creation, activation, and termination, task execution, and rendezvous has already been explored elsewhere [3, 4, 5]. Therefore, in this paper we concentrate on some of the new features of Ada 9X, specifically protected objects and asynchronous transfer of control (ATC).

3 Synchronization Primitives

This section outlines some implementation strategies for protected objects using Pthreads primitives. It also identifies ways in which these strategies introduce overhead which is due to the choice of Pthreads as a lower layer instead of a kernel specifically designed for Ada.

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

3.1 Protected Objects

Protected objects provide synchronization based on a data object rather than a thread of control. They encapsulate private data and operations on that data. Protected operations can be procedures, functions, and entries. Calls to protected procedures and entries are executed in mutual exclusion; no other operation of the same protected object can proceed in parallel when a protected procedure or entry is being executed. The functions of a protected object can execute in parallel, but not when a protected procedure or entry of that object is executing. Functions are not permitted to affect the state of a protected object.

Unlike task entries, where actions are specified by inline **accept** statements, the actions associated with a protected entry are defined in the body of a protected object by an *entry body*. An entry body is very similar to a protected procedure body; both have a header, a sequence of declarations, and a handled sequence of statements. The principal difference between a protected procedure and a protected entry is the presence of a *barrier* associated with each entry body. When an call on a protected entry is executed, the barrier is evaluated; if it is `False`, the call is queued until the barrier becomes `True`. The evaluation of the entry barrier and the queuing of the entry call are also protected operations on the associated object. Barrier evaluation, protected entry queuing operations, and protected operation execution are collectively referred to as *protected actions*.

A queued entry call has precedence over other operations on the protected object. This is often explained in terms of the *eggshell model*. The lock on a protected object is the eggshell; to perform any protected operation, a task must enter the eggshell. One protected procedure or entry call or several protected function calls can be active in the eggshell at a time. Tasks attempting to enter an eggshell which is occupied by a procedure or entry call, or tasks attempting to enter an occupied eggshell for a procedure or entry call, will be blocked.

An entry call must enter the eggshell to evaluate the associated barrier. If the barrier evaluates to `True`, the entry body is executed inside the eggshell; otherwise, the call is queued within the eggshell. Since queued entry calls are not active, other calls can enter the eggshell in which they are queued.

Queued entry calls become eligible to execute when their barriers become `True`. This need only be checked when they become true as the result of a protected procedure or entry call on the same object [1], essentially treating the barrier expressions as though they depended only on the state of the protected object. Therefore, before an operation that may have changed the state of a protected object exits the eggshell, any queued entry calls waiting on barriers which now evaluate to `True` must be executed. Only when the barriers of all entries with queued calls are `False` can the eggshell be exited. This assures that all entry calls made eligible by a state change are executed before any further operations are initiated.

3.2 Mutexes and Condition Variables

Pthreads provides synchronization primitives with similar capabilities to protected objects. Mutual exclusion is provided by operations on *mutex* objects. Before executing a critical section, a thread attempts to lock an associated mutex using *pthread_mutex_lock()*. Attempting to lock an already locked mutex blocks the calling thread. On leaving the critical section, a thread unlocks the mutex with *pthread_mutex_unlock()*; if there are threads blocked on the mutex, one of them will be given the mutex and allowed to proceed. Unlike protected operations, the critical sections and locks are not associated syntactically, and unlocking does not have to be done in LIFO order.

Pthreads allows a task to voluntarily suspend itself waiting for a condition through the use of objects called *condition variables*. A thread checks the condition under the protection of a mutex. If the condition is false, the thread calls *pthread_cond_wait()* function with the mutex and associated condition variable objects. The thread is suspended and the mutex unlocked atomically. A thread that may have changed the value of the condition calls *pthread_cond_signal()* on the associated condition variable. If there are any threads waiting on this condition, at least one of them is awakened. The awakened threads contend for the associated mutex. As each thread is given the mutex, it checks the condition again, since it must assume that other waiting threads may have executed first and may have changed the condition under the protection of the mutex. If the condition is true, the thread can then proceed. Unlike protected entries, the condition, the lock protecting it, and the suspension control object are not associated syntactically.

For example, to wait for *condition* to become true under the protection of *mutex* using *cond*, a thread could execute the following code fragment:

```
pthread_mutex_lock(mutex);
```

```
    while !condition {
      pthread_cond_wait(cond, mutex);
    }
    /* mutex is locked at this point */
```

3.3 A Naïve Implementation

The functional similarity of Pthreads synchronization to that provided by protected objects would seem to make a simple direct mapping of protected objects over Pthreads possible. For example, consider the following protected object implementation of a secure wait (i.e. exactly one waiter wakes up for each signal) with a persistent signal (i.e. a signal is not lost if it is not awaited):

```
protected P is
  procedure Sig;
  procedure Clear;
  entry Wait;
private
  Signals : Integer := 0;
end P;

protected body P is

  procedure Sig is
  begin
    Signals := Signals + 1;
  end Sig;

  procedure Clear is
  begin
    Signals := 0;
  end Clear;

  entry E1 when Signals > 0 is
  begin
    Signals := Signals - 1;
  end E1;

end P;
```

This appears very similar in functionality to the following C implementation:

```
int Signals := 0;

void Sig()
{
  pthread_mutex_lock(mutex);
  Signals := Signals + 1;
  pthread_cond_signal(cond);
  pthread_mutex_unlock(mutex);
}

void Clear()
{
  pthread_mutex_lock(mutex);
  Signals := 0;
  pthread_mutex_unlock(mutex);
}

void Wait()
{
  pthread_mutex_lock(mutex);
  while Signals <= 0 {
    pthread_cond_wait(cond, mutex);
  }
  Signals := Signals - 1;
  pthread_mutex_unlock(mutex);
}
```

However, there is an important semantic difference between these two implementations. Consider the case where there are one or more waiters, and a call is made to `Sig`. While `Sig` is executing, a call is made to `Clear`. This call will be blocked, since `Sig` is executing in the eggshell (holds the mutex). On exit, the call to `Sig` will check the barrier expressions (signal the condition variable).

At this point, the two implementations differ. The protected object will execute one of the waiting entry calls, since their barriers have become true. The call to `Clear` will remain blocked trying to enter the eggshell until all eligible entry calls have been serviced. In the Pthreads implementation, the *pthread_cond_signal()* call in `Sig` will wake up one of the waiters in `Wait`. However, this waiter will then have to contend with the caller of `Clear` for the mutex. When `Sig` calls *pthread_mutex_unlock()*, which thread will actually get the mutex will depend on the Pthreads implementation. If the optional priority scheduling is implemented, the highest priority thread will get the mutex, and this could be the one calling `Clear`. In this case, the signal will have been lost, and the waiting thread will continue to wait.

The difference in semantics between protected objects and mutexes and condition variables makes it difficult in general for protected objects to be mapped directly to Pthreads operations. In particular, it is hard to use the queuing inherent in condition variable semantics to implement the queuing provided by protected entries. To use this queuing, a task must wait on a condition variable. It must give up the mutex representing the eggshell to do this, otherwise no other protected actions would be able to proceed, and the protected object would be deadlocked. This constitutes a breech in the eggshell, since the task waiting on the entry call will have to contend for the mutex again when it is awakened, and there is no way to differentiate between a task attempting to enter the eggshell and one waiting on an entry call within the eggshell. Both are attempting to lock a mutex, and both will be treated the same by the Pthreads implementation.

Solutions to this problem require that either Pthreads or Ada treat the tasks queued on entry calls specially. There is no good way to force Pthreads to transfer a

Model	Self-Service	Proxy
parallelism	exiting task ‖ one entry body	exiting task and all entry bodies serial
latency of exiting task	unbounded	bounded
synchronization	lock ownership must transfer	unaffected by lock ownership
mapping to Pthreads	difficult: cannot transfer lock ownership	easy: proxy already owns lock

Table 1: Comparison of Implementation Options for Protected Objects

mutex to one group of waiting tasks in preference to another[4]. For the RTS to arrange for the waiting tasks to have preference, it must know which tasks are waiting, which requires it to implement its own queue. This means queuing must be supported on two levels of the overall implementation. Support for queuing more than one task on a condition variable, which is provided by the Pthreads implementation, becomes pure overhead since it does not contribute to supporting protected objects.

Another problem with using the queuing inherent in condition variable semantics to implement entry queues is the default queuing discipline provided by the two standards. The Ada default is FIFO, and this default is supported even when priority scheduling is provided. Other queuing disciplines must be specified by pragma. Pthreads specifies that threads waiting on condition variables be queued according to the scheduling queuing discipline, for which no default is specified. There is no portable way to support FIFO queuing. Supporting FIFO queuing for condition variables while scheduling by priority violates Pthreads semantics.

Even if these problems could be solved, the use of condition variable queues to implement protected entry queues, while it saves queuing resources, is suspect on efficiency grounds. In the simple example above, only one task is awakened by the exit from the eggshell, since there is only one entry. In a protected object with many entries, a waiter on each entry would have to be awakened to check the associated barrier. If few or none of the barriers had become true, this would involve many unnecessary context switches as each of these tasks wakes up, checks its barrier, and suspends again.

[4] It is possible to force a thread to be given a mutex by raising its priority over that of the other contenders. However, this can lead to unnecessary context switches and degrades the implementation of Ada priority scheduling over Pthreads.

3.4 Implementing Protected Objects with Explicit Queuing

Since the Ada implementation cannot take advantage of the queuing implicit in condition variable semantics, it is essentially forced to use *pthread_cond_wait()* as a single task wait, and *pthread_cond_signal()* to wake up specific tasks. To avoid the extra context switches involved in waking up queued tasks simply to evaluate barriers to `False`, the Ada RTS can evaluate all entry barriers as part of the eggshell exit sequence and choose the next task to execute.

The current Ada definition does not specify which task executes an entry body ([1], 9.5.3(21)). Two obvious candidates are the task that called the entry and the task that causes the barrier of the entry to become `True`. We refer to these as the *self-service model* and the *proxy model* of protected entry execution, respectively.

In the self-service model, a task exiting a protected object chooses a task waiting on a `True` entry barrier (if any) based on the entry queuing discipline in effect and wakes it up, making it the active task within the eggshell; whether there is any explicit transfer of ownership will depend on the underlying implementation of mutual exclusion. The exiting task continues with its post-call processing, and the newly awakened thread executes the entry body. When the task has completed the entry body, it will repeat the check on queued entry calls. This process will continue until all eligible calls have been executed, at which point the task actually exits the eggshell, allowing new callers to enter.

In the proxy model, the task exiting the eggshell executes the entry body itself, wakes the calling task, and then repeats the check for waiting callers, finally exiting the eggshell when there are no more entry calls waiting on `True` barriers. This is the model explored in detail by the Ada 9X Mapping/Revision Team in their implementation modules [9, 10].

A summary of the advantages and disadvantages of the two models is shown in Table 1. The principal advantages of the self-service model are that it permits (slightly) more parallelism and simplifies schedulability analysis. Parallelism is increased because the exiting task can proceed with its own execution in parallel with execution of the next queued call. On a single processor and assuming priority-ceiling locking (see 3.7) this would make no difference, but it could make a difference on a multiprocessor. Schedulability analysis is compromised by the proxy model since a task attempting to exit an eggshell must first execute all of the waiting entry calls whose barriers are open. There is no language-imposed limit on the number of such calls that can be pending. Therefore, to predict the worst-case execution time of a protected action, one must analyze logic of the potential entry callers and the internal logic of the specific protected object to include the total execution time of all the calls that might ever need to be served by proxy. In contrast, the self-service model would allow a thread to continue with its own execution immediately after the (presumably bounded) time it takes to complete the body of its own protected operation and pass lock ownership to the next queued caller.

The principal advantage of the proxy model is simplicity. If an entry body cannot be executed immediately, the calling task has only to suspend and wait. Since some other task will be responsible for executing the entry body before the caller returns from the wait, the caller does not have to arrange to execute the body on its return from the wait. More complex features of protected objects, such as timed and asynchronous entry calls, are simplified even more by this model. It is also more tolerant of different synchronization primitives, allowing the use of owned or unowned locks (see below). Assuming protected objects are used in such a way that large numbers of entry calls are not queued at the same time, or the latency of any particular protected action is not important, it can be argued that the simplicity of the proxy model outweighs the benefits of the self-service model.

3.5 The Self-service Model and Pthreads

Using Pthreads to implement the self-service execution model has many of the same problems encountered with the direct mapping approach outlined in Section 3.3. A mutex is owned by the thread that locks it, in the sense that the same thread must also unlock it. This means that the task attempting to exit an eggshell must either be able to transfer ownership to a task waiting on an entry call, or must hold the mutex on behalf of that task until it (and all other tasks waiting on `True` barriers) are finished. As pointed out in Section 3.3, there is no good way to achieve the effect of transferring mutex ownership with Pthreads.

It would be possible to build locking and suspension primitives which allow the transfer of ownership or in which locks are not owned. However, this begs the question of how best to implement protected objects using the Pthreads primitives, and generally moves the problems involved in supporting the self-service model down to the implementation of these new primitives. The option of having the task that first locks the mutex hold it until all eligible waiting tasks have executed their entry bodies forfeits the advantages of the self-service model, since the thread exiting the eggshell is still delayed for the amount of time that it takes to execute the waiting entry bodies and does not execute in parallel with any of them. For these reasons, the proxy model seems to be the most practical choice for implementation over a Pthreads platform.

3.6 Implementing the Proxy Model

This section provides an implementation sketch of the proxy execution model over Pthreads. This sketch is for simple entry calls only; there is no support for timed, conditional, or asynchronous entry calls. Such support would require a more complex transformation, and would probably be done through a more complex RTS interface. Limiting the examples to simple entry calls has the advantage that a relatively simple transformation can be shown using Pthreads calls directly.

We assume that the following operations are available for queuing tasks:

```
with Task_IDs; use Task_IDs;
package Task_Queues is
  type Queue is private;
  procedure Enqueue (Q : in out Queue; T : Task_ID);
  function  Enqueued(T : Task_ID) return Boolean;
  function  First   (Q : Queue) return Task_ID;
  procedure Empty   (Q : Queue) return Boolean;
  procedure Dequeue (Q : in out Queue);
  function  Count   (Q : Queue) return Integer;
private  ...
end Task_Queues;
```

The `Enqueue` operation adds the task `T` to the queue `Q`. The `Enqueued` function returns `True` if the task is on any queue, and `False` otherwise. `First` returns the first task in the queue, that is, the next to be considered for processing; which task is first depends on the queuing discipline in effect. The `Empty` function returns `True` if `Q` is empty; `False` otherwise. `Dequeue` removes the first task from `Q`. The `Count` function returns a count of the number of tasks on a queue; this is used to implement the `'Count` attribute of protected entries.

In addition, we assume the presence of operations that can be used to manipulate exceptions. `Current_Exception` returns a value which identifies the exception currently being handled, and `Raise_Exception` takes a value of this type and raises the corresponding exception. There is also a distinguished value `No_Exception` for which `Raise_Exception` has no effect.

We give some transformations for the example shown in Section 3.3. The complete transformation of this object is shown in Appendix A.

Each protected object declares a mutex `Lock`, used to implement the eggshell[5], and a procedure `Service_Entries`, in addition to transformations of user declarations. Transformation of procedures is straightforward. For example, the procedure `Sig` would be implemented by:

```
procedure Sig is
begin
  pthread_mutex_lock(Lock);
  Signals := Signals + 1;
  Service_Entries;
exception
when others =>
  Service_Entries;
  raise;
end Proc;
```

The procedure `Wait` implements a call to the entry of the same name, entering the eggshell by locking the mutex of the protected object and evaluating the entry barrier. If `True`, it executes the entry body sequence of statements and exits the eggshell. If `False`, it records which entry the current task is calling, queues the call, and then waits on the task's condition variable, unlocking the protected object's mutex (`P.Mutex`) in the process. When awakened, it will again own the mutex as a side effect (in this case undesirable) of the return from the *pthread_cond_wait()* call. It must unlock this mutex before it continues, completing the entry call.

```
procedure Wait is
  E : Exception_Type;
begin
  pthread_mutex_lock(Lock);
  if Signals > 0 then
    Signals := Signals - 1;
  else
    Enqueue(Entry_Queues(1), Self);
    while Enqueued(Self) loop
      pthread_cond_wait(Self.Cond, Lock);
    end loop;
    E := Self.Exception_to_Raise;
    Self.Exception_to_Raise := No_Exception;
    Raise_Exception(E);
  end if;
  Service_Entries;
exception
when others =>
  Service_Entries;
  raise;
end Wait;
```

Whenever a protected action that may change the state of the protected object completes, it exits the eggshell by calling `Service_Entries`. This procedure searches the queue, executing the bodies corresponding to any calls for which the entry barrier evaluates to `True`. Each time such a call is processed, it is dequeued and the associated task is awakened. The search for entry calls waiting on `True` barriers must then be repeated, since the preceding call may have opened the barrier of some entry that was previously closed. This process iterates until there is no non-empty queue with a `True` barrier.

An exception in a barrier evaluation should raise `Program_Error` in all tasks waiting on an entry in this protected object. This is not shown in this example, for simplicity of presentation.

```
procedure Service_Entries is
  Temp_Barrier : Boolean;
  State_Changed : Boolean := True;
  T : Task_ID;
begin
  while State_Changed loop
    State_Changed := False;
    for Entry_Index in 1..1 loop
      T:= First(Entry_Queues(Entry_Index));
      if T /= Null_Task then
        case Entry_Index is
        when 1 => Temp_Barrier := Signals > 0;
        end case;
        if Temp_Barrier then
          State_Changed := True;
          case Entry_Index is
          when 1 =>
            begin
              Signals := Signals - 1;
            exception
            when others =>
              T.Exception_to_Raise := Current_Exception;
            end;
          end case;
          Dequeue(Entry_Queues(Entry_Index));
          pthread_cond_signal(T.Cond);
        end if;
      end if;
    end loop;
  end loop;
  pthread_mutex_unlock(Lock);
end Service_Entries;
```

Each task waits on its own condition variable, and it is the only task that can wait on it. As mentioned in 3.3,

[5] Note that implementing the eggshell with a mutex does not permit parallel execution of protected function calls within the same protected object. To do this would require the implementation read and write lock primitives using mutexes and condition variables. This overhead would probably be of minimal value on a multiprocessor implementation of Pthreads, and of no value on a uniprocessor implementation.

the full generality of the *pthread_cond_wait()* primitive, which permits a task to wait on different conditions and permits several tasks to queue on a single condition, is not of any help here. A single-thread `Wait` with a corresponding `Signal` would be adequate for this approach to implementing protected objects.

3.7 Priority Ceilings

The Ada 9X Real-Time Systems Annex specifies a priority-ceiling locking policy for protected objects.

Pthreads also defines priority-ceiling mutex locking, as one of several options which a Pthreads implementation may support. If this option is supported, it can be used by the Ada runtime system for the mutexes used to implement protected objects. However, priority ceiling locking for Pthreads mutexes is inherently harder to implement than for protected objects. One reason is that mutexes are not required to be locked and released in LIFO order. Another reason is that the an implementation of Pthreads is required to also support general priority inheritance (a more complex scheme than ceiling locking) if it supports ceiling locking. A third reason is that Pthreads requires that a task be able to lock a mutex that has a lower ceiling than its current priority.

The overhead of supporting this unneeded generality in Pthreads is paid by the Ada implementation, even though it is not used. On the other hand, since Pthreads does not check that a task does not attempt to lock a mutex with a lower ceiling than the task's active priority, the Ada RTS must do this check. Since Pthreads does not provide an operation to interrogate the active priority of a thread, the Ada RTS must keep track of active priorities itself. This duplicates work that is being done already by the Pthreads implementation—another source of overhead.

If the Pthreads implementation does not support priority-ceiling locking of mutexes, the Ada RTS will need to use Pthreads operations to explicitly adjust the priority of a task when it enters and exits the eggshell of a protected object. These operations are considerably more general than required for priority ceiling locking. They are likely to be implemented as kernel calls, in which case they would add overhead that may be well in excess of the rest of the cost of a protected operation.

4 Asynchronous Transfer of Control

This section describes some strategies for using Pthreads primitives to implement asynchronous abortion of tasks, and the Ada 9X asynchronous **select** statement. It also mentions aspects of the Pthreads primitives that are likely to impose extra overhead on the Ada implementation, beyond the inherent cost of implementing the Ada semantics.

4.1 Asynchronous Select and Abortion

Another new language construct proposed for Ada 9X is the asynchronous **select** statement which provides the ability of asynchronous transfer of control (ATC). The syntax of the asynchronous **select** statement is as follows:

```
select
  <triggering_stmt> [<triggered_stmts>]
then abort
  <abortable_part>
end select;

<triggering_stmt> ::=  <entry_call_stmt> | <delay_stmt>
```

The triggering statement is either an entry call or a delay statement. The triggered statements and the abortable part are arbitrary sequences of statements. The asynchronous **select** statement can viewed as executing the abortable part and the triggering statement in parallel. Whichever one completes first causes the other to be aborted. This provides local abortion limited to the abortable part, which is potentially cheaper than the abortion of an entire task.

Slightly simplified, the dynamic semantics are:

1. The triggering statement (entry call or delay) executes.
2. The abortable part starts before the triggering statement completes.
3. If the triggering statement completes first, the abortable part is aborted, an ATC is performed, and the triggered statements are executed.
4. If the abortable part completes first, the triggering statement is canceled and the triggered statements are not executed.

The asynchronous **select** statement can be nested. The abortable part may itself contain another asynchronous **select** statement. In this case, the ATC may have to propagate to an outer asynchronous **select** scope if its triggering statement completes. In the process, all inner asynchronous **select** statements have to be aborted.

All forms of abortion, including ATC, must not take effect while a task is in an abort-deferred state. Such states include execution of a protected action and execution of a rendezvous call amongst others.

Example. ATC is useful for programming time-limited or interruptible computations. The following example depicts an interactive command-line interpreter [1]. A command is read from a terminal, and then executed. If an interrupt signal is sent from the terminal, the execution of the command is immediately aborted.

```
loop
  select
    Terminal.Wait_For_Interrupt;
    Put_Line("Interrupted");
  then abort
    Put_Line("-> ");
    Get_Line(Command, Last);
    Process_Command(Command(1..Last));
  end select;
end loop;
```

Ada 9X allows abortion (and thereby ATC) to be deferred until the next abortion synchronization point, much like abortion of a task in Ada 83 [1]. Real-time systems are more sensitive to latency in abortion and other operations, so the Ada 9X Real-Time Annex requires that abortion be acted upon immediately [2]. In this discussion, we assume abortion must be immediate if the threads layer, on which the Ada runtime system is built, provides some form of ATC. However, for portability, the design of the Ada implementation should not rely on the threads layer providing ATC. It should also check for abortion at the required synchronization points.

4.2 Asynchronous Pthreads Operations

Pthreads provides two features that are capable of asynchronously interrupting the execution of a specific thread. One of these is thread cancellation. The other is per-thread delivery of signals.

Cancellation is a way of killing a thread. In Pthreads, the interface for thread cancellation is *pthread_cancel()*. A thread may control its susceptibility to cancellation by modifying its *interruptibility state* and *interruptibility type*. The interruptibility state tells whether interruption of the thread for cancellation is enabled or disabled. The interruptibility type determines the kind of interruption that is permitted. By default, the interruptibility type is PTHREAD_INTR_ASYNCHRONOUS, which means cancellation is performed asynchronously and may interrupt any operation. To provide protection for unsafe functions against cancellation, an alternative interruptibility type PTHREAD_INTR_CONTROLLED is provided which defers cancellation until the next interruption point[6]. The interruption points are essentially all the places where a thread might suspend, except for locking a mutex. A mechanism for thread finalization is provided, in the form of a per-thread stack of cancellation *cleanup handlers*. A thread can push and pop cancellation cleanup handler functions on this stack. When a thread is canceled, the stack of cleanup handlers installed at this point are executed before the thread terminates.

The other asynchronous facility provided by Pthreads is signal delivery. A signal is conceptually similar to an interrupt. A handler function is specifiable for each signal, at the process level, so that all threads execute the same handler function for the signal. Each thread has its own signal mask, which allows the thread to postpone delivery of the signal. A second occurrence of a signal can be lost if the signal is masked and there is already an occurrence of that signal pending for the thread. A signal occurrence may be directed at a specific thread using *pthread_kill()*. Alternatively, signals can be directed at the process, in which case the signal is delivered to any thread that has the signal unmasked. Signals caused or requested by a thread are always directed at the same thread. For example, this is true of all the synchronously generated signals, the timer signals, and the asynchronous I/O signals. When a signal is delivered to the thread, the current operation is interrupted, the signal handler is executed, and the execution of the interrupted operation resumes after completion of the signal handler.

An ATC can be performed from a signal handler by a non-local jump via *longjmp()*. The long jump is a C-language operation which transfers control from the current point of execution to the point of a previous call to *setjmp()* at a lower nesting level of stack frames. Specifically, *setjmp()* can be called as the condition of an if-statement. In essence, *setjmp()* saves the the state of the thread at the point of call into a *jump buffer* variable and returns the value zero. The *longjmp()* operation reloads the state from the jump buffer which transfers control to the *setjmp()* call causing *setjmp()* to return again, but this time with a non-zero value. Thus, the

[6] The concept is similar to Ada abortion synchronization points.

model	one-threaded	two-threaded
mapping to Pthreads	moderately difficult: abortion signal	easy: cancel thread
task control block	simple: local to thread	complex: shared between threads
code generation	inline	call-backs
code optimization	easier	may be limited
general overhead	small: signal handler, *longjmp()*	considerable: create/terminate thread
interrupt safety overhead	*either* small: software flag *or* costly but transparent: signal masking	easy: controlled interruptibility

Table 2: Comparison of Implementation Options for ATC

longjmp() can be used to roll back a computation to an earlier conditional branching point, and take the other branch.

Some POSIX functions, including the majority of Pthreads functions, are declared to be *unsafe*. If an unsafe function is interrupted, the behavior of any call to another unsafe function is undefined until the interrupted function completes. Apparently, this applies to asynchronous thread cancellation as well as signal handlers. In particular, non-local jumps via *longjmp()* and the Pthreads functions that suspend a thread are unsafe (with the exception of *sigsuspend()*). If a signal interrupts an unsafe function, only safe (reentrant) functions can be executed inside the signal handler. Otherwise, the effect of any subsequent execution is undefined. Similarly, if thread cancellation interrupts an unsafe function, the effect of any subsequent execution is undefined.

4.3 One *versus* Two Thread Implementation

The asynchronous **select** may be implemented using a single thread of control or two separate threads of control, one for the triggering statement and one for the abortable part.

When the execution of the triggering statement reaches a point where it cannot continue execution—because it must wait for a protected entry barrier to open, a task to accept an entry call, or a **delay** to expire—the execution of the abortable part must be started. At this point, the two implementation models differ.

- *One-threaded model*: The current thread is only *marked* as suspended in the triggering statement but continues by executing the abortable part.
- *Two-threaded model:* A new *secondary thread* is created as part of the execution of the asynchronous **select** statement. The primary thread executes the triggering statement and suspends if necessary. Meanwhile, the secondary thread executes the abortable part.

With the *one-threaded model*, ATC can be implemented using a signal and *longjmp()*. The trigger (entry call or delay) is pending on the thread while the abortable part is executed. If the abortable part completes first, the pending trigger is removed. If the trigger completes, an abortion signal is sent to the thread. The signal handler for the abortion signal then transfers control out of the abortable part into the triggered statements.

With the *two-threaded model*, ATC can be implemented using *pthread_cancel()*. If the abortable part completes first, the secondary thread awakens the primary thread, finalizes the abortable part, and terminates. When the primary thread wakes up as a result of the signal, it checks the status of the entry call, finds it canceled, and does not execute the triggered statements. This completes the execution of the asynchronous **select** statement. Conversely, if the triggering statement completes first, the primary thread cancels the secondary thread. Finalization is performed via cleanup handlers of the secondary thread. Then, the primary thread executes the triggered statements.

For both models, an ATC via *longjmp()* or *pthread_cancel()* must be deferred if the execution of the abortable part is in an unsafe state. Otherwise the behavior of any subsequent execution is undefined. In addition, Ada 9X defines certain states as abort-deferred, and requires that abortion must not take effect while a task is in one of these states. In the *two-threaded model*, abortion can be deferred by changing the Pthreads interrupt type to PTHREAD_INTR_CONTROLLED. This ensures that thread cancellation is only acted upon at interruption points. In the *one-threaded model*, the deferral of abortion could be achieved with Pthreads by masking the abortion signal or by setting a software flag in the Ada runtime system. Since signal masking is typically a costly kernel operation, it seems preferable to defer abortion by setting a software flag. This method is explained at length in 4.4.

A comparative summary of the two models is given in Table 2. The two-threaded model allows a fairly straight-forward mapping of ATC to Pthreads primitives, but it has some serious disadvantages. First, it requires dynamic creation/termination of a thread, with possible allocation/deallocation of resources such as stack. In contrast, the overhead of the one-threaded model is limited to invoking a signal handler and performing a *longjmp()* at the time the ATC occurs. Likewise, with a kernel implementation of threads, the operations to defer and undefer cancellation are likely to be more costly than the user-level flag-setting method that can be used with the single thread model. Furthermore, for the two-threaded model, the code for the abortable part may have to be generated as a procedure which comprises the body of the newly created thread. This implies a call-back from the run-time system. Call-backs are undesirable since they may limit optimizations. The one-threaded model allows inline code generation without call-backs. Finally, the two-thread model violates an implicit invariant of most Ada runtime system designs—i.e., that a task has only one thread of control. Creating the illusion that two threads are the same task will requires that both threads share some entries of a single task control block, while other entries will have to be duplicated for each thread. This complicates the maintenance of the integrity of the task control block and is expected to increase the complexity of some RTS algorithms considerably.

Due to these disadvantages of the two-threaded model, it seems most desirable to choose the one-threaded model with a software flag in the presence of per-thread signals.

4.4 Abortion Deferral via Software Flag

This section gives a brief sketch of how abortion can be implemented with Pthreads using the one-threaded model and a software flag to defer abortion. Suppose each task control block contains a Boolean variable `Defer_Abort`. To enter a region of deferred abortion, a task simply sets this flag to `True`.

```
procedure Nonabort_Begin is
begin
  Self.Defer_Abort := True;
end Nonabort_Begin;
```

To exit such a region, the flag is simply cleared and if abortion had been requested, an ATC is performed.

```
procedure Nonabort_End is
begin
  Self.Defer_Abort := False;
  if Self.Abort_To_Level < Self.ATC_Level then
    raise Abortion;
  end if;
end Nonabort_End;
```

Integer variables `ATC_Level` and `Abort_To_Level` in the task control block reflect the current nesting level of ATC and the lowest level of any pending abortion request, respectively, the latter being protected by a mutex. If no abortion is requested, `Abort_To_Level` is set to some implementation-defined value above any possible asynchronous **select** nesting level. The actual transfer of control is indicated by raising the pseudo-exception `Abortion` (visible only to the run-time system). This may be actually implemented by a *longjmp()* or some other means of unwinding the stack, but finalization actions associated with the abortable part will have to be executed as part of the transfer of control.

To initiate an ATC, the code for completion of a triggering statement sets the abortion level of the target thread (the one which started the asynchronous **select**), and then sends the target task a dedicated signal, `Abort_Signal`. The flag `Aborting` indicates whether abortion is already pending for the target task.

```
procedure Abort_To_Level (Target : TCB; L : Integer) is
begin
  pthread_mutex_lock(Target.Mutex);
  if Target.Abort_To_Level > L then
    Target.Abort_To_Level := L;
    if not Target.Aborting then
      Target.Aborting := True;
      pthread_mutex_unlock(Target.Mutex);
      pthread_kill(Target.Pthread_ID, Abort_Signal);
      return;
    end if;
  end if;
  pthread_mutex_unlock(Target.Mutex);
end Abort_To_Level;
```

The target thread is then interrupted by the signal and executes a signal handler which will perform the ATC unless abortion is deferred.

```
procedure Abort_handler is
begin
  if not Self.Defer_Abort then
    raise Abortion;
  end if;
end Abort_Handler;
```

When suspended thread receives the `Abort_Signal`, the `Abort_Handler` cannot transfer control since an unsafe POSIX suspension function has been interrupted. Rather, the `Defer_Abort` flag is set to `True` so that the `Abort_Handler` simply returns. The thread is awakened as a side-effect of handling the signal and returns from the suspension function. At this point, the abortion level must be checked so that the transfer of control can

model	signal handler	dedicated thread
mapping to Pthreads	easy: signal handler model	duplicates Pthreads signal model
elaboration overhead	small: install signal handler	considerable: create/terminate server
signaling overhead	moderate: signal handler	small: signal delivery to server
rendezvous overhead	moderate: special case for intr. entry	none
context switch overhead	same in both cases	
interrupt safety overhead	small but distributed: software flag	none: signal masking for non-servers

Table 3: Comparison of Implementation Options for Interrupt Entries and Handlers

be initiated. (This is similar to `Nonabort_End` but the `Defer_Abort` flag is not cleared).

A nice property of this solution is that abortion of an entire task is a special case of `Abort_To_Level` with a level of zero, which is the nesting of asynchronous **select** statements of the task at creation time.

As for protected objects, the implementation of asynchronous abortion does not require the full power of the Pthreads primitives. The entire thread cancellation mechanism, including cleanup handlers, is not needed. Likewise, the ability to modify the signal mask of a thread is not used. The overhead with the Pthreads implementation of this unused functionality is a penalty paid by the Ada implementation for using Pthreads as a foundation.

4.5 Interrupt Entries and Procedures

A task entry associated with an interrupt (signal) is called an *interrupt entry*. In addition, a protected procedure associated with an interrupt is called an *interrupt procedure*. Since interrupts are asynchronous events, the implementation options for interrupts are similar to ATCs.

- *Signal Handler:* A thread executing an interrupt entry simply unmasks the corresponding signal and suspends. On receipt of the signal, a signal handler is invoked which performs the rendezvous or executes the protected procedure.
- *Dedicated Thread:* Since only one signal handler can be active at a time, a dedicated thread can be created upon installing the signal handler. This thread handles a certain signal exclusively and executes the corresponding protected procedure or performs a rendezvous.

Using the approach of a *dedicated thread*, the thread is the only thread with the signal unmasked. It suspends waiting for the signal. On receipt of the signal, it masks the signal and either performs the acceptor-part of a rendezvous for interrupt entries or it executes a protected procedure. In the latter case, the eggshell has to be entered which may cause suspension.

It might be possible to dedicate one thread to handle the set of signals of all active interrupt entries and handlers. But the interrupt response time may degrade under such a scheme since the thread may be suspended when attempting to enter the eggshell with all signals masked. Thus, a single thread for each signal seems preferable.

The *signal handler* approach appears to be complicated in the presence of protected interrupt procedures. Assume that all tasks with visibility of a protected object with an interrupt handler have the corresponding signal unmasked and may execute the handler. When a signal is received, a signal handler starts to execute on one of these tasks. To execute the protected procedure, the eggshell has to be entered. But this requires locking a mutex which is an unsafe operation. Similar problems have been observed in the implementation of interrupt entries using this approach [5]. Deferral of the signal seems the only solution which can be accomplished by using a software flag or signal masking (similar to abortion deferral).

A comparative summary of the two implementation options is given in Table 3. Considering the overhead of another software flag for signal deferral, the dedicated thread seems the preferable solution. Also, rendezvous can be implemented in the run-time system without having to consider the special case of an interrupt entry. The context switch overhead may be about the same for both approaches. On receipt of a signal, a context switch to the designated thread is performed. But to execute a signal handler, a frame has to be pushed onto the stack of the current thread which typically inflicts overhead of at least one context switch.

However, there are some drawbacks of the approach of a designated thread. Certain signals can only be received by the thread who requested them, *e.g.* timer signals and asynchronous I/O notification. Timer signals can be simulated by using a dedicated timer server for all time requests. Such a timer server can be implemented as a protected object with an interrupt procedure for the timer signal using the approach of the dedicated thread. Similar servers may be used to implement asynchronous I/O. Notice that this approach does not take advantage of the Pthreads signal model. The multiplexing of timer requests on a single thread is likely to duplicate the efforts of Pthreads, which will usually have to multiplex thread timer requests on a single underlying hardware timer.

A delay as a triggering statement of a asynchronous **select** can be implemented as a request to the timer server. On timeout, the timer server can simply perform the ATC to abort to the appropriate level. An interrupt entry as the triggering statement of a asynchronous **select** causes the dedicated thread to perform the ATC similarly.

5 Conclusion

It appears possible to implement the proposed new tasking features of Ada 9X using the POSIX threads primitives. However, this is likely to impose a noticeable overhead on the Ada implementation, as compared to using a runtime kernel specifically designed to support Ada. This is a case of *abstraction inversion*. The Pthreads services are at a level of abstraction comparable to the Ada tasking services, but to do not match closely enough to permit direct mapping of Ada operations to Pthreads operations. As a result, the Ada implementation is forced to use the Pthreads operations in a wasteful way, building mechanisms of its own that turn out to be very similar to mechanisms that must already exist within the underlying implementation of Pthreads.

The abstraction inversion problem with Pthreads is more noticeable for Ada 9X than for Ada 83, because of the new lower-level tasking constructs. This can be seen by comparing protected actions with rendezvous. As we have shown, the queuing operations of mutexes and condition variables cannot be used directly to implement protected entry queues. Likewise, the priority ceiling option of Pthreads mutexes is not powerful enough to save the Ada runtime system from keeping track of active priority. This semantic mismatch is also true of the implementation of rendezvous. As with protected entry calls, there is no good way to use the queuing provided by Pthreads in rendezvous, so entry callers wait on their own condition variables and are signaled individually by the accepting task. Priority changes for rendezvous are also expensive. The wasted overhead may account in part for the approximately 25% performance penalty observed in one implementation of rendezvous over Pthreads when compared to an implementation over threads customized for Ada [5]. However, rendezvous is in general a more complex synchronization mechanism than anything provided by Pthreads, so the contribution of Pthreads overhead to the overall cost of rendezvous is comparatively small. In contrast, since protected objects are inherently much simpler, and provide functionality very similar to that of mutexes and condition variables, the comparative overhead is likely to be more noticeable.

On the positive side, Pthreads promises a readily available multithreaded platform that can be used to simplify the job of implementing Ada tasking and increase the portability of implementations. In particular, an Ada RTS built over such a threads-layer should be portable between single and multiprocessor configurations. Further, I/O operations performed by one task should not block the execution of other tasks. Implementing Ada tasks as POSIX threads may also be desirable in situations where the Ada application needs to use libraries written in the C language that happen to make use of POSIX threads. For many applications, these benefits will outweigh the additional runtime overhead.

References

[1] Ada 9X Project Office. *Ada 9X Integrated Language Specification: Draft*, December 1992. Available by anonymous FTP from ajpo.sei.cmu.edu.

[2] Ada 9X Project Office. *Ada 9X Snapshot of Annexes Prior to Revision: Version 4.9*, October 1992. Available by anonymous FTP from ajpo.sei.cmu.edu.

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

[4] 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.

[5] E.W. Giering and T.P. Baker. Using POSIX threads to implement Ada tasking: Description of work in progress. In *TRI-Ada '92 Proceedings*. ACM, 1992. To appear.

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

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

[8] Frank Mueller. A library implementation of POSIX threads under UNIX. In *Proceedings of the USENIX Conference*, pages 29–41, January 1993.

[9] Ada 9X Mapping/Revision Team. Ada 9X implementation module 1: Protected records, operator visibility, child packages. Technical report, Intermetrics, Inc., January 1991. Available by anonymous FTP from ajpo.sei.cmu.edu.

[10] Ada 9X Mapping/Revision Team. Ada 9X implementation modules II–IV. Technical report, Intermetrics, Inc., June 1991. Available by anonymous FTP from ajpo.sei.cmu.edu.

A Transformation of a protected object using Pthreads

One possible transformation of the protected object shown in Section 3.3 using Pthreads and the the queuing operations described in Section 3.6 is

```
package P is
  procedure Sig;
  procedure Clear;
  procedure Wait;
end P;

with Task_IDs; use Task_IDs;
with Task_Queues; use Task_Queues;
package body P is
  Signals : Integer := 0;
  Lock : Mutex;
  Entry_Queues : array(1..1) of Queue;
  Wait_Count, : Natural := 0;

procedure Service_Entries;

procedure Sig is
begin
  pthread_mutex_lock(Lock);
  Signals := Signals + 1;
  Service_Entries;
exception
when others =>
  Service_Entries;
  raise;
end Proc;

procedure Clear is
begin
  pthread_mutex_lock(Lock);
  Signals := 0;
  Service_Entries;
exception
when others =>
  Service_Entries;
  raise;
end Proc;

procedure Wait is
  E : Exception_Type;
begin
  pthread_mutex_lock(Lock);
  if Signals > 0 then
    Signals := Signals - 1;
  else
    Enqueue(Entry_Queues(1), Self);
    while Enqueued(Self) loop
      pthread_cond_wait(Self.Cond, Lock);
    end loop;
    E := Self.Exception_to_Raise;
    Self.Exception_to_Raise := No_Exception;
    Raise_Exception(E);
  end if;
  Service_Entries;
exception
when others =>
  Service_Entries;
  raise;
end Wait;

procedure Service_Entries is
  Temp_Barrier : Boolean;
  State_Changed : Boolean := True;
  T : Task_ID;
begin
  while State_Changed loop
    State_Changed := False;
    for Entry_Index in 1..1 loop
      T:= First(Entry_Queues(Entry_Index));
      if T /= Null_Task then
        case Entry_Index is
        when 1 => Temp_Barrier := Signals > 0;
        end case;
        if Temp_Barrier then
          State_Changed := True;
          case Entry_Index is
          when 1 =>
            begin
              Signals := Signals - 1;
            exception
            when others =>
              T.Exception_to_Raise := Current_Exception;
            end;
          end case;
          Dequeue(Entry_Queues(Entry_Index));
          pthread_cond_signal(T.Cond);
        end if;
      end if;
    end loop;
  end loop;
  pthread_mutex_unlock(Lock);
end Service_Entries;
```