Priority Ceiling Protocol in Ada

Kwok-bun Yue Sadegh Davari Ted Leibfried
University of Houston - Clear Lake
Houston, TX 77058
yue@cl.uh.edu davari@cl.uh.edu leibfried@cl.uh.edu

ABSTRACT

The priority ceiling protocol (PCP) is an effective protocol for minimizing priority inversions in real-time scheduling. Priority inversion occurs when a high priority task is blocked by a low priority task, such as at a shared semaphore or a protected operation. PCP guarantees the absence of chained priority inversion or deadlock. The ceiling locking (CL) priority on protected objects of Ada-95 also has these properties but has some limitations. For example, tasks cannot suspend themselves inside a protected operation. PCP has no such restrictions. Thus, PCP is more appropriate for some real-time applications. PCP has not been implemented in any language, including Ada. A guideline for *emulating* PCP using Ada-83 exists but it lacks generality and flexibility. This paper discusses an implementation of PCP in Ada-95, the Ada-95 features that enable the implementation, the design of the implementation and some related issues.

1. Introduction

In real-time applications, preemptive priority driven schedulers are used to schedule concurrent tasks to meet deadline requirements. The rate-monotonic scheduling (RMS) is the best known preemptive *fixed* priority scheduling algorithm [7,9] for scheduling periodic tasks. RMS is optimal in the sense that if a real-time application is schedulable (all tasks meeting all deadlines) by any *fixed* priority scheduling algorithm, it is also schedulable in RMS. Since its introduction, RMS has been generalized and extended in various aspects, including aperiodic tasks with dynamic priorities [3,6,9].

RMS enables the users to perform schedulability analysis to guarantee that the application is schedulable. For each task, the period of time between its arrival and its deadline must accommodate the sum of the worst cases of the following CPU times: (1) preemption time: the time executed by all higher priority tasks, (2) execution time: the execution time of the task itself, and (3) blocking time: the delay caused by lower priority tasks because of *priority inversion* [4,9].

A common source of priority inversion is when a low priority task is executing in a critical section. A high priority task will then be blocked from executing its critical section which shares the same resource with the low priority task. Since the low priority task may be preempted by any number of medium priority tasks, the duration of priority inversion may be *unbounded*: the duration of priority inversion is not a function of the duration of the critical sections. Priority inversion cannot be eliminated entirely. However, to improve the performance of real-time systems, the duration of priority inversion should be minimized.

Many synchronization protocols have been proposed for reducing priority inversion as well as preventing deadlocks. Mutually exclusive access to shared resources from the critical sections of concurrent tasks may be protected by different synchronization techniques. Examples are semaphores, critical regions, monitors and protected objects (in Ada-95). Since most proposed protocols use semaphores to protect critical sections, we will also use semaphores in the following discussion of the protocols. However, it is straightforward to modify these protocols for synchronization techniques other than semaphores.

If a task T_i wants to execute the critical section protected by the semaphore S_j, it executes:

Request(S_j); -- request access to (lock) semaphore S_j:
-- i.e., P(S_j) or Wait(S_j)
-- CR_{ij}: the critical section of T_i that is protected by S_j.
Release(S_j); -- release access to (unlock) semaphore S_j:
-- i.e., V(S_j) or Signal(S_j)

Protocols differ in the policy of granting locks to semaphores and the policy of dynamically changing the priorities of tasks that have locked a semaphore. The priority inheritance protocol (PIP) is among the most popular protocols that avoid unbounded priority inversion [9]. In PIP, when a lower priority task T blocks the execution of higher priority tasks, T inherits the priority of the highest priority task that it blocks. PIP is supported by some operating systems. Although it avoids unbounded priority inversion, deadlock and chained priority inversion (where a task is blocked by *more than one* lower priority task) are possible [4,9,10].

The priority ceiling protocol (PCP) [4,5,9] is a more recent protocol that has the following desirable properties: (1) a high priority task can be blocked by at most one lower priority task (no chained priority inversion), even if the tasks *suspend* themselves within critical sections, and (2) no deadlock is possible. PCP is not known to be supported by any of the popular operating systems. It is based on preemptive scheduling and has the following rules:

1. A lower priority task that blocks a higher priority task T inherits the priority of T.

2. (Locking Condition) A task T can only lock a semaphore S if:

   (a) the semaphore S is not yet locked, and
   (b) the priority of T is greater than the *priority ceilings* of all semaphores that are currently locked by tasks *other* than T.

The priority ceiling of a semaphore is defined as the highest priority of all tasks that *may* request to lock the semaphore at any time.

PCP has been extended to the Optimal Mutex Policy (OMP) by Rajkumar et. al. [8]. OMP is optimal and is thus theoretically better than PCP. OMP provides both the necessary and sufficient conditions for limiting the worst-case blocking duration to a single critical section for any lower priority task. PCP provides only the sufficient condition but not the necessary condition. PCP is sub-optimal in the sense that it is more restrictive than OMP. Under some scenarios, it is possible that a request to lock a semaphore is granted by OMP, but not PCP. However, OMP contains much more complicated rules and its implementation is expensive [8]. With a much lower overhead, PCP is sufficient for most real-time applications.

2. The Ceiling_Locking Protocol of Ada

With its annexes in system programming and real-time programming, Ada provides a different solution to the problem of priority inversion. In Ada, instead of semaphores, protected objects are used to protect critical sections. The real-time annex allows the programmer to use the pragma Locking_Policy to specify the details of protected object locking. There is one predefined locking policy: Ceiling_Locking (CL) [1].

CL is similar to PCP in its use of the ceiling priority. The ceiling priority of a protected object should be an upper bound of the active priorities of all tasks that may call operations of the protected object. CL has a different set of rules:

1. A task that locks a protected object inherits the ceiling priority of the protected object during the entire protected operation.

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

CL has many desirable properties which include: (1) there is no chained priority inversion if the task does *not* suspend itself in its critical section, (2) no deadlock is possible, and (3) it is much easier to implement than PCP. For many real-time applications, CL is sufficient and is thus a suitable choice to be included in Ada.

Like PCP, CL guarantees the absence of chained priority inversion, but unlike PCP, CL relies on the Ada rule which disallows operations that may suspend tasks within protected operations. For example, the subprograms defined in the language-standard input-output package are *potentially blocking* and are thus disallowed inside protected operations [1]. PCP has no such restriction.

If suspension *were* allowed in Ada-95, then chained priority inversion may happen, as illustrated by the following example. The tasks T_1, T_2, T_3 and T_4 have priorities of 1, 2, 3 and 4 respectively, 4 being the highest. The protected object P_1 is called only by tasks T_1 and T_3 and thus has a priority of 3. The protected object P_2 is called only by tasks T_2 and T_4 and has a priority of 4. Consider the following sequence of operations.

(1) Task T_2 calls an operation of P_2 and executes with an active priority of 4.
(2) Task T_2 suspends itself for I/O. Note that Ada-95 disallows this operation.
(3) Task T_1 calls an operation of P_1 and executes with an active priority of 3.
(4) Task T_3 attempts to call an operation of P_1 and is blocked.
(5) The I/O operation of T_2 is completed and T_2 preempts T_1 because of its higher active priority.

The task T_3 must now wait for *both* T_2 and T_1 to complete their protected operations. This situation will not actually occur in Ada-95 because a task is not allowed to suspend itself within protected operations and step (2) cannot occur. Note that if PCP is used, T_1 will not be allowed to enter its critical section because locking condition (b) is not satisfied.

Another advantage of PCP (and PIP) is its closer adherence to priority scheduling. The priority of a task is only raised when it is necessary to minimize priority inversion. In CL, 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 protected operation. For example, consider the following sequence:

(1) A task T_1 locks a protected object S,
(2) A task T_2 that has a priority higher than T_1 but less than the ceiling priority of S, arrives and tries to start executing its *non*-critical section.

In CL, T_2 is blocked because T_1 has inherited the ceiling priority of S. In PCP, since T_1 has not blocked any higher priority task, its priority does not change. Thus, T_2 preempts T_1 and starts its execution.

Thus, whereas CL is suitable for many real-time applications, there are other applications where PCP is more appropriate.

3. Related Work In PCP

It is not possible to implement PCP directly in Ada-83 without implementation-dependent features and extensions. This is because Ada-83 lacks various features that are needed for the implementation of PCP, such as dynamic priority assignments, priority entry queuing, task identification, etc.

In [9], a *coding guideline* for *emulating* PCP in Ada-83 was described to overcome the FIFO nature of the entry queues and other Ada-83 deficiencies. This emulation has similar characteristics as CL of Ada-95. Under this guideline, client tasks must not call each other directly. All tasks are synchronized using critical sections protected by *server tasks*. A server task protects a shared resource and contains the code of the critical sections of all tasks that access the critical section. Each server is made up of an endless loop containing a single select statement with no guard. Server tasks are given priorities higher than that of any client task.

Using this emulation guideline, in order to execute its critical section CR_{ij} for the shared resource S_j, the task T_i calls

Server_j.CR_Entry_i(...). The server task Server_j contains the following code:

```
– body of Server_j
loop
  select
    accept ...
    ...
  or
    accept CR_Entry_i(...) do
      – code for CR_ij
      ...
    end CR_Entry_i;
  ...
  end select;
end loop;
```

Although useful, this emulation guideline in Ada-83 has some limitations. Since each shared resource is protected by a task, there is additional task overhead (unless passive implementation of tasks are supported by the Ada-83 compiler) that could be significant. More importantly, assigning higher priorities to server tasks is not acceptable if a server suspends itself during rendezvous [9] as tasks may line up in the FIFO entry queue in Ada-83. Not permitting suspension excludes useful operations such as I/O or nested critical sections. Even if a priority-based entry queue is available, nested critical sections are particularly difficult since one server must call another server directly. Deadlock is possible if suspension is allowed. This also contradicts a common design principle for making server tasks passive [2].

4. PCP in Ada

Ada-95, in its Annex C on System Programming and Annex D on Real-Time Systems, provides all features for implementing PCP efficiently [1]. Furthermore, semaphores can now be implemented as protected objects instead of tasks, thus avoiding the additional task overhead [11]. Note that if CL is sufficient for the application, then protected objects can be used directly to provide mutual exclusion.

There are several essential features for implementing PCP which are effectively supported by Ada-95. These features are not directly supported by Ada-83 (without implementation-dependent facilities) or are only partially supported.

(1) Ada-95 provides the ability to access resources according to priority. The entry queues in Ada-83 are based on FIFO, and not priority. Annex D of Ada-95 allows the programmer to define the queuing policy by the pragma Queuing_Policy. In particular, the predefined queuing policy Priority_Queuing ensures the servicing of entry queues based on priorities.

(2) Ada-95 provides the ability to identify individual tasks. Annex C defines the package Ada.Task_Identification for this purpose. It includes the definitions of the data type Task_ID and the function Current_Task that returns a value to identify the calling task. It also defines the operator = for finding out whether two task id's identify the same task. This can be simulated in Ada-83 by using task pointers or indices of task arrays.

(3) Ada-95 provides the ability to determine the current active priority of a task and the ability to set the active priority of a task dynamically. This is provided in the package Ada.Dynamic_Priorities by the functions Get_Priority and Set_Priority, as described in Annex D.

(4) In task synchronization, Ada-95 provides the ability to release a resource to wait for a certain condition to occur.

In PCP, if a task T calls a Request operation, the scheduler needs to process the request. If the locking condition is not satisfied, it is necessary for T to release the scheduler to process other requests and to wait for the condition. There is no entirely satisfactory way to do this in Ada-83 using the rendezvous model [2]. In Ada-95, the requeue statement can be used for this purpose by putting the calling task in the queue of the requeue entry, releasing the protected object to perform other protected operations [1].

An Ada-95 implementation of PCP is shown in Appendix 1. The pragma Queuing_Policy(Priority_Queuing) is first used to ensure that semaphore operations are served in the order of task's active priorities.

It may seem that the logical choice in Ada-95 is to implement one semaphore by one protected object. However, before granting a lock of a semaphore, it is necessary to check the status of every other semaphore in the system to ensure that Locking Condition (b) of PCP (discussed in Section 2) is satisfied. Implementing one semaphore as one protected object will thus require complicated communication and synchronization between the semaphore protected objects. Hence, a single protected object is instead used to serve as the scheduler for granting all semaphores.

In this implementation (Appendix 1), the protected object "Semaphores" implements the scheduler. It provides only two operations in its public interface: the entry Request and the procedure Release. Two parameters are needed in each operation: the task_id of the task that calls the protected operations and the index of the semaphore the operation is intended for. The Request operation is implemented as an entry since it contains a requeue statement.

The priority of the scheduler should be higher than that of any client task which may call a semaphore. The range of the priorities of the tasks, Task_Priority_Range, and the number of semaphores in the system, N_SEMAPHORES, are application dependent and are left unfilled in the code.

The exception EXCEEDING_CEILING_ERROR is raised when the basic assumption of PCP is violated: when a task with a priority higher than the priority ceiling of a given semaphore calls the Request operation of the semaphore. The exception UNAUTHORIZED_RELEASE_ERROR is raised when a task that has not locked a semaphore S calls a Release operation to release S. More sophisticated fool-proof checking to avoid abuses in calling the Request and Release operations are also possible.

In the private part of the scheduler, the array CEILING_PRIORITY stores the ceiling priorities of the semaphores used in the real-time system. The Boolean array Locked is used to indicate whether the semaphores are locked. When a semaphore is locked, the task_id and the priority of the task that locked the semaphore are stored in the appropriate entries of the arrays Locking_Task and Lock_Time_Priority respectively. The array Active_Priority is used to store the active priorities of the locking tasks.

When a task T calls a Request operation, the id of task T and the index of the semaphore S is passed to the entry Request of the scheduler. The barrier of the entry Request is always true. The condition for raising the exception EXCEEDING_CEILING_ERROR is checked first. After that, the locking conditions are checked. The function High_Enough_Priority implements the checking of the Locking Condition (b) using the arrays Locked, Locking_Task and CEILING_PRIORITY. If the locking conditions are satisfied, the semaphore is allowed to be locked and the task_id and priority of the calling task are saved.

If the semaphore cannot be locked, the following two steps are performed in order: (1) if the semaphore has already been locked, then the active priority of the task that locked the semaphore is updated by setting the appropriate entry of the array Active_Priority, and (2) the calling task is requeued to the *current requeue entry* for future completion of the Request operation. The current requeue entry is one of the entries Request_Again_1 and Request_Again_2 and is indicated by the variable Requeue_1_Turn. The variable is true if and only if Request_Again_1 is the current requeue entry.

The barriers of the entries Request_Again_1 and Request_Again_2 are the Boolean variables Ready_1 and Ready_2, which are initially false. They will remain false until a client task calls a Release operation which may then change the truth value of the locking conditions for the tasks waiting at the current requeue entry.

When a task T calls the Release procedure to release a semaphore S, the procedure first checks whether T has locked S. If not, UNAUTHORIZED_RELEASE_ERROR is raised. It then resets the calling task priority to its original priority at lock time and marks S as being unlocked. If some tasks are waiting in the current requeue entry, they need to be re-examined as the locking conditions may now be satisfied. In this case, the barrier of the current requeue entry is set to true to allow re-examination of the waiting tasks. That is, if the current requeue entry Request_Again_1 is true (i.e. Requeue_1_Turn is true), then Ready_1 is set to true. Otherwise, Ready_2 is set to true.

The bodies of the entries Request_Again_1 and Request_Again_2 are symmetric to each other and are similar to that of the entry Release. If the number of tasks waiting in the current requeue entry becomes zero, then its barrier is set to false. The current requeue entry is set to the other requeue entry. If the semaphore can now be locked, it is locked in a manner similar to that of the entry Release. Otherwise, the task is requeued back to the other requeue entry.

Thus, the two entries Request_Again_1 and Request_Again_2 are examined alternately like a ping-pong system. Initially, all tasks are requeued to the entry Request_Again_1. When a Release operation arrives, all tasks in Request_Again_1 and any newly arrived tasks that cannot lock a semaphore are requeued to Request_Again_2. When another Release operation arrives, all tasks in Request_Again_2 and any newly arrived tasks that cannot lock a semaphore are requeued to Request_Again_1. This process is then repeated.

Note that it is not possible to use just a single requeue entry. If only one requeue entry were used, a task that is not granted a semaphore may need to be requeued to the same entry. Since the entry is priority-based, this same task will be immediately re-examined again.

5. Conclusions

In this paper, we have presented an implementation of PCP in Ada-95. Although the implementation requires some overhead as compared to the Ceiling Locking policy of Ada-95, it allows tasks to suspend themselves in their critical sections. PCP also adheres to priority scheduling better.

Acknowledgement

We would like to thank the Institute of Space Systems Operations and the University of Houston - Clear Lake Faculty Research Support Fund for partially funding the project. We would also like to thank the anonymous reviewer whose comments have improved the paper significantly.

References

[1] Ada 95 Mapping/Revision Team, *Ada 95 Reference Manual*, Intermetics, Inc., 1995.

[2] Burns, A., *Concurrent Programming in Ada*, Cambridge University Press, 1985.

[3] Burns, A. & Wellings, A., Implementing Analysable Hard Real-Time Sporadic Tasks in Ada 9X. *Ada Letters*, Vol.14, No.1, Jan/Feb 1994, pp38-49.

[4] Davari, S. & Sha, L., Source of unbounded priority inversions in real-time systems and a comparative study of possible solutions. *ACM Operating Systems Review*, Vol.26, No.2, April 1992, pp110-120.

[5] Goodenough, J. & Sha, L., The priority ceiling protocol: a method for minimizing the blocking of high priority Ada tasks. *Ada Letters, Special Issues: Proc. 2nd Int'l Workshop on Real-Time Ada Issues VIII*, Vol.7, Fall 1988, pp20-31.

[6] Lehoczky, J. et. al., Fixed priority scheduling theory for hard real-time systems, in *Foundations of real-time computing scheduling and resource management*. ed. by van Tilborg, A. & Koob, G., Kluwer Academic Publishers, 1991.

[7] Liu, C. & Layland, J., Scheduling algorithms for multiprogramming in hard real time environments. *Journal of the ACM*, Vol.20, No.1, 1973, pp46-61.

[8] Rajkumar, R., et.al., An optimal priority inheritance policy for synchronization in real-time systems. In Son, S., (edit) *Advances in real-time systems*. Prentice-Hall, 1995.

[9] Sha, L. & Goodenough, J., Real-time scheduling theory and Ada. *IEEE Computer*, Vol.23, No.4, April 1990, pp53-62.

[10] Sha, L., Rajkumar, R. & Lehcozky, J., Priority inheritance protocols: as approach to real-time synchronization. *IEEE Transactions on Computer*, September 1990.

[11] Yue, K., Semaphores in Ada-94. *Ada Letters*, Vol.14, No.5, 1994, pp71-79.

Appendix 1 Ada's implementation of PCP

```
pragma Queuing_Policy(Priority_Queuing);
with Ada.Task_Identification; use Ada.Task_Identification;
with Ada.Dynmaic_Priorities; use Dynamic_Priorities;
with System; use System;
...
-- Range of priorities of tasks accessing critical sections.
TASK_MIN_PRIORITY          : constant Any_Priority      := ...;          -- application dependent.
TASK_MAX_PRIORITY          : constant Any_Priority      := ...;          -- application dependent.
subtype Task_Priority_Range is Any_Priority range TASK_MIN_PRIORITY..TASK_MAX_PRIORITY;

-- Priority of the scheduler scheduling semaphore operations should be higher than
-- that of any of the tasks that call the operations.
SCHEDULER_PRIORITY         : constant Any_Priority      := TASK_MAX_PRIORITY + 1;

N_SEMAPHORES               : constant                   := ...;          -- application dependent.
subtype Semaphore_Range is Integer range 1..N_SEMAPHORES;
type Semaphore_Priorities is array(Semaphore_Range) of Task_Priority_Range;
type Semaphore_Locks is array(Semaphore_Range) of Boolean;
type Semaphore_Locking_Tasks is array(Semaphore_Range) of Ada.Task_Identification.Task_Id;

-- Raised when a task with a priority higher than the ceiling priority of a semaphore calls
-- the Request operation of the semaphore.
EXCEEDING_CEILING_ERROR          : exception;

-- Raised when a task that has not locked a semaphore S calls the Release operation on S.
UNAUTHORIZED_RELEASE_ERROR    : exception;

protected Semaphores is
  -- higher priority than all tasks that access critical sections.
  pragma priority(SCHEDULER_PRIORITY);

  -- Request and Release operations of the semaphore.
  -- Release must be a procedure as it changes the barriers of Request_Again_1 and Request_Again_2.
  entry Request(Calling_Task: Ada.Task_Identification.Task_ID;
                S            : Semaphore_Range);
  procedure Release(Calling_Task: Ada.Task_Identification.Task_ID;
                    S            : Semaphore_Range);
private
  -- Indicate whether a semaphore is locked or free.
  Locked: Semaphore_Locks := (others => FALSE);

  -- Tasks that lock the semaphore.
  Locking_Task: Semaphore_Locking_Tasks := (others => NULL_TASK_ID);

  -- Ceiling priority of semaphores
  CEILING_PRIORITY: Semaphore_Priorities:= ...;              -- application dependent.

  -- Original priority of the tasks that lock the semaphores.
  Lock_Time_Priority    : Semaphore_Priorities   := (others => TASK_MIN_PRIORITY);

  -- Current active priority of the tasks that lock the semaphores.
  Active_Priority    : Semaphore_Priorities   := (others => TASK_MIN_PRIORITY);

  -- True if the current requeue entry is Request_Again_1.
  -- False if the current requeue entry is Request_Again_2.
  Requeue_1_Turn : Boolean       := True;
```

```
  -- Ready to examine the entry Request_Again_1; the barrier of Request_Again_1.
  Ready_1          : Boolean      := FALSE;
  -- Ready to examine the entry Request_Again_2; the barrier of Request_Again_2.
  Ready_2          : Boolean      := FALSE;

  -- Request_Again_1 and Request_Again_2 requeue entries of the semaphore.
  entry Request_Again_1(Calling_Task: Ada.Task_Identification.Task_ID;
                        S             : Semaphore_Range);
  entry Request_Again_2(Calling_Task: Ada.Task_Identification.Task_ID;
                        S             : Semaphore_Range);
end Semaphores;

protected body Semaphores is
  --
  -- Check whether Calling_Priority of the task Calling_Task is larger than the ceiling priorities of all locked semaphores.
  --
  function High_Enough_Priority(Calling_Task    : Ada.Task_Identification.Task_ID;
                                Calling_Priority: Task_Priority_Range) return Boolean is
  begin -- High_Enough_Priority
    for S in Semaphore_Range loop
      if Locked(S) and then Calling_Task /= Locking_Task(S)
        and then CEILING_PRIORITY(S) >= Calling_Priority then
        return FALSE;
      end if;
    end loop;
    return TRUE;
  end High_Enough_Priority;

  --
  -- The Request operation on the semaphore S called by the task Calling_Task.
  --
  entry Request(Calling_Task: Ada.Task_Identification.Task_ID;
                S             : Semaphore_Range) when TRUE is
    Calling_Task_Priority: Task_Priority_Range    := Get_Priority(Calling_Task);
  begin -- Request
    if Calling_Task_Priority > CEILING_PRIORITY(S) then
      raise EXCEEDING_CEILING_ERROR;
    elsif (not Locked(S)) and then High_Enough_Priority(Calling_Task, Calling_Task_Priority) then
      -- Semaphore available for locking.
      Locked(S) := TRUE;
      Locking_Task(S) := Calling_Task;
      Lock_Time_Priority(S) := Calling_Task_Priority;
      Active_Priority(S) := Calling_Task_Priority;
    else -- Semaphore cannot be locked.
      -- Update the active priority of the task that locks the semaphore if appropriate.
      if Locked(S) and then Active_Priority(S) < Calling_Task_Priority then
        Active_Priority(S) := Calling_Task_Priority;
        Set_Priority(Calling_Task_Priority, Locking_Task(S));
      end if;
      -- Wait in current requeue entry for future checking of the availability of the semaphore.
      if Requeue_1_Turn then
        requeue Request_Again_1;
      else
        requeue Request_Again_2;
      end if;
    end if;
  end Request;

  --
  -- The Release operation on the semaphore S called by the task Calling_Task.
  --
  procedure Release(Calling_Task: Ada.Task_Identification.Task_ID;
```

```
                    S              : Semaphore_Range) is
  begin -- Release
    if Calling_Task /= Locking_Task(S) then
      raise UNAUTHORIZED_RELEASE_ERROR;
    end if;
    Set_Priority(Lock_Time_Priority(S), Calling_Task);
    Locked(S) := FALSE;
    if Requeue_1_Turn then  -- Check Request_Again_1
      if Request_Again_1'Count > 0 then
        Ready_1 := TRUE;
      end if;
    else  -- Check Request_Again_2
      if Request_Again_2'Count > 0 then
        Ready_2 := TRUE;
      end if;
    end if;
  end Release;

  --
  -- If semaphore S is not available to the calling task Calling_Task, the entry Request will be requeue to Request_Again_2
  --
  entry Request_Again_1(Calling_Task: Ada.Task_Identification.Task_ID;
                        S             : Semaphore_Range) when Ready_1 is
    Calling_Task_Priority: Task_Priority_Range := Get_Priority(Calling_Task);
  begin -- Request_Again_1
    if Request_Again_1'Count = 0 then
      Ready_1 := FALSE;
      Requeue_1_Turn := FALSE;
    end if;
    if (not Locked(S)) and then High_Enough_Priority(Calling_Task, Calling_Task_Priority) then
      -- Semaphore available for locking.
      Locked(S) := TRUE;
      Locking_Task(S) := Calling_Task;
      Lock_Time_Priority(S) := Calling_Task_Priority;
      Active_Priority(S) := Calling_Task_Priority;
    else
      requeue Request_Again_2;
    end if;
  end Request_Again_1;

  --
  -- If semaphore S is not available to the calling task  Calling_Task, the entry Request will be requeue to Request_Again_1
  --
  entry Request_Again_2(Calling_Task: Ada.Task_Identification.Task_ID;
                S             : Semaphore_Range) when Ready_2 is
    Calling_Task_Priority: Task_Priority_Range := Get_Priority(Calling_Task);
  begin -- Request_Again_2
    if Request_Again_2'Count = 0 then
      Ready_2 := FALSE;
      Requeue_1_Turn := TRUE;
    end if;
    if (not Locked(S)) and then High_Enough_Priority(Calling_Task, Calling_Task_Priority) then
    -- Semaphore available for locking.
      Locked(S) := TRUE;
      Locking_Task(S) := Calling_Task;
      Lock_Time_Priority(S) := Calling_Task_Priority;
      Active_Priority(S) := Calling_Task_Priority;
    else
      requeue Request_Again_1;
    end if;
  end Request_Again_2;
end Semaphores;
```