The Implementation of the Priority Ceiling Protocol in Ada-2005

Albert M. K. Cheng and James Ras
Real-Time System Laboratory
Department of Computer Science
University of Houston
Houston, TX 77204, USA
cheng@cs.uh.edu, jras@cs.uh.edu

Abstract

Ada 2005 is an even safer and more agile language than its predecessors, with all of the efficiency that is a hallmark of the Ada language design. It still has a strong focus on real-time response, multi-tasking, and sound engineering. It is the only international standard for object-oriented real-time distributed programming. High-integrity systems rarely make use of high-level language features such as Ada Protected Objects. Protected Types provide a passive mechanism for communication and synchronization between tasks. In this paper, we present some of these features, which are implemented using the Priority Ceiling Protocol (PCP). PCP is an extension of the Priority Inheritance Protocol (PIP), with the added features of preventing deadlocks and priority inversions. Priority inversion occurs when a high priority task is blocked for an unbounded time by lower priority tasks. Unfortunately though, PCP is not known to be supported by any operating systems. This article describes an implementation of PCP in Ada-2005. A detailed discussion of this protocol, and other related issues, are presented.

Introduction

When Ada was first released nearly 25 years ago its developers, and potential consumers, had high expectations. Unfortunately, it came out looking large and complicated. Its approach to multi-tasking, although considered safer for concurrent and real-time systems programming, proved unsuccessful. It was built on the model of rendezvous, which is elegant in theory, but was an unnecessary departure from the more practical tradition of threads and message passing. Ada 95 remedied this problem, and further evolved into Ada 2005. Now, the multi-tasking facilities in Ada are conceptually similar to other languages, but better, and less error-prone.

Ada compilers must pass a significant test suite to become validated. There are currently three standardized versions, Ada 83, Ada 95 and, the most recent, Ada 2005 and 2006. Ada 2006 is an important language. There has been much discussion on whether the language should be called Ada 2005 or Ada 2006. For various reasons the WG9 meeting in York in June 2005 decided that the vernacular name should be Ada 2005, although now, the names are used interchangeably.

This article describes Ada's object oriented programming support for real-time programming. It focuses on Ada's support for real-time scheduling. Ada was designed to meet the predictability and reliability requirements of dependable systems. Over time, those kinds of systems, which are typically embedded real-time systems, became Ada's main responsibility. Ada supports hard real-time. In hard real-time, computations must finish by certain, absolute deadlines to avoid dire consequences. If an appropriate scheduling algorithm is used, a set of periodic tasks predictably meets its deadlines. Ada supports rate-monotonic and earliest deadline first scheduling. In this article we look at the implementation of the Priority Ceiling Protocol. A detailed discussion of this protocol and other related issues are presented.

Real-Time Scheduling

In real-time systems, it is of paramount importance that the end time adheres to the time constraints of tasks. A tremendous amount of research has been carried out on scheduling problems associated with such systems, primarily focusing on protocols to schedule concurrent tasks to meet deadline requirements. The rate-monotonic scheduling (RMS) is the best-known pre-emptive fixed priority scheduling algorithm for scheduling periodic tasks. RMS is an optimal system, in the sense that if a set of tasks can be scheduled by any static priority algorithm, then RMS will be able to schedule that task set. RMS bases its schedulability analysis on the processor utilization level, below which all deadlines can be met [3].

RMS calls for the static assignment of task priorities based upon their period; the shorter a task's period, the higher its priority. For example, a task with a 1 millisecond period has higher priority than a task with a 100 millisecond period. If two tasks have the same period, then RMS does not distinguish between the tasks. RMS allows application designers to ensure that tasks can meet all deadlines, even under transient overload. However they do not know exactly when any given task will be executed by applying proven schedulability analysis rules. The problem occurs when the lower priority job gains access to a shared resource, and then a higher priority job requests access to the same shared data. The higher priority job is blocked, until the lower priority job completes its access to the shared data. To solve this problem, semaphores are commonly used to block one task, while another one is using the resource. This blocking, in a real-time system, can cause tasks to miss deadlines. Mok proved that unrestricted use of semaphores in real-time systems causes the schedulability problem to be NP-complete [8]. For pre-emptive, priority based scheduling, mutual exclusion leads to a problem called inversion. *Priority inversion* occurs when a high priority task is blocked, while a lower priority task uses a shared resource. If a medium priority task pre-empts the lower priority task while it holds the lock, the blocking time of the high-priority task becomes unbounded [9, 11].

For example, Figure 1 shows a situation where two tasks share the same resource S. An invocation of low-priority task L starts executing first and locks the resource. Then when a high priority task H arrives, it pre-empts task L. However, after executing for a short time, task H also needs to access the shared resource. When it attempts to do so it finds that S is already locked by task L. At this point task H is blocked, and control is returned to the low-priority task. Task L then continues executing until it finishes using shared resource S and releases the lock. At this point task H is free to continue, so it immediately pre-empts task L and accesses S.

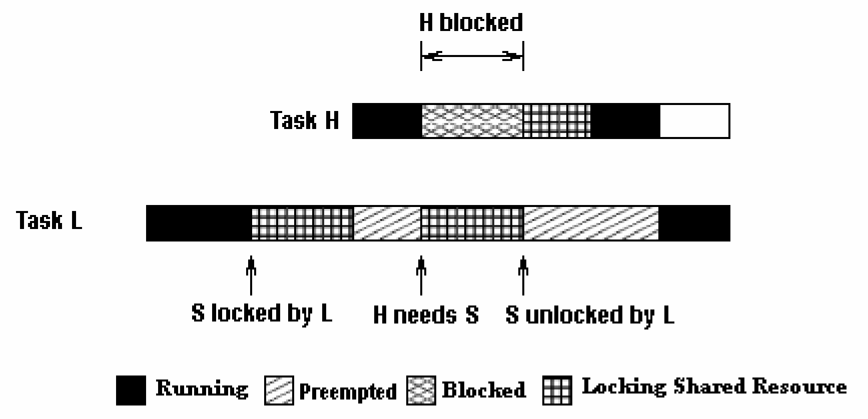

Fig. 1. Blocking by a lower-priority task.

The problem with blocking is that the analysis of the timeliness becomes more difficult. When Task H is blocked, the system is said to be in a state of priority inversion, because a lower-priority task has the thread focus even though a higher-priority task is ready to run. One can imagine third and fourth tasks of intermediate priority that do not share the resource, are therefore able to run and pre-empt Task L, thereby lengthening the amount of time before Task L releases the resource, and allows Task H to run.

For example, Figure 2 shows the behavior of a task set in which tasks L and H both share resource S. At first, the behavior is the same as in Figure 1. After task H becomes blocked. control returns to task L. However, when a medium-priority task M arrives, it has a higher priority than task L, thus, task M can start executing immediately. Indeed, medium priority tasks can execute indefinitely at this point, even though a high priority task H is waiting. Because an arbitrary number of tasks can be fit in the priority scheme between Task H and Task L, this problem is called *unbounded priority inversion* and is a serious problem for the schedulability of tasks. Therefore, to improve the performance of real-time applications, the duration of priority inversion must be minimized.

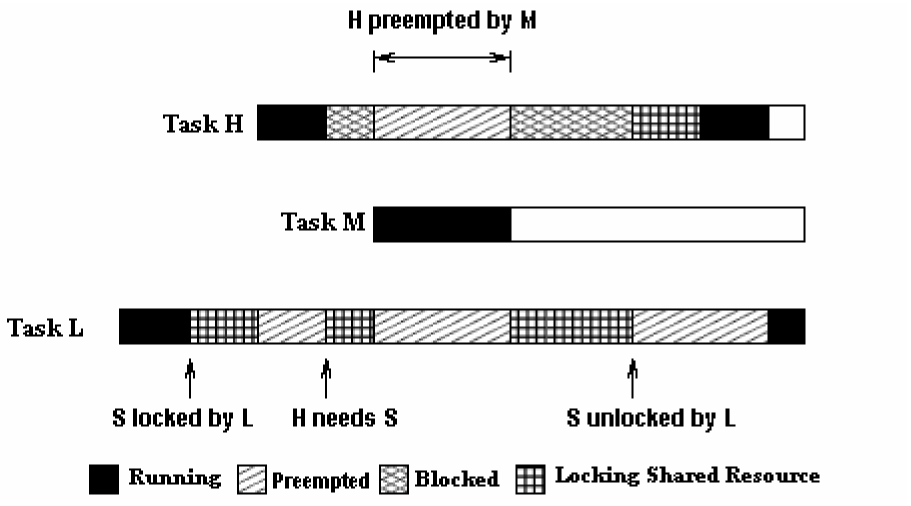

Fig. 2. Example of priority inversion.

The priority inversion problem is unavoidable whenever lock-based synchronization (i.e., mutual exclusion) is used to maintain consistency. Synchronization primitives such as locks, barriers, semaphores, and protected objects offer convenient options suited for situations requiring mutually exclusive access to resources. A semaphore can be viewed as a protected variable, whose value can be modified only with the Request_Semaphore and Release_Semaphore procedures. A binary semaphore is restricted to values of True or False.

A binary semaphore can be used to control access to a single resource. In particular, it can be used to enforce mutual exclusion for a critical section in user code. In this instance, the semaphore would be created with an initial value of False, to indicate that no task is executing the critical section of code. Upon entry to the critical section, a task must call the Request_Semaphore procedure to prevent other tasks from entering the critical section. Upon exit from the critical section, the task must call the Release_Semaphore procedure to allow another task to execute it.

Priority inversion cannot be eliminated entirely. However, it is possible to bound the waiting time and thereby avoid unpredictable delays. Previous work has addressed the priority inversion problem, and proposed priority inheritance solutions, with the Priority Inheritance Protocol and the Priority Ceiling Protocol [9].

The Priority Inheritance Protocols

A number of resource locking protocols have been proposed for bounding blocking times and avoiding priority inversion. For instance, a simple but potentially inefficient solution is to disallow pre-emption while a task holds a lock [3], and for fixed priority scheduling, the usual solution is some form of priority inheritance protocol [9].

The Priority Inheritance Protocol (PIP) is based on pre-emptive scheduling. The basic idea of PIP is that, when a lower priority task blocks one or more higher priority tasks, its priority is increased to that of the highest priority task blocked waiting for that resource. After exiting its critical section, the lower priority task returns to its original priority level. A high priority task can be blocked by a lower priority task in one of two situations. Firstly, it may be directly blocked, a situation in which a higher priority task attempts to lock a locked semaphore. *Direct blocking* is necessary to ensure the consistency of shared data. Secondly, a medium priority task can be blocked by a lower priority task, which inherits the priority of a high priority task. We refer to this form of blocking as *Push-through Blocking*, which is necessary to avoid having a high-priority task be indirectly pre-empted by the execution of a medium priority task.

For example, Figure 3 shows how the task set behaves under the priority inheritance protocol. When task L locks the shared resource its (active) priority is raised to equal that of task H only when H needs to lock S. This means that when task M arrives it is able to execute without pre-emption until H arrives. After L releases the lock on S, its active priority drops back to its base level, and task H is allowed to lock S. The medium-priority task, M, then must wait for the high-priority task H to complete. And after M is done, L is allowed to complete.

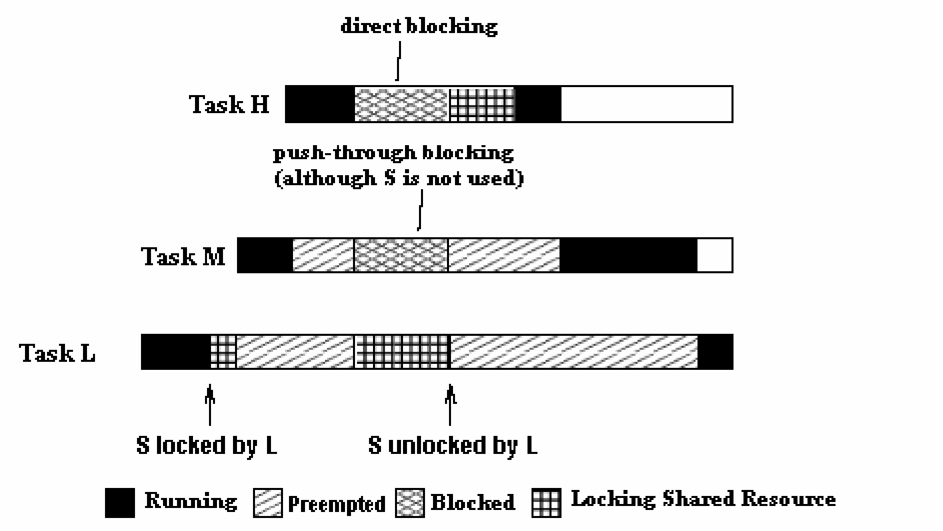

Fig. 3. Priority Inheritance with a shared resource. (PIP)

The advantage of the priority inheritance protocol is that when a low-priority task is blocking a high-priority one, the low-priority task is allowed to execute with less pre-emption, thus allowing control of the resource to go to the high-priority task more quickly. The priority inheritance protocol prevents priority inversion and also bounds blocking times, thus making interacting task sets analyzable. A side-effect of the protocol, however, is that a task may experience blocking on a shared resource even though it does not use the resource. Also, PIP alone does not prevent deadlocks. For example, if two tasks attempt to lock two different semaphores in opposite order, a deadlock is formed. This deadlock problem *can* be solved, if the programmer imposes a total ordering on the semaphore locking. However, there can still be a second problem. The blocking duration for a job, though bounded, can still be substantial, because a chain of blocking can be formed [9].

The Priority Ceiling Protocol

The Priority Ceiling Protocol (PCP) is similar to PIP and is also based on pre-emptive scheduling. PCP also has the following desirable properties: (1) it prevents deadlocks; and (2), it prevents chained blocking, so, a high priority task can be blocked by *at most* one lower priority task, even if the task suspended itself within the critical section. PCP introduces a third type of blocking, *ceiling blocking*, in addition to the *direct blocking* and *push-through blocking* caused by PIP. Ceiling blocking is needed for the avoidance of deadlocks and chained blocking.

The basic premise of PCP is based on the following rules: (1) a lower priority task that blocks a higher priority task J, inherits the priority of task J, and only when the task releases the semaphores it holds, will its priority be restored to its ordinary value. (2) A task J can only lock a semaphore S if: (a) the semaphore S is not yet locked, and (b) the priority of task J is greater than the *priority ceilings* of all semaphores that are currently locked by tasks *other* than task J. The priority ceiling of a semaphore is defined as the highest priority task of all those that request to lock that semaphore at any time [10].

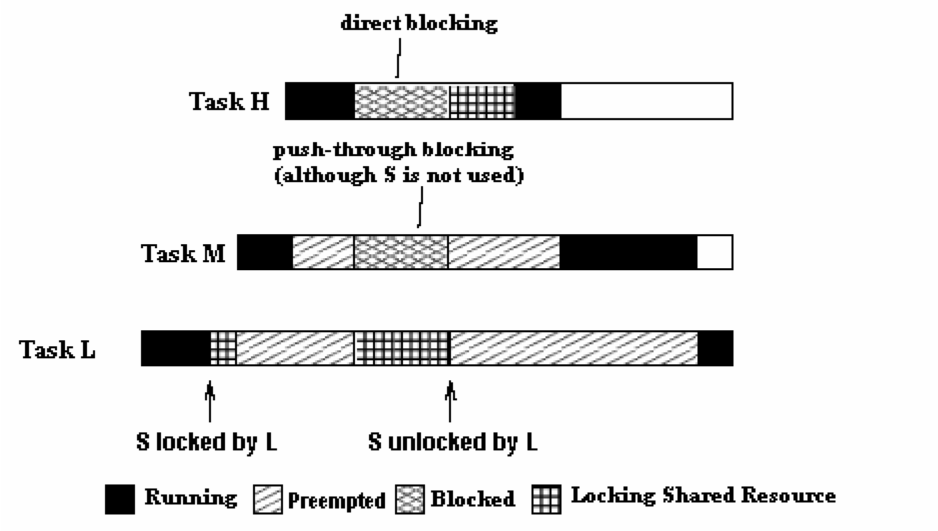

Fig. 4. Ceiling locking of a shared resource. (PCP)

For example, Figure 4 shows how the task set from Figure 3 behaves under the ceiling locking protocol. PCP behaves as PIP does when there is only a single shared resource. The ceiling priority associated with shared resource S is 'H'. When task L locks the shared resource, its (active) priority is raised to equal that of task H only when H needs to lock S. This means that when task M arrives it is able to execute without pre-emption until H arrives. After L releases the lock on S, its active priority drops back to its base level, and task H is allowed to lock S. The medium-priority task, M, then must wait for the high-priority task H to complete. After M has concluded, L is allowed to complete.

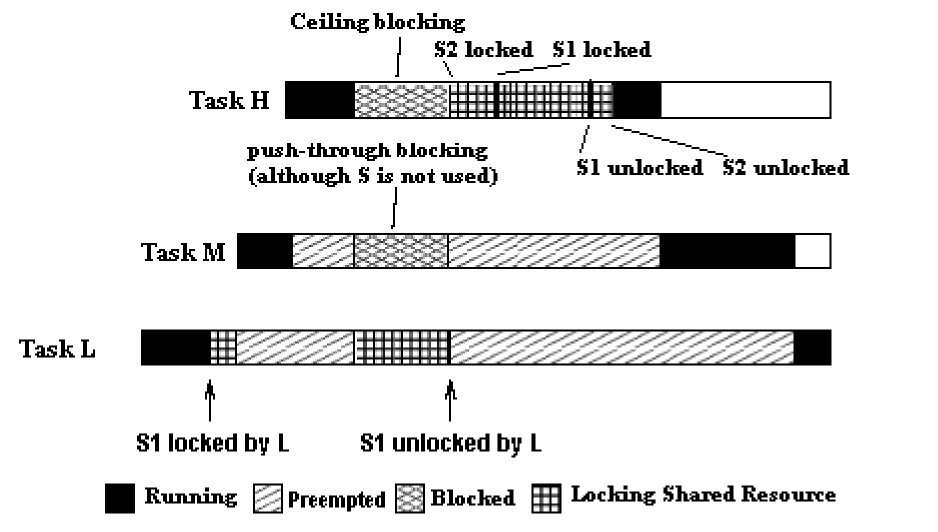

Fig. 5. Example of Ceiling blocking. (PCP)

Figure 5 shows an example where Ceiling Blocking is operating. Note that the priority ceilings of semaphores S1 and S2 are equal to the priority of H. When Task H attempts to lock S2 which is free from utilization, it is prevented since its priority is not greater than the priority ceilings of all semaphores that are currently locked by L. This is a new form of blocking introduced by the priority ceiling protocol, in addition to the direct and push-through blocking encountered in PIP. At this point, Task L resumes its execution of the critical section at the newly inherited priority level of H. When Task L unlocks S1, L returns to its base priority level, and Task H is awakened and is allowed to lock S2.

Related Work in PCP for Ada

In its basic structure, as described above, the priority inheritance protocol can be expensive to implement because the run-time scheduler must keep track of which task holds which locks. Fortunately, a slight variant of priority inheritance has proven to be almost as effective and dramatically cheaper to implement. Ada provides a different solution to the problem of priority inversion. This is the Ceiling Semaphore Protocol, also known as the Highest Locker protocol, or the Ceiling Locking protocol (CLP) within the Ada community.

Now, digressing slightly from the Ceiling Locking protocol, into Ada: Ada 83 specified pre-emptive task scheduling based on static priorities, but left certain aspects implementation dependent. If the priorities of both tasks engaged in a rendezvous are defined, the rendezvous is executed with the higher of the two priorities. If only one of the two priorities is defined, the rendezvous is executed with at least that priority. If neither is defined, the priority of the rendezvous is undefined. Ada 95 provided dynamic priorities for tasks, but ceiling priorities for protected objects are statically assigned by means of a pragma when the object is created. The semantic complexity, as well as the potential inefficiency of any resulting implementation was the main objections to including dynamic ceilings in Ada 95. Nevertheless, the ability to dynamically change protected object ceiling priorities is especially required in situations where dynamic task priority change occurs, or where a library containing protected objects with inadequate ceilings is used by an application-specific set of tasks and interrupt handlers. Now, in Ada 2005, this is possible.

The Ceiling Locking Protocol is the simplest of the priority inheritance protocols. The Ceiling Locking Protocol is similar to PCP in its use of the ceiling priority, but it has a different set of rules on how a task set behave under the ceiling locking protocol. The protocol is also based on pre-emptive scheduling and has the following rules:

1. A task may lock a protected object if it is not yet locked.

2. When it enters a critical section it immediately inherits the priority ceiling of the protected object, and recovers its entry priority when it exits the section.

The ceiling priority of the protected object is the upper bound of the active priorities of all tasks that may employ the functions of the protected object. This protocol effectively prevents any task from starting to execute until all the shared resources it needs are free.

For example, Figure 6 shows how the task set from Figure 4 behaves under the ceiling locking protocol. The ceiling priority associated with shared resource S is 'H'. When task L locks the shared resource its (active) priority is immediately raised to equal that of task H. This means that task L executes without pre-emption until it releases the lock and its active priority drops back to its base level. While L has the resource locked, task H arrives, then task M, but both are blocked until L releases the lock on S. The medium-priority task, M, then must wait for the high-priority task H to complete.

Most importantly, the ceiling locking protocol has several desirable properties:

+ It is cheap to implement at run time. By raising priorities as soon as a resource is locked, whether a higher-priority task is trying to access it or not, the protocol avoids the need to make complex scheduling decisions while tasks are already executing.

+ The manner in which task priorities are manipulated ensures that a task cannot start until all shared resources it requires are free. This means that no separate mutual exclusion mechanism, such as semaphores, is needed to lock shared resources. The run-time scheduler implements both task scheduling and resource locking using the one mechanism.

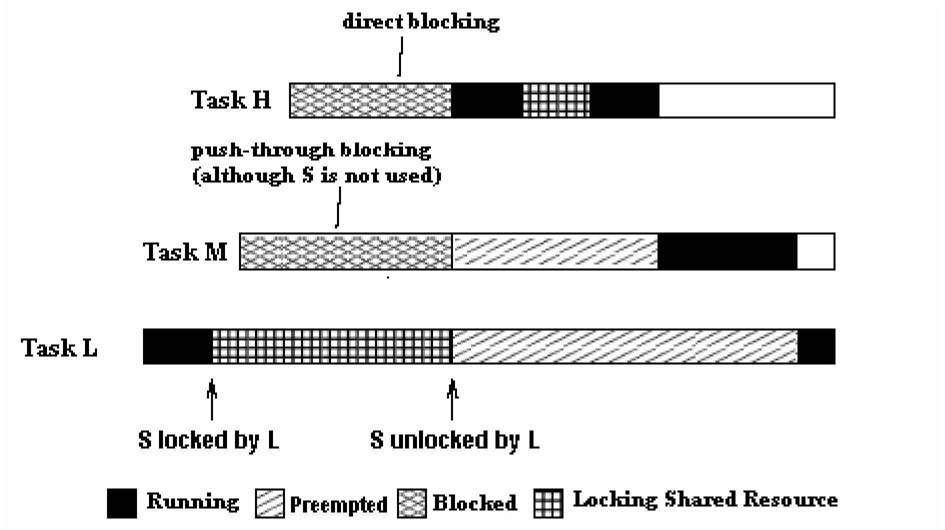

Fig. 6. Ceiling locking of a shared resource. (Ceiling Locking Protocol)

+ Since a task cannot start until all resources it may need are free, run-time deadlocks caused by circular dependencies on shared resources are impossible.

+ Each task can be blocked at most once, at its beginning, by a single lower-priority task. This provides the necessary bound on blocking times needed for analyzing schedulability of interacting task sets.

For these reasons, ceiling locking is the default priority inheritance protocol in the Ada 2005 real-time programming language. However, PCP and PIP are more suitable at times since they adhere to priority scheduling. The priority of a task is raised only when it is necessary to minimize priority inversion. In the Ceiling Locking Protocol, a task that locks a protected object will always have its priority raised to that of the ceiling priority of the protected object during the entire critical section. Thus, whereas the Ceiling Locking Protocol is excellent for many scheduling problems, there are many other scheduling problems where PCP and PIP are more appropriate.

PCP in Ada

Ada 2005 adds a lot of new features to Ada that enhance its safety, readability, and expressiveness. In particular, new enhancements have been made to real-time programming. Ada 2005 provides a catalog of features for implementing PCP efficiently. Semaphores can be implemented as protected objects instead of tasks, thus avoiding additional task overhead. Protected objects can be used directly, to provide mutual exclusion.

Protected Objects

A protected object is a data structure that is encapsulated to make it task-safe, which means that multiple tasks can operate on it, without risk of conflict. A new protected action is not started on a protected object while another protected action on the same object is underway, unless both actions are the result of a single call on a protected function. Protected objects have lock variables, which are inaccessible to the programmer. The compiler includes the necessary operations in the locks. A protected type can have protected operations of three kinds: functions, procedures, and entries [12].

During a protected action, it is a bounded error to invoke an operation that is potentially blocking. The following are defined as potentially blocking operations: a select_statement, an accept_statement, an entry_call_statement, a delay_statement, an abort_statement, and a task creation or activation. Some of these operations are not directly blocking. However, they are still treated as bounded errors during a protected action, because allowing them might impose an undesirable implementation burden.

The innovative rendezvous of Ada 83 provides a good overall model for task communication, through which tasks are mutually excluded automatically and avoid the difficulties created by using semaphores. In this process a user of shared data must remember to lock and unlock semaphores. However, it has not proved adequate for the problems of shared data access. A serving task is often needed to control access to shared data or shared resources, which gives systems a relatively high overhead, and increases possibilities of race conditions. Consider the problem of protecting a variable Semaphore from uncontrolled access. Take a look at the following request/release scheduler package:

```
package body Scheduler is

        procedure Release (S : Positive) is

        begin
          Semaphore_Status(S)  := False;       -- not locked
        end Release;

        procedure Request (S : Positive) is
        begin
          Semaphore_Status(S)  := True;        -- locked
end Request;

end Scheduler;
```

However, this is unsatisfactory. Nothing prevents different tasks in our system from calling **Release** and **Request** simultaneously and thereby causing interference. The proper solution is to implement semaphores using protected objects. However, before granting the lock of a semaphore, it is required that PCP checks the status of all semaphores that are locked. Implementing each semaphore as a separate protected object requires complicated communications and synchronizations. For this reason, a single protected object acts as the scheduler for granting operations on semaphores.

There are two important observations to be made concerning the implementation. First, for the scheduler to execute when needed, its priority must be higher than that of any client task which may call it [2]. Second, although a simple protected type for the semaphore is more appropriate for the implementation, with a priority value as a discriminant, still the code illustrates the power of Ada 2005. If a number of protocols, for example, PCP, PIP and CLP, implement the same protected interface Scheduler, we can then reference their instances by the Scheduler'Class. It is also useful to use this approach based on requeue for abstractions such as Events. For example, given that the body of the protected type contains the logic of the procedure Initialize, this would allow us to give unique information to each semaphore type, such as a semaphore number or a priority value.
Now consider the following:

```
protected  type Semaphores_Scheduler
   (N: Positive; Scheduler_Priority: Any_Priority) is new Scheduler with

        pragma Priority(Scheduler_Priority);

        procedure Release (tid : Task_ID; S : Positive)
        entry Request (tid : Task_ID; S : Positive);

  private
        Semaphore :        Boolean := False;
        Current_Queue:   Boolean := False;
        Ready_Queue:     Boolean := False;
        entry Queue_Again  (Boolean);

end Semaphores_Scheduler;
```

```
protected body Semaphores_Scheduler is

        procedure Release (tid : Task_ID; S : Positive)  is
        begin
          Semaphore_Status(S)  := False;
        end Release;

        entry Request (tid : Task_ID; S : Positive)  when True is
        begin
      if(Semaphore_Status(S) = False) then
           Semaphore := True;
      else
         requeue Queue_Again;
      end if;
end Request;

        entry Queue_Again  (for Queue in Boolean) when
                          (Queue = Current_Queue) and
                                (Ready_Queue = Current_Queue) is
        begin
               ...
        end Queue_Again;

end Semaphores_Scheduler;
```

In this implementation, the protected object *Semaphores_Scheduler* provides controlled access to the private variable *Semaphore*. It provides only two operations in its public interface: the procedure *Release* and the entry *Request*. The Request operation is implemented as an entry since it contains a requeue statement. It sometimes happens that a request for service needs to be provided in two parts and that the calling task has to be suspended after the first part until the conditions are such that the second part can be done. Once the requeued call has been executed, control is returned to the task that made the original call.

The entry, Queue_Again, is needed because it is not possible to use only one requeue entry. If only one requeue entry were used, a task that was not granted a semaphore may need to be requeued to the same entry. Since the entry is priority-based, the same task will be immediately executed again. Therefore, two queues are used: one queue is used when trying to satisfy requests, and requests that cannot be satisfied are reassigned to the other queue. Then, the roles of the two queues are swapped. This avoids problems when the calling tasks have different priorities.

Protected Interfaces

One of the most important object-oriented features of Ada 2005 is the introduction of Abstract types and Interfaces. It is sometimes convenient to declare a type solely to act as the foundation upon which other types can be built by derivation, and not in order to declare objects of the type itself. Interfaces allow polymorphism. This addresses the awkwardness in Ada 95, which introduced both protected types and extensible types, but did not combine the two by allowing protected types to be extended [1, 13].

An interface consists solely of a set of operation specifications: an interface type has no data components and no operation implementations. A type may implement multiple interfaces, but can inherit code for primitive operations from only one parent type. This model has much of the power of multiple inheritance, but without most of the constructs that causes anomalies in inheritance and concurrency. The anomaly is a failure of inheritance and concurrency to work well with each other, negating the usefulness of multiple inheritance as a mechanism for code-reuse in a concurrent setting. When deriving a subclass through inheritance, the presence of synchronization code often forces method overriding on a scale much larger than when synchronization constructs are absent, to the point where there is no practical benefit to using inheritance at all. However, if we are able to modify or extend synchronization constraints in a derived type, so that they are compatible with the parent's constraints, without breaking encapsulation then we have solved the inheritance anomaly.

The combination of the interface mechanism with the concurrency feature means that it is possible, for example, to build a system with distinct scheduling protocols that provide similar interfaces.
Let us look at an example:

```
type Scheduler is protected interface;

  procedure Initialize
        (Object: in out Scheduler;
        Semaphores_Ceiling_Priority: Semaphores_Priorities) is Null;

  procedure Request_Semaphore
        (Object: in out Scheduler;
          tid: Task_ID; S: Positive) is abstract;

procedure Release_Semaphore
        (Object: in out Scheduler;
          tid: Task_ID; S: Positive) is abstract;
```

We define a *Semaphores_Scheduler* type that implements the protected interface *Scheduler* and all its operations. Therefore, the protected type *Semaphores_Scheduler*, must implement all of the abstract primitives of the interfaces concerned. The body of the *Semaphores_Schedule* must contain the logic of the procedures *Request_Semaphore* and *Release_Semaphore* [2]. We do not have to provide an actual body for the procedure *Initialize*. This is a new improvement to the OO model in Ada 2005. Such a null procedure behaves as if it has a body consisting of just a null of

```
procedure Initialize is
begin
        null;
end Initialize;
```

It will be inherited by a type derived from *Scheduler* and it could then be overridden by a procedure with a conventional body. There is of course no such thing as a null function because a function must return a result. Null procedures make interface types much more useful.

Dynamic Ceiling Priorities

In Ada 95, the ceiling priority of a protected object is constant and thus it can only be set once, by means of a pragma Priority, when the object is created. In contrast, task priorities may be set dynamically. Nevertheless, the ability to dynamically change protected object ceiling priorities is especially required in situations where dynamic task priority change occurs. Now in Ada 2005 it is possible to change the priority of a protected object.

In our implementation the scheduler grants requests on all semaphores. In the private part of the scheduler, the array Semaphore_Ceiling_Priority stores the ceiling priorities of the semaphores in the application. The priority ceiling to each semaphore is equal to the highest priority task that may use that semaphore. The Boolean array Semaphore_Status is used to indicate whether the semaphore is locked or not. When the scheduler grants a request to lock a semaphore, the task_id and the priority of the task that locked that semaphore are stored in the appropriate private arrays Semaphore_Locking_Task_ID and Task_Orginal_Priroity respectively. Note that a task executes at its assigned priority except when a higher priority task requests the same locked semaphore. In this case, it executes at the active priority of the calling task. The array Task_Current_Priroity is used to store the active priorities of the locking tasks.

Semaphore Request Operations

The scheduler has the entry Request and the procedure Release. When a task calls a Request to lock a semaphore, the parameters, Task_ID and S, the index of that semaphore, are passed to the entry Request of the scheduler. Since the barrier of the entry Request is always true, the entry body executes. The function Lock_Condition determines if the calling task's priority is greater than the priority ceiling of all semaphores that are currently locked by other tasks. If the locking conditions are satisfied, the semaphore is allowed to be locked. If the semaphore cannot be locked, the active priority of the locking task is raised to equal that of the calling task. The calling task is requeued to future completion of the Request operation after the locking task releases the lock.

Semaphore Release Operations

When a task calls a Release operation to release a semaphore, the parameters, Task_ID and S, the index of that semaphore, are passed to the procedure Release of the scheduler. If the calling task does indeed have a lock on the semaphore that it is trying to release, then the task's active priority is reset to its original priority, and the semaphore's status is reset to False, for unlocked. If some task is waiting in the entry Queue_Again, it is awakened and its requests are re-examined. If the locking conditions still cannot be satisfied, then it is requeued to future completion.

Conclusions

The priority ceiling protocol addresses the priority inversion problem, i.e., the possibility that a high-priority task can be delayed by a low-priority task. Under the priority ceiling protocol, a high priority task can be blocked at most once by a lower priority task. This paper defines how to apply the protocol to Ada-2005. An extensive example illustrating the protocol in Ada is given.

From a theoretical point of view, the Ceiling Locking Protocol produces less overhead than PCP, due to less context switching and less time-consuming lock and unlock operations. However, PCP is more suitable because it adheres to priority scheduling. The priority of a task is raised only when it is necessary to minimize priority inversion. In Ceiling Locking Protocol, a task that locks a protected object will always have its priority raised to that of the ceiling priority of the protected object during the entire critical section. Thus, whereas the Ceiling Locking Protocol is excellent for many scheduling problems, there are many other scheduling problems where PCP and PIP are more appropriate.

Acknowledgments

We want to thank Dr. John B. Goodenough and Dr. Lui Sha for their work on the Priority Inheritance Protocols and Ada. And special thanks to all the researchers and volunteers who made Ada 2005 possible.

References

[1] A. Burns and A. J. Wellings, Real-Time Systems and Programming Languages, 3rd. ed: Addison-Wesley, 2001.

[2] Aburas, Jim. "Optimal Mutex Policy in Ada 95", Dec. 1995, ACM Press, vol. XV, issue 6, pp. 46-56.

[3] ACM Ada Letters, Proceedings of the 8* International Real-Time Ada Workshop: Tasking Profiles (September 1997).

[2] Burns, A. and Wellings, A.J. Concurrency in Ada, Cambridge University Press (1995)

[3] Buttazzo, G. C. 1997. Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications. Kluwer.

[4] Liu, C. & Layland, J., Scheduling algorithms for multiprogramming in hard real-tie environments. Journal of the ACM, Vol.20, N0.1, 1973.

[5] Miranda, J., GNAT and Ada 2005, Reference Manual, January 2005.

[8] Mok, A. "Fundamental design problems of distributed systems for the hard real-time environment," PhD thesis, 1983.

[9] Sha, L., Rajkumar, R. & Lehoczky, J., Priority Inheritance Protocols: An Approach to Real-Time Synchronization. IEEE Transactions on Computer, September 1990.

[10] Sha, L. & Goodenough, J., Real-time scheduling theory and Ada. IEEE Transactions on Computer, Vol. 23, No. 4, April 1990.

[11] Sha, L., Rajkumar, R. & Lehoczky, J., Generalized rate-monotonic scheduling theory: a framework for developing real-time systems. Proceedings of the IEEE, Volume 82, Issue 1, Jan. 1994.

[12] Wellings, A.J., R.W. Johnson, B.I. Sanden, J. Kienzle, T. Wolf, and S. Michell. "Integrating Object-Oriented Programming and Protected Objects in Ada 95." ACM 2000: 506-539.

[13] Wellings, A.J. Concurrent and Real-Time Programming in Java. John Wiley & Sons, 2004.

Appendix 1 Implementation of PCP in Ada-2005

```
-- Priority Ceiling Protocol (PCP)

pragma Detect_Blocking;                        -- detect possible blocking in protected section
pragma Restrictions(No_Relative_Delay);        -- no delay statements allowed
pragma Restrictions(No_Select_Statements);     -- no select statements allowed

-- Implementation-defined Restrictions
-- What happens if a task terminates silently?
-- Real-time tasks normally have an infinite loop as their last statement

pragma Restrictions(No_Abort_Statements);      -- no Abort Task statements allowed
pragma Restrictions(No_Task_Termination);      -- Tasks which terminate are erroneous

pragma Detect_Blocking;
pragma Queuing_Policy(Priority_queuing);

with System; use System;
with Ada.Text_IO, Ada.Integer_Text_IO;
with Ada.Task_Identification; use Ada.Task_Identification;
with Ada.Dynamic_Priorities; use Ada.Dynamic_Priorities;

package PCP is

  -- Range of priorities of tasks accessing Semaphores. Ada Default Priority = 15
  TASK_MIN_Priority    : constant Priority     := Priority'First;        -- range  0 .. 30
  TASK_MAX_Priority    : constant Priority     := Priority'Last;

  subtype Task_Priority_Range is Any_Priority
    range TASK_MIN_Priority..TASK_MAX_Priority;

  type NSemaphores is array
    (Positive range <>) of Boolean;

  type Semaphores_Locking_Task is array
    (Positive range <>) of Task_ID;

  type Semaphores_Priorities is array
    (Positive range <>) of Task_Priority_Range;

  type Task_Priorities is array
    (Positive range <>) of Task_Priority_Range;

  Release_Error   : Exception;
  Ceiling_Error   : Exception;
```

```
type Scheduler is protected interface;

    procedure Initialize
      (Object: in out Scheduler;
       Semaphores_Ceiling_Priority: Semaphores_Priorities) is Null;

    procedure Request_Semaphore
      (Object: in out Scheduler;
       tid: Task_ID; S: Positive) is abstract;

    procedure Release_Semaphore
      (Object: in out Scheduler;
       tid: Task_ID; S: Positive) is abstract;

  protected type Semaphores_Scheduler
                                    (N: Positive; Scheduler_Priority: Any_Priority) is new Scheduler with

    -- discriminant N is implementation-dependent
    pragma Priority(Scheduler_Priority);

    procedure Initialize
      (Semaphores_Ceiling_Priority: Semaphores_Priorities);

    overriding
    procedure Release_Semaphore
      (tid: Task_ID; S: Positive);

    overriding
    entry Request_Semaphore
      (tid: Task_ID; S: Positive);

  private
    Semaphore_Status                    :    NSemaphores(1..N)     := (others => False);
    Semaphore_Locking_Task_ID    : Semaphores_Locking_Task(1..N)
                                                                   := (others => NULL_TASK_ID);

    Semaphore_Ceiling_Priority     : Semaphores_Priorities(1..N)     := (others => 0);
    Task_Orginal_Priroity          : Task_Priorities(1..N)           := (others => 0);
    Task_Current_Priroity          : Task_Priorities(1..N)           := (others => 0);
    Current_Queue                  : Boolean                         := True;
    Ready_Queue                    : Boolean                         := True;

    entry Queue_Again
       (Boolean) (tid: Task_ID; S: Positive);

    function Lock_Condition
       (tid: Task_ID;
        Task_Priority: Task_Priority_Range) return Boolean;

  end Semaphores_Scheduler;

end PCP;
```

```
package body PCP is

  protected body Semaphores_Scheduler is

    procedure Initialize
       (Semaphores_Ceiling_Priority : Semaphores_Priorities) is
    begin -- implementation-dependent
      for I in 1..N loop
        Semaphore_Ceiling_Priority(I) := Semaphores_Ceiling_Priority(I);
      end loop;
    end Initialize;

    procedure Release_Semaphore
       (tid : Task_ID; S : Positive) is

    begin  -- the release operation for semaphore S
      if(tid /= Semaphore_Locking_Task_ID(S)) then raise Release_Error; end if;
      Ada.Dynamic_Priorities.Set_Priority(Task_Orginal_Priroity(S),tid);
      Semaphore_Status(S) := False;
      if(Queue_Again(Current_Queue)'Count = 0) then
        -- start and check other queue
        Current_Queue := not Current_Queue;
        Ready_Queue   := Current_Queue;
      end if;

    end Release_Semaphore;

    entry Request_Semaphore
       (tid : Task_ID; S : Positive) when True is
        Task_Priority : Task_Priority_Range := Ada.Dynamic_Priorities.Get_Priority(tid);

    begin  -- before calling Request_Semphore, set tid := Current_Task -- identity of task

      if (Task_Priority > Semaphore_Ceiling_Priority(S)) then
         Ada.Text_IO.Put_Line("Ceiling_Error");
         raise Ceiling_Error;
      elsif((Semaphore_Status(S) = False) and then
            (Lock_Condition(tid, Task_Priority))) then
        Semaphore_Status(S)              := True;
        Semaphore_Locking_Task_ID(S)         := tid;
        Task_Orginal_Priroity(S)        := Task_Priority;
        Task_Current_Priroity(S)        := Task_Priority;
      else -- Semaphore is already locked, locking task inherits higher Priority
        if( Semaphore_Status(S) and then (Task_Current_Priroity(S) < Task_Priority) ) then
          Task_Current_Priroity(S) := Task_Priority;
          Ada.Dynamic_Priorities.Set_Priority(Task_Priority,Semaphore_Locking_Task_ID(S));
        end if;
        requeue Queue_Again(Current_Queue);  -- semaphore still locked, try later
      end if;

      exception
        when others =>
              -- if semaphore locked, release it for the next task
             Release_Semaphore(tid,S);
             raise;
    end Request_Semaphore ;
    entry Queue_Again
       (for Queue in Boolean)
          (tid : Task_ID; S : Positive) when
             (Queue = Current_Queue) and (Ready_Queue = Current_Queue) is

        Task_Priority : Task_Priority_Range := Ada.Dynamic_Priorities.Get_Priority(tid);

    begin  -- if sempahore is not available, task is requeued to other queue (not current)

      if(Queue_Again(Current_Queue)'Count = 0) then
        -- start and check other queue
        Current_Queue          := not Current_Queue;
        Ready_Queue            := Current_Queue;
      end if;

      if((Semaphore_Status(S) = False) and then (Lock_Condition(tid, Task_Priority))) then
        Semaphore_Status(S)              := True;
        Semaphore_Locking_Task_ID(S) := tid;
        Task_Orginal_Priroity(S)        := Task_Priority;
        Task_Current_Priroity(S)        := Task_Priority;
      else
        requeue Queue_Again(not Current_Queue);  -- semaphore still locked, try later
      end if;
    end Queue_Again;

    function Lock_Condition
       (tid : Task_ID;
        Task_Priority : Task_Priority_Range) return Boolean is

    begin  -- check if calling task priority is higher than ceiling of locked semaphores

      for S in 1 .. N loop
        if (Semaphore_Status(S) = True) and then
             (tid /= Semaphore_Locking_Task_ID(S)) and then
               (Semaphore_Ceiling_Priority(S) >= Task_Priority) then
             return False;
        end if;
      end loop;
      return True;

    end Lock_Condition;

  end Semaphores_Scheduler;

end PCP;
```