POSIX/Ada Real-Time Bindings: Description of Work in Progress

E.W. Giering III T.P. Baker[1]

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

Abstract

Ada bindings are being developed for the proposed POSIX Realtime Extension and Threads Extension. POSIX is a standard application program interface to operating system services. The Threads Extension is of particular interest for Ada implementors and users, since it provides services that overlap significantly with Ada tasks. Work in progress on the implementation of Ada tasks over POSIX threads is beginning to shed light on the problem of producing an Ada binding for the POSIX Threads Extension.

1 Introduction

The POSIX/Ada Real-Time (PART) project[2] is seeking to accelerate development of standard Ada language bindings for the POSIX Realtime Extension [8] and Threads Extension [9] (Pthreads). An important objective of this project is to identify and resolve interactions between the Ada task model and Pthreads that might affect the Ada language implementation or the application program. In order to expose such interactions, and to provide an empirical basis for development of an Ada language binding, a prototype Ada runtime system is being developed using the Pthreads primitives.

POSIX is a family of standards for 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 national (ANSI) and international (ISO/IEC) 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 operating system[3].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 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 first POSIX standard, and has been approved by the ISO as ISO/IEC 9945-1:1990.

The two POSIX standards with which PART is concerned are 1003.4 (Realtime Extension) and 1003.4a (Pthreads). Both of these exist only as drafts, which are currently in the IEEE ballot process. The 1003.4 interface 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. Pthreads 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 synchronization. Pthreads is of special interest because it provides access to services that overlap significantly with the Ada language support for tasks.

Pthreads offers several potential benefits for Ada implementors and users, if Ada tasks are implemented as threads. A task can make an operating system (OS) service request without danger of blocking the entire program. Since the Pthreads interface is designed for a multiprocessor architecture, Ada applications will be able to take advantage of multiprocessor architectures to truly execute tasks in parallel, without special support from the Ada compiler vendor. The result can be

[1]Visiting at the Software Engineering Institute, Carnegie Mellon University.

[2]PART is a project at the Florida State University which is funded by the Ada Joint Program Office under the Ada Technology Insertion Program, through the U.S. Army CECOM and Telos Corporation.

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

improved real-time response and throughput. To the extent that an Ada runtime system (RTS) uses standard POSIX interfaces, it should be portable across POSIX-compliant operating systems. This should reduce the cost of developing and maintaining the Ada implementation, and is likely to result in less variation in task behavior across Ada implementations. Pthreads also provides some extended features, such as dynamic priorities, which have not yet been standardized by the Ada language but which are in demand by realtime users.

More details are provided in the rest of the paper. Section 2 provides some background on the relationship of Ada tasks, POSIX processes, and POSIX threads. Section 3 describes progress on producing Ada bindings for the draft POSIX realtime standards. Section 4 describes our implementation of Pthreads, and Section 5 describes our implementation of Ada tasks over Pthreads. Section 6 outlines bindings to Pthreads operations beyond those provided by simply mapping Ada tasks to POSIX threads. Section 7 draws conclusions and outlines future work.

2 Background

2.1 Ada Tasks *vs.* POSIX Processes

The Ada language supports concurrent programming via tasks. Ada tasks execute within a single address space. Since tasks are specified by the language, Ada implementations have to provide support for them in their runtime systems. The Ada task model is not commonly implemented by commercial operating systems or real-time executives, so the RTS support for Ada tasks is usually written by the Ada language implementor.

In contrast, the C language does not support concurrent programming directly. Concurrent programming applications in C invoke the services of an underlying operating system via explicit subprogram calls. POSIX has increased portability of concurrent programming in C by standardizing an application interface to the operating system.

In POSIX, concurrent processing is supported via *processes*. The POSIX process is a heavier weight entity than the Ada task. Each process runs in a separate address space. There is a great deal of data that must be stored by the operating system for each process, and the operations involved in creating, deleting, and communicating between processes have considerable overhead. In cases where processes share a single processor, switching processes also tends to have a high overhead.

An Ada application running under POSIX can benefit from using a combination of processes and tasks. Processes support multiprogramming, and tasks provide a form of lightweight concurrency within a process. The strengths of processes are that they support programming in the large: large systems can be constructed out of independently developed and compiled programs, written in different languages, and can be reconfigured dynamically. The strengths of tasks are that they can be implemented with much less overhead, and inter-task interfaces are checked statically by the compiler and linker.

2.2 POSIX Threads

C language applications also have a need for lightweight threads of control within a process. This is the motivation for Pthreads. Threads play the same role for a C application as do tasks for an Ada application.

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()*; Pthreads then calls this function in a separate thread of control.

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 at each unlock. Mutexes are similar to binary semaphores; the principal difference is that the thread which owns 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 – see 3.1). A mutex is associated with the condition variable by the *pthread_cond_wait()* call; the 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:

```
pthread_mutex_lock(mutex);
while(!condition)
  pthread_cond_wait(condition_variable, mutex);

/* Mutex is locked and
   condition is true at this point */
```

Pthreads 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.

Pthreads may be implemented by a user-level scheduler within each process. This is called a *library implementation*. This is not the only way POSIX threads may be implemented, though. Because Pthreads is a standard for operating system interface, the operating system vendor may choose to support threads directly in the OS. This is called a *kernel implementation*.

A kernel implementation of threads has several potential advantages over a library implementation. One of these is that a single thread can block without causing the whole process to block[4]. Another possible advantage is improved response time for high-priority threads, if the OS kernel implements global priority scheduling for threads, and especially if the OS kernel itself can be preempted. A third potential advantage is increased throughput, if there is more than one processor and the OS allows several threads of a process to run on different processors at once. On the other hand, kernel implementations generally pay for these advantages in higher runtime overhead.

[4] This may also be possible in a library implementation on systems which support asynchronous equivalents to the suspending calls.

3 Ada Binding to POSIX

3.1 Ada Bindings to POSIX Standards

POSIX is intended to provide source code portability of operating system services for application programs (as opposed to system software). Only the source code constructs which access system services are defined; how they are represented in the compiled, binary version of the application is not. The intent is that the source code to a POSIX compliant application will execute correctly when compiled on any POSIX compliant host. This means that for each language that is to access POSIX, an interface in that language, called a *binding* to that language, must be provided.

The POSIX 1003.5 working group is responsible for developing Ada bindings to POSIX standards. This group has developed a binding for 1003.1, designated 1003.5. It has recently started work on a binding for 1003.4 and 1003.4a, designated 1003.20.

Ultimately, POSIX is to be defined in a language-independent manner. Language bindings will be based on the language-independent standard. The IEEE is currently working on converting the 1003.1 and 1003.4 standards to this format. However, at this time, the proposed 1003.4 and 1003.4a interfaces are only defined for the C programming language. Work on Ada bindings must be based on these C interface specifications. This involves making judgements as to what is language-independent functionality and what is specific to the C language.

Producing an Ada binding from a C interface is awkward. Due to the differences between C and Ada, the types and operations of the Ada bindings cannot correspond exactly one-for-one with the C specifications. An example of this problem is the interface for handling signals.

Signals are a form of software interrupt which can be delivered to a process by the operating system. Signals are handled in C by passing a procedure pointer to the POSIX implementation. When a signal is delivered to the process, this procedure is called. Ada lacks procedure pointers, but provides *interrupt entries*, task entries which can be bound to an interrupt such that the occurrence of the interrupt causes the entry to be called. The task owning the interrupt entry can handle the interrupt by waiting for the entry call using an **accept** statement. There is no problem with the underlying semantics of signals; the two languages simply do not have a common method that can be used to represent signal handling; Ada lacks procedure pointers and C lacks multiple threads of control and entries. The current 1003.5 and 1003.20 drafts therefore use interrupt entries to handle POSIX signals.

Another special problem for the Ada bindings is that the Ada RTS is an OS client; it depends on the operating system for essential services. With C it is the other way around. Historically, UNIX systems have been written in C. Therefore, the C language implementation does not depend on the operating system, though the operating system is likely to depend on the C language implementation. This difference means that some services which the POSIX C-language interface provide directly to the application are needed by the Ada RTS.

The kind of interface to operating system services required by the Ada RTS is quite different from what is appropriate for an Ada application. The interface used by the Ada RTS must be at a lower level than the RTS itself. It must not rely on language features that invoke RTS services, which in turn rely upon the OS interface (a circularity). For example, since the Ada RTS implements tasks, its interface to the OS cannot be in terms of Ada tasks. The interface should also avoid introducing any unnecessary overhead. Given there is a C-language interface to the underlying operating system, and the RTS is written in Ada, it can use `interface` pragmas to access the C-language functions that provide OS services. If this interface is awkward to use and depends on features of the Ada implementation, it does not matter, since it is only used internally by the Ada implementation. In contrast, an interface intended for Ada applications should be convenient and portable. This means avoiding dependences on the Ada language implementation, enforcing type checking, and generally protecting the user from potentially disastrous errors.

The 1003.5 draft only addresses the needs of Ada applications. It does not address the OS interface needs of the Ada RTS. It does recognize these needs, however. Certain OS services that are directly available to the application in the C-language interface are reserved for use by the Ada RTS. For example, the signals SIGFPE, SIGILL, SIGSEGV, and SIGALRM, and several of the operations on signals are not made directly available. This is because these features are likely to be used by the Ada RTS to implement exceptions and time delays. If an application uses them directly it will have implementation-dependent results, and at worst may crash the Ada RTS.

In 1003.5, some of the OS services that may interact with the RTS are made available to Ada applications, with warnings that the effect is implementation dependent. For example, the *fork()* operation creates a duplicate of a process. Some Ada implementations create multiple server-processes to execute Ada tasks, and some simply interleave the execution of tasks within a single process. Therefore, the effect of *fork()* on an Ada program with multiple tasks cannot be specified completely without ruling out reasonable implementations. It is possible to define a safe operation for starting a new process, combining the effects of *fork()* and *exec()*, but this does not quite cover all the applications of *fork()*. Therefore, 1003.5 provides two interface packages. The "safe" one contains `Start_Process`, and the unsafe one contains `Fork` and `Exec`.

3.2 Ada Binding to 1003.4

Under the auspices of the PART project, a complete Ada binding specification has been written for Draft 12 of 1003.4. Producing this Ada binding was not especially difficult, since precedents for treatment of most of the fundamental issues were already set by 1003.5. Draft 0.4 was reviewed by the 1003.5 working group in April, and Draft 0.5[4] is currently being circulated to the 1003.4 and 1003.5 working groups for further review. Steps are being taken to transfer copyright to the IEEE, at which point it will become Draft 1 of 1003.20.

3.3 Ada Binding to Pthreads

Defining a suitable Ada binding for Pthreads is more difficult than for 1003.1 and 1003.4, since the overlap between the services provided by Pthreads and the Ada language is much greater. It is likely that the Ada RTS will use Pthreads services to implement Ada tasks, or will implement Ada tasks in a way that interferes with the implementation of Pthreads. Similar problems came up in 1003.5, with signals and process creation. The solutions taken there were to reserve certain operations for use by the RTS, and to provide some operations to the application with warnings that they are unsafe. This same approach can be used with Pthreads, but will need to be applied to virtually the whole of Pthreads, rather than just a few features.

When Ada is implemented on a UNIX platform without threads, the Ada RTS switches control between tasks within the process in a way that is invisible to the host operating system, using the native machine instructions of the host processor[5]. If Ada tasks are implemented

[5] There are exceptions to this pattern. Some Ada implementations do map tasks onto separate processes, but to do this effi-

in this way on a system that supports Pthreads, there will be two potentially interfering implementations of threads of control within a process. The Ada application could not invoke any Pthread services safely. In particular, it would not be possible for the Ada application to make use of multi-threaded library services that might be implemented in C, such as window and database interfaces. Use of such a library by an Ada program would imply coexistence of the two competing mechanisms for switching execution between threads of control. It is difficult to see how disaster can be avoided unless these mechanisms are layered or the implementation of one has considerable knowledge of the other[6].

The rationale of Draft 6 of Pthreads indicates that it is intended to support the implementation of Ada tasks [9]. The principal problem with a Pthreads-based Ada RTS is overhead. The Pthreads interface is not specifically designed for Ada, nor can a Pthreads-based Ada RTS take advantage of specific machine features as an Ada-only RTS can. This can lead to loss of efficiency. If Pthreads are supported in the OS kernel, kernel entry and exit overhead can add to this problem.

There are reasons an Ada implementor may not wish to map certain tasks one-to-one onto threads. (See Section 6.2.) One case is that of "passive" tasks, which should not be mapped onto threads at all. Another case is where the architecture has multiple processors, and the compiler is able to detect potential parallelism within a single task; in this case, it might be desirable to map portions of a single task onto multiple threads. For this reason, an Ada binding for Pthreads should probably not provide direct visibility of the threads that might be underlying Ada tasks.

Specifying that Ada tasks are implemented as POSIX threads goes a long way toward providing an Ada binding for Pthreads, but it is not the whole answer. There are Ada task features (e.g. rendezvous) not supported by Pthreads, and Pthreads features (e.g. user-controlled priorities) not supported by Ada. A complete binding will have to address all the services provided by Pthreads, either by pointing out how they are provided by tasks or by defining an interface to them.

It is far from clear which Pthreads services can or should be used directly by an Ada application. Some calls will require large amounts of "glue" code to protect Ada tasks and Pthreads from each other. This glue will be dependent on the implementation of both Pthreads and Ada. The selection of a portable subset of Pthreads functionality must be based on some idea of "reasonable" implementations of Pthreads and Ada, as well as opinions regarding programming practices and the amount of interface overhead which is tolerable.

Experience implementing Ada runtime systems using POSIX threads should shed light on these issues, and also provide information about the potential overhead of using POSIX threads on task performance. One such implementation is described in the following sections.

4 Pthreads Implementation

4.1 Objectives and Scope

In order to provide a basis for implementation of an Ada RTS layered over Pthreads, and to better understand the Pthread semantics, the PART project is producing an implementation of Pthreads. The primary considerations in this effort are faithfulness to the Pthreads specification, and timely completion. Efficiency is also a consideration. The scope is presently limited to services needed by an Ada RTS.

4.2 Implementation Approach

We had originally planned to obtain an operating system which supports POSIX 1003.4 and Pthreads, or which provides a close enough approximation that a POSIX compliant C-language interface could be developed using just glue-code. We were unable to obtain such an operating system for our computing equipment, which consists of Sun SPARCstations[6].

We chose to implement Pthreads as a C library. A working Pthreads implementation was needed very soon, so as not to hold up the Ada RTS development work. We already had a partial library implementation of Pthreads that was developed as a Master's project [11], and felt it would take less time and effort to write and debug a library implementation, at least those portions relevant to the implementation of tasks[7]. We were also interested in verifying whether it is truly possible to implement all the Pthreads services without special kernel

ciently for the general case requires process memory sharing features not available on all UNIX systems.

[6] Sun is a trademark of Sun Microsystems, Inc. SPARC and SPARCstation are trademarks of SPARC International, Inc.

[7] In retrospect, it seems producing a complete library implementation of Pthreads (if this is indeed possible) may not be easier than producing a kernel implementation. One does not have the problems inherent in modifying the OS kernel, but one then has to provide a thick wrapper around all potentially blocking POSIX functions.

support. Another advantage of a library implementation is that it has potentially less overhead than a kernel implementation. A kernel call can be quite expensive, involving a context switch and operation dispatch. Finally, a library implementation allows more direct comparison between our Pthreads-based RTS and our baseline, the non-Pthreads commercial RTS.

The vendor RTS which serves as our baseline provides its own implementation of multiple threads within a UNIX process, similar to Pthreads. However, the baseline RTS, including its support for multiple threads within a process, is written expressly to support Ada. Since one of the goals of this research is to determine to what extent Pthreads are compatible with the implementation of Ada tasks, we wanted to make as direct a comparison between a Pthreads-based RTS and Ada-only RTS as possible. Since the baseline RTS is implemented without OS kernel support for tasks, a more meaningful comparison can be made between it and one based on a library implementation of Pthreads. This will hopefully provide some insight into the inefficiencies introduced by the Pthreads interface.

There are necessarily critical sections within the Pthreads implementation, which must run to completion without interference from concurrent threads of control. One way to protect critical sections is via locking operations on specific data objects; this fine-grained locking allows several threads of control to execute Pthreads service calls concurrently, and allows calls to be preempted. Another way to protect critical sections is via the extreme of coarse-grained locking; Pthreads services can be implemented as a *monolithic monitor*, in which Pthread service calls cannot execute concurrently or be preempted. Fine-grained locking has a clear advantage on a multiprocessor system. It allows parallel execution of Pthreads service calls. On a single processor system, the advantage is less clear cut. Fine-grained locking does reduce priority inversion (the execution of a low priority thread at the expense of a high priority thread), which occurs when a high priority thread tries to enter a critical section held by a low priority thread. This is important because the ability to schedule a system of tasks to meet timing constraints is compromised by priority inversion [12, 10]. The monolithic monitor is likely to cause greater priority inversion by enforcing mutual exclusion where it is not actually needed. However, fine-grained locking requires many more locking and unlocking operations; this execution time overhead can also compromise the ability to meet timing constraints, since less processor time is available for the application. Our Pthreads implementation is a monolithic monitor, because it is intended for a single processor system, and we expect the execution times of Pthread service calls to be generally short. In this context, we believe that fine-grained locking would result in poorer performance.

4.3 Implementation Status

At the time of this writing, the Pthreads implementation is complete enough that it is able to support the needs of the Ada RTS for running validation tests.

During the implementation work we noticed that the Pthreads specification leaves quite a few semantic details unspecified — so that behavior will be arguably less portable than with Ada tasks. We also observed that Pthreads are heavier-weight than expected. In particular, supporting per-thread signal delivery imposes considerable overhead.

5 Ada RTS Implementation

5.1 Objectives and Scope

In order to learn more about the relationship between Pthreads and Ada tasks, the PART project is developing a prototype implementation of an Ada RTS based on Pthreads. From this implementation, we hope to:

- Determine whether Ada tasks can be implemented on top of Pthreads.
- Discover how close one can get to a portable Ada RTS, whose only interface to the operating system is through the C-language interfaces defined by the POSIX standards.
- Expose ways in which Pthreads support for the Ada RTS might be improved.
- Obtain measurements which can be used to estimate the overhead of using Pthreads, by comparing the performance of a Pthreads-based Ada RTS *versus* an Ada-only RTS for the same compiler, hardware, and operating system.
- Expose potentially dangerous interactions between uses of raw Pthreads operations and their use by the Ada RTS, from which the Ada binding should attempt to protect the application.
- Publish guidance for implementors who implement Ada over Pthreads, and potential Ada users.

After this project started, we learned that several compiler vendors are also producing Ada implementations for various implementations of threads, some of which purport to be Pthreads-compliant. The primary differences between the PART project and these efforts seems to be that the PART project results will be published, including implementation techniques and performance results (even if unfavorable to our work), and that the PART project avoids using OS features not specified by POSIX, even if this means poorer performance or reduced functionality.

The primary consideration of the Ada RTS prototype is faithfulness to the Ada language specification. We plan to implement all task semantics, in a fashion consistent with the Ada language definition and the Ada compiler validation tests. Efficiency is a secondary consideration, but also important. Unless both the Ada RTS and Pthreads implementation are designed with attention to efficiency, performance measurements will not mean much.

The scope of the RTS implementation includes all runtime services required by the Ada language which involve operating system services or task support. Subject to limitations of schedule and budget, it should provide validatable runtime support for the full Ada language.

5.2 Implementation Approach

To reduce the time required to produce a prototype, the Pthreads-based RTS is designed as a modification of a commercial compiler's RTS. The Ada compiler is one produced by the Verdix Corporation for the Sun SPARCstation computer running SunOS (Sun's version of the UNIX operating system)[8]. Verdix loaned FSU the source code to their RTS for this purpose. The chief modification consists of removing all the code that implements tasks and rewriting it based on Pthreads. Other portions of the RTS, such as support for I/O and `Calendar`, are modified to comply with the POSIX 1003.1 and 1003.4 interfaces.

Our basic approach is to implement each Ada task as a POSIX thread. The Ada RTS will exist as a set of library subprograms which use the Pthreads operations to manipulate the state of these threads in such a way that Pthreads will schedule them according to Ada semantics. The scheduling of the threads is entirely under the control of Pthreads, as is the state of the thread. Runtime stack allocation is also under the control of Pthreads.

[8] SunOS is a trademark of Sun Microsystems, Inc.

In many cases the mapping between an Ada task and the corresponding thread is straightforward. For example, the scheduling policies that Pthreads is required to support are compatible with Ada priorities. 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.

An Ada RTS implemented over Pthreads should be multithreaded. There are several places in the RTS, such as during rendezvous, where a context switch in the RTS is required, so the Ada RTS cannot always run to completion without interruption. In addition, Pthreads does not specify how many processors it uses to execute threads, so it is important that a portable Ada RTS should not do anything that would unnecessarily limit parallelism. The possibility of parallelism must also be taken into account in designing RTS algorithms. For example, a thread's access to shared resources cannot be protected by raising its priority over that of all other threads. With multiple processors, all threads can be running simultaneously regardless of priority.

5.3 The Ada Task Control Block

Since there are Ada task features such as rendezvous and task dependence not directly supported by threads, it is necessary to associate Ada-specific data with each thread that implements an Ada task (*task thread*). This data is stored in a record called the Ada Task Control Block (ATCB). In our RTS, a pointer to the ATCB of a task serves as the *task identifier*.

5.4 Task Creation and Activation

In Ada a task may be created before it is activated. This is in contrast to Pthreads, where *pthread_create()* creates the thread, along with all the structures supporting it, and starts it executing immediately. An Ada RTS over Pthreads can either create a thread at the time of task creation, in which case the thread must suspend itself until the task is activated; or it can wait to create the thread until task activation time (as discussed in Section 3.3.2.6 of the Pthreads draft [9]). In the latter case, task creation involves only allocating and initializing the ATCB, but it allows a task to be created when there is actually insufficient memory to create a thread. Since this should be detected at the point of task creation and raise `STORAGE_ERROR`, we decided that a thread should be created when a task is created.

The need to access the ATCB from the thread has constrained the design of task activation. Our implementation stores a pointer to the ATCB at the bottom of the thread's stack. Pthreads provides no way for a thread to obtain the address of any stack except its own, so the thread itself must initialize this pointer to the ATCB. This is done by passing the address of a wrapper procedure and the ATCB to *pthread_create()*. Pthread calls the wrapper with the ATCB address in a new thread of control; the wrapper then stores the ATCB at the bottom of its stack and calls the task body. The task body address and other machine dependent information needed to call it are passed in the ATCB; these fields are not used after thread initialization, but this cost in idle storage is balanced by elimination of separate dynamic storage management for the initialization parameters.

5.5 Rendezvous

The approach we use to implement rendezvous is outlined in a note to the Pthreads draft [2]. It uses a mutex and two condition variables for each task, as well as a queue for each entry. Under this scheme, a task waiting for a rendezvous (the first calling or accepting task to execute its rendezvous primitive) waits on one of its condition variables and the mutex of the accepting task. The use of the acceptor's mutex in both the acceptor and the caller simplifies synchronization between the two tasks, and having each caller use its own condition variable allows the acceptor to wake up the caller with which it actually makes rendezvous using *pthread_-cond_signal()*.

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. A task can be declared in a dynamic scope such as a procedure, block statement, or another task. In order to ensure that this environment persists so long as the task can execute, the concept of task *master* is introduced. A task that is a declared object is said to *depend* directly on a master that corresponds to the scope where the task object is declared. A task that is created via an allocator is said to depend on a master that corresponds to the scope where the access type is declared. 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. In other words, 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 accept an entry call, after which the task is no longer eligible for termination.

Pthreads lacks support for the Ada concept of task dependency 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 structure created at runtime. Our implementation uses an alternate algorithm, originally used in an implementation of the Model RunTime System Interface (MRTSI) [1].

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 be 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 therefor 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 waiting 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 a synchronization point as defined in the ARM [13]; these 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()*. Of course, it does not provide for Ada task termination processing, which must be done before an aborted task can be terminated. This kind of finalization operation is frequently needed by threads to protect themselves from being canceled while resources are held (e.g. a locked mutex) or data structures are in an inconsistent state. Pthreads therefore provides for *cleanup handlers* which a thread can define to be executed when it is canceled.

Our implementation places much of abort processing in cleanup handlers defined for each task thread. Unfortunately, cleanup handlers cannot by themselves implement all of Ada **abort** semantics. When thread cancellation takes effect, the thread becomes eligible to execute its cleanup handler; when it actually executes depends on when the thread is next scheduled. However, some parts of Ada abortion must take place before the **abort** statement finishes execution. 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** statement implementation performs these actions and then cancels the thread; the remainder of abortion processing is done by the thread itself in its cleanup handler.

Not all parts of the RTS code can be aborted at all times. There are places where RTS data structures are inconsistent. Also, entry calls cannot be aborted while they are in rendezvous. To prevent cancellation in such critical sections, Pthreads defines *controlled interruptibility*, which prevents cancellation from occurring except at certain points in the code, generally those points where the thread could suspend (very similar to Ada synchronization points). It also allows cancelability to be disabled, which can be used to protect the calling task during a rendezvous. Our implementation does not do this; calling tasks waiting on rendezvous completion are not canceled. Instead, they check themselves for abnormality when the rendezvous completes.

Controlled interruptibility is used throughout the Ada RTS. The logic behind this is that it is not as important for the RTS to be asynchronously abortable as it is for the application code. In particular, the RTS should not have any unbounded loops. It is only important that the RTS be interruptible at suspension points, which should be the only places where an RTS call can experience unbounded delay.

5.8 Implementation Status

At the time of this writing, our Ada RTS is able to activate, terminate, and abort tasks, and execute all forms of rendezvous except interrupt entries. We are currently correcting defects which were revealed by the executable task tests (C9) of the Ada Compiler Validation Capability (ACVC).

6 Ada Binding for Pthreads Operations

One of the goals of the PART project is to develop an Ada binding to Pthreads. Implementing tasks as POSIX threads already constitutes a basic binding; an Ada application can access much of the Pthreads functionality via the standard Ada operations on tasks. Access to other Pthreads services can be provided by Ada packages. We are beginning to consider the design of such package interfaces. Our Ada RTS implementation has exposed some issues that must be addressed, and we have tried to consider other possible implementations as well.

Since our Ada RTS is implemented in Ada, we have already developed an Ada package interface to Pthreads, for use by the RTS. This is a very direct low-level binding. It consists of type definitions corresponding to the C-language data types, and procedure definitions with interface pragmas corresponding to the C-language functions of Pthreads. This interface is adequate to implement an RTS, but it makes implementation-dependent uses of `'ADDRESS`, and is unsafe.

A standard application-level binding for Pthreads should be portable and safe, or at least it should isolate and point out any nonportable or unsafe features. On the other hand, it need not provide access to all the services needed to implement an RTS.

The following subsections discuss Pthreads operations which might be used in Ada applications, and the cost of providing them, both in safety and efficiency. Each subsection here corresponds to a section of Draft 6 of the Pthreads specification [9]. Ada tasks are assumed to be implemented as threads unless otherwise specified.

6.1 Thread Creation and Management

The *pthread_create()* operation is used to create a thread. It is not clear that direct access to this operation should be provided to the Ada application. Assuming Ada tasks are implemented as threads, the capability of creating and terminating a thread is already available. The overhead involved in just running a thread, such as scheduling and dispatching, will be the same, whether it is created as an Ada task or directly as a POSIX thread. The difference is the overhead of task creation, activation, and termination, which are considerably more complicated than thread creation and termination. An application which dynamically creates and destroys large numbers of simple threads of control might benefit from using threads instead of tasks. This seems to be the only reason to provide this operation directly to an Ada user.

Allowing an Ada application to create threads that are not Ada tasks is unsafe, because of potential dangling references and calls to Ada task operations from non-task threads. If a lexically nested Ada procedure is invoked as a thread, there is risk that the thread may persist longer than the lexical context. The thread may then read or update storage that has been deallocated and reused for another purpose. (This is the reason for Ada's complex task termination rules.) This problem can be avoided if threads are limited to library-level procedures, but such a rule cannot be enforced without (costly) compiler support. If a non-task thread calls an RTS operation which applies to the calling task, the will not recognize the thread as a task, and the RTS is likely to fail. In particular, the thread may call a procedure which (perhaps very indirectly) calls a task entry. This can be partially solved by a compromise: the implementation of the Pthread Ada binding and Ada RTS can allocate an ATCB for a non-task thread at the first point where it is required. (See 6.8 for more detail.)

Assuming that the capability of creating non-task threads is desired, and the user takes responsibility for safe usage, the the Ada binding for this operation and the operations that serve only to support it (such as the functions for manipulating thread attributes) should be placed in a package with a name such as `POSIX_Unsafe_Thread_Primitives`, analogous to the `POSIX_Unsafe_Process_Primitives` of the 1003.5 draft [7]. The safe Pthread functionality for thread creation, analogous to `POSIX_Process_Primitives`, does not require a package since it is accessed through Ada tasks.

An Ada binding for unsafe thread creation must provide a way for the user to specify the subprogram to be executed by the new thread. One possibility is via a parameter of type `SYSTEM.ADDRESS`, obtained by applying the `'ADDRESS` attribute to the subprogram name. The Ada specification states that this is the address of the first instruction in the subprogram, but implementations are not very uniform in their interpretation of this; there may be portability problems. In any case, there is no way to verify that a particular value of type `ADDRESS` is actually the address of a subprogram, much less that this is a library-level procedure with the right parameter-result profile. If this approach is used, care must be taken by the user to verify that the subprogram not only is library-level, but also that it does not require any implicit parameters. For implementation with generic code-sharing, this is likely to rule out generic subprograms and subprograms that are declared within a generic unit. An alternative is to specify the subprogram via a generic formal parameter. This approach has the advantage of permitting more consistency checking, and possibly relaxing the restrictions on the thread subprogram, but is likely to require special compiler support.

The propagation of exceptions from threads must also be addressed. Without special precautions, what will happen when an exception is propagated from a thread will depend on the Ada compiler and the Pthreads implementation. If the subprogram is the first frame on its stack, there will be no place to which to propagate the exception, but the exception propagation mechanism is unlikely to expect this situation, since it expects subprograms to have callers. (The main program procedure is a special case, for which the compiler may generate special code.) There is also the possibility that Pthreads will have called the user function from one or more levels of wrapper-subprograms (this is true of our implementation); the exception would then be propagated into those frames. At best, exception propagation would pop these frames without allowing the functions to complete any needed cleanup. At worst, the exception mechanism might expect information about exception handlers to be stored in the frames; this could easily cause the program to crash.

The problem of exception propagation can be handled in the implementation of the Pthreads Ada binding by passing *pthread_create()* the address of a wrapper function and the its argument, the address of a structure containing the address of the user-provided thread subprogram and its argument. The wrapper would call the user subprogram, providing a handler with a null body for all exceptions. This handler would cause the wrapper to return normally when an exception was propagated out of the user-provided subprogram.

The *pthread_join()* operation allows a thread to wait on the termination of another thread. When used between threads representing Ada tasks it can be dangerous, since Ada tasks also wait for each others' termination. For example, a task joining with its master would deadlock, since the master cannot terminate until all of its dependents do. This is therefore an unsafe operation if the IDs of task threads are available to the application.

The *pthread_exit()* operation terminates the calling thread. This bypasses Ada termination, and is therefore an unsafe operation when applied to task threads. It causes cleanup handlers to be called, so it can be made safe by including all needed termination processing in a cleanup handler defined for all task threads, as described in Section 6.7, below.

The *pthread_detach()* operation indicates to Pthreads that the memory of the target thread can be reclaimed upon termination. The *pthread_join()* operation automatically detaches the target thread, and its effects on an already detached thread are undefined. Since the RTS may use *pthread_join()*, it is unsafe for an application to detach a task thread. If non-task thread creation is allowed, however, this is needed to reclaim thread storage.

The *pthread_self()* function returns the calling thread's ID. Many of the Pthreads operations require the ID of the target thread; if bindings to these operations are to be allowed, some way of obtaining the thread ID of an Ada task is needed. The *pthread_self()* function can be used for this purpose, but it only allows access to the thread ID of the calling task. Ideally, the compiler would provide an attribute, say `'THREAD`, which, when applied to the name of any task, would return its thread ID. In the absence of compiler support, a generic function instantiated with a task type, taking a task argument and returning its thread ID, could be provided. This would be hard to use, however, since a separate instantiation would have to be defined for each task type, and the task type would have to be visible.

Note that the availability of the thread ID of a task can make otherwise safe functions unsafe. For example, see the discussions under *pthread_join()* and *pthread_detach()*, above. This is especially true if the thread ID is passed outside the normal Ada scope of the task.

The *pthread_equal()* operation should be used for all comparisons between thread IDs. Note that the use of the predefined Ada "=" operator is unsafe for this purpose, and overloading "=" for this type is not practical, due to Ada language restrictions on user definitions of "=".

The *pthread_once()* operation takes a pointer to a function as argument, so that it has many of the safety problems described above for *pthread_create()*. Its effect is to call this procedure once, if and only if it has not been called via *pthread_once()* earlier in the execution of the same program. This can be used to ensure initialization functions associated with use of a C library are executed exactly once, even though where there may be several calls to them, possibly from different threads. Since such initialization is already provided in Ada by the elaboration mechanism, this function is unnecessary for an Ada application.

6.2 Synchronization Primitives

If the application is permitted to create threads directly, operations on mutexes and condition variables are needed to synchronize them. Since these primitives are likely to have lower execution time overhead than Ada rendezvous, they may also be useful for synchronization between Ada tasks.

The use of mutexes and condition variables does not pose any safety problems beyond those inherent with most synchronization primitives, such as deadlock and priority inversion. It would be unsafe for an application to manipulate the mutexes and condition variables used by the Ada RTS, but the application has no access to these.

Safety of these operations rests on the assumption that each Ada task corresponds to a unique thread. Some Ada compilers have task optimizations which can implement a task without a separate thread of control. For example, the Verdix compiler used in this project supports a *passive task* optimization. **Accept** statements within a passive task are optimized to critical regions protected by a semaphore; they are executed in the caller's thread of control, eliminating the need to do a context switch [14]. However, this means that synchronization operations appearing in the body of a passive task may actually be executed by several different threads. The problem is that the result of locking a mutex in one thread and unlocking it in another is undefined. This could happen if one **accept** statement in a passive task locked a mutex and another unlocked it. In the case of the Verdix compiler, tasks are only optimized in this way if explicitly requested with a pragma, so the application programmer would know when these operations might pose a risk.

6.3 Thread-Specific Data

Pthreads provides an interface for associating data with a thread, based on keys. Each thread can associate a pointer value with each key; the value associated with a given key is unique to each thread. The thread can retrieve this pointer by providing the key. This provides program-wide, per-thread global data.

This functionality is not available in Ada. Though each Ada task can have its own data, only units lexically nested within the task can use that data. If the task calls subprograms not lexically nested within it (e.g. in library-level packages), the local data of the task is not accessible to them. The Pthread mechanism is not subject to this limitation.

A thin Ada binding could present the per-thread data as a value of type `System.Address`. Converting this to and from another type (an access value, for example) can be done via `Unchecked_Conversion`. In this case there is no way to assure that the value stored and retrieved are of the same type, and that there are no dangling references to deallocated storage.

This problem can be addressed with a thicker binding, using a generic to associate data of a particular type to the thread. The generic manages the keys, so that the application program can never use the key to access an object of the wrong type. This key management means that the application must indicate when it is finished with the data association provided by a particular instantiation of the generic by calling a destructor function.

6.4 Thread Priority Scheduling

Pthreads supports priority scheduling of threads. The scheduling policies are the same as those defined in 1003.4 for processes. Three scheduling policies must be supported; SCHED_FIFO, SCHED_RR, and SCHED_-OTHER. SCHED_FIFO is conceptually described as a series of FIFO queues, one for each priority. Threads which are eligible to execute are kept in the queue for their priority. The thread at the head of the highest priority nonempty queue is the currently executing thread. When a thread is preempted by a higher priority thread, it remains at the head of its queue; when a task suspends, it is placed at the tail of its queue. SCHED_RR (round robin) is exactly the same as SCHED_FIFO, except that the executing thread will be preempted and placed at the tail of its priority queue after it has executed for a time quantum, deferring to the next thread in the queue. SCHED_OTHER represents an implementation defined scheduling method, which may (but need not) be the same as SCHED_FIFO or SCHED_RR. Other scheduling policies may be provided by an implementation. Policies and priorities can both be changed dynamically on a per-thread basis.

Ada allows static priorities to be assigned to tasks using the `PRIORITY` pragma. Tasks must be scheduled such that a task with a lower priority does not execute if there is a task with a higher priority that is eligible for execution. The priority of an Ada task only changes when the task is activating or is executing the body of an **accept** statement, which must execute at the higher of the priorities of the accepting and calling tasks. The scheduling of tasks for which no priority is specified is implementation-dependent.

The SCHED_FIFO and SCHED_RR policies both support Ada priority scheduling. Defining an Ada program in which none of the tasks have priorities would allow the use of SCHED_OTHER or other implementation dependent scheduling policies without violating Ada task semantics. Access to this capability would require an Ada binding for the functions that set the scheduling attributes of a thread.

The result of scheduling threads with other policies in the same process as threads scheduled under SCHED_-FIFO or SCHED_RR depends on the implementation. Therefore, an Ada program wishing to use implementation defined scheduling should not use Ada priorities in any task, and should explicitly set the scheduling policy of each task it creates. Setting the policy of a newly created task may be a problem, since its thread will already have been created with some scheduling policy (probably SCHED_FIFO) before the application can access the thread ID to change its policy. There will be a small window in which SCHED_FIFO threads are scheduled against threads with implementation dependent policies. This has implementation specific results, but will probably be acceptable on most systems. A better Ada binding requires compiler support, such as a pragma that specifies the scheduling policy to be used, for a particular task or throughout a compilation.

Similar cautions apply to the functions which dynamically change the priority of a thread; a binding to these functions should not be used by an application to change the priority of a task for which a static Ada priority is already defined.

The binding must also provide a means for the application to obtain the range of priorities which can be assigned to threads. This will be implementation dependent, but so is the range of Ada priorities. In an Ada RTS implemented using Pthreads, these ranges should be the same.

Pthreads provides (optional) priority inheritance and priority ceiling policies on mutexes [12, 3]. These protocols bound priority inversion, aiding in the predictable scheduling of real-time tasks. Again, such mutexes should only be used in tasks without Ada priorities, lest they violate Ada task semantics by changing the task priority where Ada specifies it should remain fixed.

Another scheduling attribute that can be modified is contention scope. This specifies how threads will contend with threads in other processes. Threads with local contention scope are scheduled within the process; when the process is scheduled, the threads with local contention scope within it compete for the processor. Threads with global contention scope (available only with kernel implementations) contend with all other threads on the system. Ada does not specify how tasks contend for processor time with entities outside of the program, so it should be acceptable for the application to change the contention scope of task threads.

6.5 New Process Creation

Pthreads specifies that when a process forks only the thread calling *fork()* is replicated in the new process. This is a problem if tasks are implemented as threads. For example, if a task which was dependent on another task called *fork()*, the new process would contain an executing thread whose master does not correspond to any thread in that process. Since 1003.5 already says that the effect of forking a process with more than one active task is unspecified, no change is needed for a Pthreads binding.

6.6 Signals

The normal POSIX method of handling signals is for a process or thread to pass a pointer to the handler function pointer to *sigaction()*. This handler is then called in the context of the process or thread (that is, using its stack) by the POSIX implementation when a signal is delivered.

Pthreads specifies per-thread signal handling; signals are delivered and handled by threads. When a signal is directed at a process, it is handled by choosing a thread to call the signal handling function. Signals can also be directed at individual threads explicitly using the *pthread_kill()* function, or implicitly by thread execution (e.g. SIGFPE is delivered to a thread when it divides by zero).

In 1003.5 the handling of signals delivered to processes is via Ada interrupt entries. Pthreads signal handling can be treated similarly, but there are some unique problems. Since the Ada task threads may coexist with other threads, either created by the program or by libraries implemented in other languages, there may be other threads attempting to handle the signal besides the one containing an Ada interrupt entry. When a signal is directed at the process, it will be delivered to only one of the threads. This could cause the Ada task to miss signals. However, this will be a problem in any language: if a C-language program uses a multithreaded library that defines signal handlers the library may intercept signals that the program expects to receive itself, or *vice versa*. Libraries will have to document any signals that they use.

6.7 Thread Cancellation

The *pthread_cancel()* function is used by our RTS to abort a task. Since some functions must be executed before the **abort** statement completes, it is not always possible for cancellation handlers to do all of the work of abortion in a timely manner. For this reason, part of the abortion of target tasks is done by the **abort** statement implementation itself (see Section 5.7). A direct binding to *pthread_cancel()* would be unsafe using this implementation.

The cancellation handler can be modified to do all necessary termination processing. The principal reason that this has not been done is that tasks which in entry calls cannot be aborted while the rendezvous is taking place. To protect these calls, cancelability has to be disabled; an expensive operation. Since a task must be abortable if it is waiting on an entry but not in rendezvous, the task must remain cancelable until the rendezvous starts. A task can only disable its own cancelability, so an entry call would have to suspend, be reawakened by the accepting task at the start of rendezvous, disable its cancelability, and suspend again to wait for the end of rendezvous. All of this logic and extra context switches in rendezvous make safe use of *pthread_cancel()* very expensive.

Other functions are provided for controlling when a cancellation request by *pthread_cancel()* can take effect, and defining cleanup handlers to be executed before the thread terminates. A thread can control whether it can be canceled or not. In our RTS, where abortion is implemented via cancellation, deferring thread cancellation translates into deferring abortion. This is permitted by Ada semantics, which specify that an abnormal task may (but need not) continue to execute until it reaches a synchronization point (see Section 5.7). Since the synchronization points are all in the RTS, the RTS can make sure that the thread can be canceled at those points.

Cleanup handlers are specified as function pointers, so that they have many of the safety problems outlined in Section 6.1. Bindings to the cleanup handler definition functions *pthread_cleanup_push()* and *pthread_cleanup_pop()* can be provided with the same caveats that apply to *pthread_create()*, that is, that only library level, non-generic subprograms are used as cleanup handlers, and that the binding provides a wrapper to catch all exceptions. As with *pthread_create()*, these will still be unsafe and should be placed in a package whose name indicates that fact.

The operations for pushing and popping cleanup handlers are unsafe in general, since the RTS may rely on the content of the cancellation handler stack. In particular, in our RTS the abortion mechanism relies an abortion handler being at the bottom of the cancellation handler stack. It would be disastrous for a user to pop this.

6.8 Calling Ada Code from Threads

Whether or not the Ada binding provides a way to create non-task threads, it is possible for non-task threads to exist, and to attempt to execute Ada code. An Ada application may use an off-the-shelf package that is implemented in C with Pthreads. For example, this is likely to be the case with future window-based user interfaces. Such C-language code may create threads of its own. If the interface involves call-backs of Ada subprograms, threads created by the C-language code may attempt to call Ada procedures. In a complex system, one may have arbitrary nesting of calls, C-to-Ada, Ada-to-C, etc.

A non-task thread may try to create a task, abort a task, call an entry, or execute a **delay** statement. It cannot attempt to accept an entry, since Ada does not allow accept or selective wait statements outside of a task body. Abortion of a task need not be a problem. In our implementation, abortion is an operation on the task being aborted; the aborter does not have to be a task. In particular, the aborter's ATCB is not referenced by the abortion operation. Likewise, if the **delay** statement is implemented using the 1003.4 *nanosleep()* function it can work for a non-task thread. There is a problem with the creation of a task, since the procedure acting as the body of the thread becomes the master of the new task. Termination processing must be done on completing this procedure. In our implementation, the parent task is used to coordinate this processing, using structures in the parent's ATCB, but in this case there is no parent task, only a thread. There is a similar problem with calling the entry of another task. In our implementation, this can require that the caller's ATCB be placed on an entry queue in the acceptor. Mutexes and condition variables in the caller's ATCB are used in the coordination of the rendezvous, and these would be missing as well.

One way to support the execution of Ada task operations from non-task threads is to create an ATCB for the thread on demand [5]. A null ATCB is associated with the thread on creation. Each RTS call that may need to access the calling thread's ATCB checks whether the calling thread has one. If not, the RTS creates one and associates it with the thread before proceeding. This avoids most of the overhead in *pthread_create()*, at the expense of adding overhead to some Ada task operations to check whether the ATCB exists. Recovering the storage of the ATCB, and breaking potential dangling references from RTS data structures when the thread terminates is also a problem. Conceivably this could result in failure of the RTS. Some degree of safety could be achieved by adding a thread wrapper with an exception handler and cancellation handler that will always deallocate the ATCB and unlink references to the thread from RTS data structures before the thread terminates. Of course, these measures can be defeated if the thread calls C code that makes malicious use of Pthreads operations.

7 Conclusions and Future Work

Though our Ada RTS is still under development, there does not appear to be any serious obstacle to its completion. We have found no insurmountable problems in using Pthreads to implement Ada tasks. Preliminary investigation also indicates that most of Pthreads functionality can be safely used by an Ada application program.

Despite these results, Pthreads may not be a good thing for all POSIX applications. It is clear that implementing Ada tasks over Pthreads involves more overhead than if tasks are implemented directly. The Pthreads specification leaves many details to the implementation; writing portable code using Pthreads primitives will require considerable care. It also appears that some features of the Pthreads (notably per-thread signal state) may make them inherently more costly to implement than Ada tasks.

In the time remaining in this project, we intend to complete the Ada RTS and obtain performance data on it. This will provide some indication of the loss of efficiency due to the use of the standardized interface. We also hope to have time to implement and test a binding to the remainder of the Pthreads services.

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 runtime system. Contributions to this paper by Ted Baker were supported in part by the Software Engineering Institute, at Carnegie Mellon University.

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.

References

[1] Ada Runtime Environment Working Group: MRTSI Task Force. A model runtime system interface for ada. Technical report, ACM SIGAda, 1990.

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

[3] T.P. Baker. Stack-based scheduling of realtime processes. Technical report, Florida State University, Department of Computer Science, Tallahassee, Florida, 1990.

[4] T.P. Baker. Realtime extension for portable operating systems: "thin" ada binding. Technical report, Florida State University, Department of Computer Science, Tallahassee, Florida, 1992.

[5] Bevin Brett. Personal communication.

[6] David K. Hughes. Ada tasks as POSIX pthreads, with optional extensions: An extended tasking model approach to binding Ada to P1003.4a. POSIX note P1003.5-DAL-00, April 1992.

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

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

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

[10] Ragunathan Rajkumar, Lui Sha, John P. Lehoczky, and K. Ramamritham. An optimal priority inheritance protocol for real-time synchronization. Technical report, Carnegie Mellon University, 1989. Submitted for publication.

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

[12] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols - an approach to real-time synchronization. Technical report, Carnegie Mellon University, Departments of CS, ECE and Statistics, Pittsburgh, Pennsylvania, 1987.

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

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