Technical Report

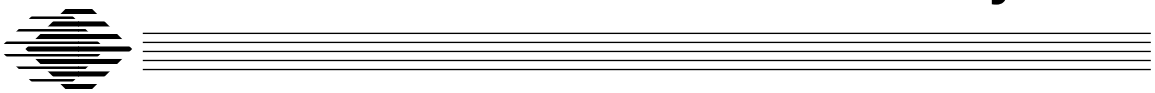
CMU/SEI-89-TR-015
ESD-TR-89-023

Implementing Priority Inheritance Algorithms in an Ada Runtime System

Mark W. Borger
Ragunathan Rajkumar
April 1989

Technical Report
CMU/SEI-89-TR-015
ESD-TR-89-023
April 1989

Implementing Priority Inheritance Algorithms in an Ada Runtime System



Mark W. Borger

Real-Time Scheduling in Ada Project

Ragunathan Rajkumar

Carnegie Mellon University

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the

SEI Joint Program Office
HQ ESC/AXS
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

(signature on file)

Thomas R. Miller, Lt Col, USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number F19628-95-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 52.227-7013.

This document is available through SAIC/ASSET: 1350 Earl L. Core Road; PO Box 3305; Morgantown, West Virginia 26505 / Phone: (304) 284-9000 / FAX: (304) 284-9001 / World Wide Web: http://www.asset.com/sei.html / e-mail: webmaster@www.asset.com

Copies of this document are available through the National Technical Information Service (NTIS). For information on ordering, please contact NTIS directly: National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161. Phone: (703) 487-4600.

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center / Attn: BRR / 8725 John J. Kingman Road / Suite 0944 / Ft. Belvoir, VA 22060-6218. Phone: (703) 767-8274 or toll-free in the U.S. — 1-800 225-3842).

Implementation Priority Inheritance Algorithms in an Ada Runtime System

Abstract: This paper presents a high-level design—in the form of necessary data structures, mechanisms, and algorithms—for implementing the basic priority inheritance and priority ceiling protocols in an Ada runtime system. Both of these protocols solve the unbounded priority inversion problem, where a high-priority task can be forced to wait for a lower priority task for an arbitrary duration of time. The protocols and their implementation also address the issues of non-deterministic selection of open alternatives and FIFO entry call queues. These protocols allow the timing analysis of a given set of Ada tasks in order to guarantee their deadlines in real-time systems. Importantly, it is possible to implement the protocols within the current semantics of the Ada language given the interpretations of Ada rules described by Goodenough and Sha in the Software Engineering Institute Technical Report 33 (1988). Strategies and possible alternatives are discussed for implementing these protocols in an Ada runtime system targeted to a uniprocessor execution environment.

1. Introduction

This paper presents a high-level design, in the form of necessary data structures, mechanisms, and algorithms, for implementing the basic priority inheritance and priority ceiling protocols [6, 10] in an Ada runtime system. These real-time scheduling protocols were developed by the Advanced Real-Time Technology Project at Carnegie Mellon University. They have been transitioned to a commercially available Ada runtime system by the Real-Time Scheduling in Ada Project at the Software Engineering Institute (SEI). This technical report summarizes implementation experiences to date, and is intended primarily for Ada runtime implementors. The paper also provides some valuable information for real-time Ada application developers, in particular those using Ada tasking in a real-time application.

1.1. Background

Despite Ada's known software engineering benefits (e.g., support of modern software engineering principles such as modularity, data abstraction, and information hiding; reduction in software integration time), only recently has the language—in particular the tasking model—been seriously considered for real-time software development. This recent scrutiny of Ada by the real-time software community has identified some inadequacies of the language with respect to expressing and handling real-time behavior [2, 4, 5, 8, 9]. These limitations relate to: non-deterministic selection of open alternatives, FIFO entry call queues, lack of an adequate time abstraction for expressing

real-time behavior, and problems with the delay statement semantics. Limitations of Ada that may preclude the use of certain real-time scheduling algorithms are discussed in [1] and [4]. Fortunately, some of these problems can be solved by using an integrated scheduling approach, which considers task priority information in all runtime scheduling decisions.

1.2. Priority Inversion

A significant problem that might arise in the use of the Ada rendezvous, or any common synchronization primitive for that matter, is called *priority inversion*. Priority inversion occurs when a high-priority task is forced to wait for the execution of a lower priority task. The most common occurrence of priority inversion in Ada arises when two or more tasks make entry calls to the same task, for example, when a (server) task is used to enforce mutual exclusion with respect to a shared resource (logical or physical). Client tasks requiring the shared resource make entry calls to the server task. This not only enforces mutual exclusion but also modularizes the operations on the shared resource. However, suppose that client task T_1 makes an entry call to a server task S that is in rendezvous with a lower priority client task T_2. Higher priority task T_1 must therefore wait until task T_2 completes its rendezvous with S. Priority inversion is unavoidable in such conditions. Nevertheless, it is essential that the duration of priority inversion be tightly bounded and be as small as possible. Only then can the timing behavior of a given set of tasks be analyzed and their deadlines guaranteed. An indiscriminate use of the Ada rendezvous can lead to unbounded priority inversion and unpredictable timing behavior, as shown by the following example.

Example

Suppose that a high-priority client task and a low-priority client task share a resource controlled by a server task. Also suppose that the server task is assigned the lowest priority in the system. First, the low-priority client rendezvous with the server task. The rendezvous is executed at the priority of the low-priority client. The high-priority client now becomes ready for execution, preempts the server task and makes an entry call to the server task. The high-priority task becomes blocked since the server task is not ready to accept the call. The server task, having the highest priority among the tasks ready to run, resumes execution. However, it is preempted by a medium-priority task that does not require the services of the server task. This medium-priority task can now run to completion before the server task can resume execution. The result is that the high-priority task has to wait for the complete execution of a medium-priority task.

The blocking duration of the high-priority task in the example above can be arbitrarily long since the server task can be preempted by any intermediate-priority task. In real-world applications these intermediate priority tasks can be periodic in nature, and therefore, can exacerbate the priority inversion problem by making the blocking duration of the high-priority task extremely long. Furthermore, assigning the highest priority to the

server task can cause some unnecessary blocking as well: high-priority tasks not requiring its services will be preempted by the server task executing on behalf of lower priority task requesting service. Finally, FIFO queuing on entry queues can contribute to additional priority inversion. If entry calls are serviced in the order they arrive, a high-priority task that arrives late will be forced to wait until all lower priority tasks before it are serviced. For instance, in the example above, if there are lower priority tasks on the server entry queue, then the (unbounded) priority inversion problem worsens. Hence, it becomes important that a priority queuing discipline be emulated.

The concept of priority inheritance solves the unbounded priority inversion problem. Priority inheritance requires that if a task *T* blocks higher priority tasks from executing, it should execute at the priority of the highest priority task it blocks. Thus, the server task in the example above will begin execution at high priority when the high-priority client makes an entry call to the server task. As a result, the server task cannot be preempted by the intermediate priority task(s), and the unbounded priority inversion problem is avoided.

1.3. Real-Time Scheduling Protocols

The *basic priority inheritance* and *priority ceiling* protocols [10] are both based on the concept of priority inheritance, which prevents the unbounded priority inversion problem. Both of these protocols were originally defined under the assumption that binary semaphores are used to synchronize access to shared resources, but they can be applied to monitors and the Ada rendezvous [6] as well. Under the basic inheritance protocol, a deadlock can occur if server tasks make entry calls to one another in cyclic order.[1] Hence, deadlock prevention measures like partial ordering of server task entry calls must be used. However, a task can still be blocked for a long duration.[2] Also, prioritized entry queues and priority-based selects must be emulated using other known techniques [3]. In contrast, under the priority ceiling protocol, deadlocks are avoided and a task is guaranteed to be blocked for at most a bounded duration. In addition, the protocol guarantees that collectively the entry queue(s) of any server task can contain at most one waiting task (assuming that server tasks do not suspend except on entry calls and *accept* statements). As a result, FIFO versus priority queuing, and arbitrary selection among open alternatives become non-issues.

Both of these real-time scheduling protocols solve the unbounded priority inversion problem where a high-priority task can be forced to wait for a lower priority task for an arbitrary duration. The protocols and their implementation also address the issues of nondeterministic selection of open alternatives and FIFO entry call queues. These protocols

[1]The problem of deadlock is not really a problem of the basic inheritance protocol, but of certain structures that can be programmed using rendezvous.

[2]The basic inheritance protocol bounds the duration of blocking, but it does not minimize it.

allow the timing analysis of a given set of Ada tasks in order to guarantee their deadlines in real-time systems. Importantly, it is possible to implement the protocols within the current semantics of the Ada language given the interpretations of Ada rules described in [7].

1.3.1. Use of Ada Features

In [6], the basic inheritance and the priority ceiling protocols were applied to the use of the Ada rendezvous as the synchronization mechanism. The assumptions underlying the protocols, when applied to Ada, require that a set of constraints be imposed on the structure of Ada tasks. Thus, the priority inheritance protocols can be applied to Ada if Ada tasks are written using a well-defined, restrictive style. That is, while the semantics of the Ada language allow tasks to be structured in multiple ways, these constraints assume only a subset of these possible structures. The essential point is that the protocols, in principle, can be applied to all task structures, but only under these constraints do the properties of the protocol lead to results that can be analyzed in a quantitative manner.

1.3.2. Coding Restrictions

The following coding restrictions with respect to the use of Ada tasking features are imposed if the priority inheritance protocols are to be applied in Ada [6] in an analyzable manner:

1. All *accept* statements in a task must be contained in a single select statement that is the only statement in the body of an endless loop. There must be no guards on the select alternatives and no nested accept statements.[3] However, entry calls to other tasks with the same structure may be made. Such a task structures model the notion of critical regions guarded by a semaphore, thus allowing application of the previously developed theory to a system of Ada tasks. A task that contains such accept statements is called a *server* task. A *client* task is a non-server task that makes at least one entry call to a server.[4]

2. There must be no conditional or timed entry calls. (These forms of call have no simple analogues in the binary semaphore version of the theory; while they may be implementable, they are excluded to simplify application of the theory to Ada.)

---

[3]Though there is no simple analogue to guards in the semaphore version of the theory, it is still possible to include guards in the task structure. In this case, an entry call would be made only if the protocol allows the call *and* the corresponding guard evaluates to TRUE. If not, the calling task τ will be suspended and the entry call will be treated as if it were not made. The callee will inherit the priority of the calling task τ (if the caller has higher priority). When the guards are reevaluated, τ is resumed and the entry call is retried as before. The process repeats and the entry call is successful both when the protocol allows the call and the corresponding guard is TRUE. For the sake of simplicity, in this paper guards are not included in the task structure.

[4]A task that contains no accept statements or entry calls is a non-server task but not a client task. Therefore, the set of non-server tasks is not the same as the set of client tasks.

3. A server task is *not* assigned any priority using *pragma* PRIORITY. Since the Ada rules do not specify any particular priority or scheduling mechanism for tasks with undefined priorities, it becomes possible to implement these priority inheritance protocols within the current semantics of the language [7]. (Although the application code should not specify a server task priority, in the runtime system a server task must be assigned a priority lower than that of any of its client tasks.[5] This restriction ensures that entry calls are executed with the correct priority. For example, in the simplest case, a rendezvous is executed with the priority of the calling task. This corresponds to executing a critical region with the priority of the process that contains it.)

1.4. Strategies for Implementing the Protocols

Chapters 2 and 3, respectively, discuss how one can implement the basic priority inheritance and the priority ceiling protocols in an Ada runtime system. An implementor can choose one of two strategies to implement these protocols in the Ada runtime system:

1. The protocols can be implemented within an existing runtime system. Surprisingly, the changes required are few and can be done rather easily. Additional options (not necessarily mutually exclusive) exist here as well:

    - The protocols can be implemented so as *not* to disrupt the behavior of Ada programs that do not meet the constraints imposed by the protocols. Hence, already written programs can still use the modified runtime system.
    - Extensive changes can be made to the existing runtime system so as to support *only* the priority inheritance protocols. Since the allowed task structures are well defined, the assumptions underlying the runtime system can be changed. As a result, some existing code can be eliminated while adding some code to implement the protocols.
    - The protocols can be implemented in an incremental fashion to facilitate easy testing and debugging. The basic priority inheritance protocol is relatively straightforward to implement, and therefore, should be implemented first. The priority ceiling protocol can then be implemented as the second step.

2. The tasking module of the Ada runtime system can be rewritten from scratch to support only the priority inheritance protocols and the restricted task structures. As explained in Section 3.1, the priority ceiling protocol, entry queues can grow no longer than 1 in length. In addition, a task that is blocked on the entry call of a server task need not be removed from the

[5] A non-server task that makes no entry calls can have a priority lower than the priority of a server task. A client task can have a priority lower than a server task if it never calls that server directly or indirectly through other server tasks.

ready queue (holding tasks that are ready to run). These properties of the protocol can be exploited to produce an efficient, streamlined implementation.

2. Basic Priority Inheritance Protocol

This chapter presents the definition of the basic priority inheritance protocol in Ada terms and discusses the implementation of the protocol in an Ada runtime system. See Appendix B for detailed examples.

2.1. Definition of the Basic Priority Inheritance Protocol for Ada

A server task *S* is said to be *executing on behalf of* client task *T* if *S* is in rendezvous either with *T* or with a server task that is in rendezvous with *T*. A server task *S* is said to be *blocking* a task *T* if *T* (or a server executing on behalf of *T*) has made an entry call to *S*, but is not in rendezvous with *S*. The basic priority inheritance protocol [10] consists of the following set of rules:

1. A server task *S* executes at its runtime assigned priority[6] when it does not block any tasks or when it is not executing on behalf of a client.
2. If a server task *S* is executing on behalf of a client task *T* or is blocking one or more tasks, the server executes at either the priority of *T* or the priority of the highest priority task (if any) that *S* blocks, whichever is higher.
3. Tasks requesting the services of a server task are serviced in priority order.

By Rule 1, a server executes at its base priority when no tasks exist that have called the server. Rule 2 means that a server task can execute at higher than its assigned priority under two conditions. The first condition occurs when the server is in rendezvous with a task *T*. Then, the server executes at the priority of *T*. If *T* itself is executing on behalf of a client, *S* executes at the priority of the client. The second condition arises when a server is blocking one or more tasks. In other words, other tasks have made entry calls to the server but have yet to rendezvous with it. In this case, the server executes at the priority of the highest priority task that it blocks. If both these conditions are satisfied, the server task executes at the higher of the two applicable priorities. Rule 3 requires that tasks be serviced in priority order. This implies that entry queues are priority-ordered and a *select* statement picks the highest priority task that is suspended on an accept statement.

Under the basic priority inheritance protocol, a client task can be blocked in two ways: (1) directly, when the called task either has a queued task or is executing a rendezvous; or (2) indirectly, when a server task inherits a higher priority. These cases of blocking are referred to as *direct* and *push-through* blocking, respectively.

---

[6] A server task's base or assigned priority means the static priority that the runtime system has given to it. Remember that coding guidelines state that pragma PRIORITY should not be used to assign a priority to server tasks.

2.2. Implementing the Basic Priority Inheritance Protocol in an Ada Runtime

The basic priority inheritance protocol can be implemented within the semantics of Ada as follows. No server task is assigned a priority using the *pragma* PRIORITY. Since Ada rules do not specify any particular priority or scheduling discipline for tasks with undefined priorities, a system-dependent pragma (or the runtime system implicitly) can assign the priority of server tasks to be lower than that of their clients. The execution priority of server tasks gets modified when client tasks make entry calls to or complete their rendezvous with the server tasks. Thus, Rules 1 and 2 (see Section 2.1) of the basic priority inheritance protocol can be implemented within the current Ada semantics. Rule 3 requires that tasks be serviced in priority order. This implies that a *select* statement must pick the highest priority task that is suspended on an accept statement and that entry queues are effectively priority-ordered. Since Ada allows the choice of the open alternative in a *select* statement to be arbitrary, an implementation that supports the basic priority inheritance protocol can specify the choice to be the alternative with the highest priority task. To affect the semantics of priority entry queues,[7] a runtime system can block server calls (i.e., not queue the call) when a server task is already executing on behalf of a client, because this rule specifies when it is "sensible" to allow the execution of the high-priority task to continue [7].[8] Thus, the basic priority inheritance protocol can be implemented within the current Ada semantics for tasking.

The basic priority inheritance protocol potentially requires priorities to be modified when tasks make entry calls. A server task may inherit the priority of a higher priority client task even though the server is not in rendezvous with the client. Thus, each (server) task has an assigned priority and an executing priority. In addition, since priority inheritance is transitive, the runtime system needs to maintain in each task control block the state information regarding any caller and callee tasks. An implementation of the basic priority inheritance protocol thus requires minor modifications to the data structures used by the runtime system and support for these data structures. The runtime data structures required and the modifications to be made to an Ada runtime system to implement the protocol are presented below. The discussion focuses on only those modules in the runtime system that are affected by the basic priority inheritance protocol. The proposed list of modifications should be treated as guidelines, as alternative implementations are also possible.

---

[7] An efficient implementation supporting the basic inheritance protocol would, ideally, implement entry queues as priority queues.

[8] A stricter, more conservative interpretation of Ada rules would require the prioritized servicing of entry calls to be coded using entry families [3]. Guards used by the solutions proposed in [3] do not violate the assumptions of the basic inheritance protocol and are permitted. However, such an implementation would be expensive both in terms of code size and runtime overhead.

2.2.1. Runtime Data Structures

The implementation of the basic priority inheritance protocol requires the additions to the runtime data structures listed below. **Note**. It is assumed that these runtime data structures already exist.

1. *Task Control Block (TCB)* - the runtime task control block associated with every Ada task must contain the following information to support the implementation of the basic priority inheritance protocol.

    a. *Base_Priority* - the task's default execution priority assigned by the runtime system or by using pragma PRIORITY.

    b. *Current_Priority* - the priority at which the task can currently run.

    c. *Called_Task* - the server task that was called by this client task, if any.

    d. *TCB_F_Link* - a pointer to the TCB of the next task after this one on the Job Queue.

    We assume that a task's TCB is initialized when the task is first created by the runtime system and remains available throughout the lifetime of the task.

2. *Prioritized Job Queue* - a prioritized, linked Job Queue of the tasks ready to execute. The task at the head of the Job Queue (i.e., the one with the highest (inherited) priority) will always be the currently executing task. Tasks of equal priority will be managed under a FIFO policy, although server tasks that inherit the priority of a task they block must be inserted and removed from the Job Queue in LIFO order to guarantee that the task at the head of the queue is always the highest priority ready-to-execute task.[9] This means that when tasks are coded using only the restricted features, a task that is blocked need not be taken off the Job Queue as would be normally done. The protocol guarantees that when a task "reaches" the head of the Job Queue, it is ready to execute.

2.2.2. Implementation Mechanisms

The implementation of the basic priority inheritance protocol in an Ada runtime system will require support for the following mechanisms:

1. *Job Queue Management* - the simple queue operations of adding in priority order and removing Task Control Blocks (TCB's) to/from the Job Queue are necessary to support the basic priority inheritance protocol.

2. *Entry Queue Management* - entry queues must be priority ordered to sup-

[9] That is, suppose that a task *T* at the head of the Job Queue blocks on a server *S* further down in the Job Queue. The server *S* inherits *T*'s priority and must be requeued. The requeuing operation can be considered as the removal of *S* from the Job Queue and insertion of *S* into the Job Queue in LIFO order. Since *S* and *T* have equal priorities, *S* is inserted at the head of the queue and becomes the next task to execute. The LIFO queuing discipline would be used if *S* was itself suspended, on an *accept* statement, when *T* makes the entry call.

port the basic priority inheritance protocol.[10] Also, when a task is added to an entry queue, its priority has to be transmitted transitively down the chain of called tasks. Furthermore, each task in this entry call chain whose priority has been elevated and that is on the Job Queue (i.e., ready to run) must be re-inserted into the Job Queue since its priority has changed.

3. *Priority Management* - the priority of the server tasks changes depending on which client tasks they are blocking. The priority of a server *S* is the maximum of the priority of a task (if any) that *S* is in rendezvous with and the priorities of the tasks (if any) in its entry queues. The server *S* executes at its assigned priority only when *S* is not in rendezvous and there are no tasks in its entry queues. Upon completion of a rendezvous, the server reverts back to the maximum of its assigned base priority and the priorities of any blocked client tasks.

2.2.3. Runtime Modifications

This section contains the details of how to implement the basic priority inheritance protocol given the above data structures and mechanisms in the runtime system. The pseudo-code provides *only* the additional support that is required to implement the basic priority inheritance protocol and assumes that the other steps required for each operation as defined by the language are already in place. We have made the following assumptions with respect to implementing the basic priority inheritance protocol in an Ada runtime system.

1. Rendezvous execute within the context (i.e., address space) of the (called) server task.
2. The Job Queue contains tasks which are ready to run. The task at the head of the Job Queue will always be the currently executing task denoted as the *Current_Task*.
3. A task switch occurs when the *Current_Task* changes, i.e., a new task has been inserted at the head of the Job Queue.

[10] A priority queuing discipline on the entry queue is a violation of Ada rules, which require entry queues to be FIFO ordered. If strict adherence to these rules is necessary, the user must be required to use prioritized entry families to ensure a priority-ordered servicing of requests [3]. Guards are required to implement this solution. The guards used for this solution do not violate the conditions of the protocol and are permitted.

2.2.3.1. Global Data

The following object must be defined and globally accessible from the Ada runtime source code.

1. *Current_Task* or *Head_Of_Job_Queue* - a pointer to the currently executing Ada task. This also represents a pointer to the head of the Job Queue.[11]

2.2.3.2. Entry Calls

The modifications to Ada's task-calling semantics necessary to implement the basic priority inheritance protocol algorithm are captured in the pseudo-code of Figure 2-1.[12]

```
----------
--  Server inherits caller's current priority.
----------
   Transmit_Callers_Priority(To => Called_Task, From => Current_Task);

if Called Server Task cannot rendezvous with Current Calling Task then
   Add Current Task to Server's Entry Queue;
end if;

Continue normal processing including a call to Check_Job_Queue;
```

Figure 2-1: Basic Priority Inheritance: Entry Calls

When a task *T* makes an entry call to a server task *S*, the execution priority of *S* changes. Since *S* would have already been executing at the highest priority of its currently calling clients, *T* must have higher priority than those clients; otherwise, *T* would not have been able to run and make the call to *S*. Hence, *T*'s priority must be inherited by *S* and its callees, if any. Server *S* and any of its callees are then requeued in the Job Queue. If any of these tasks are already in the Job Queue, they must be removed from the queue before reinsertion.

2.2.3.3. Selective Wait

The modifications to Ada's selective wait semantics necessary to implement the basic priority inheritance protocol algorithm can be summarized as follows. When a server *S* encounters a *select* statement, it picks the highest priority task that is waiting on any of its entry queues. This ensures that the highest priority waiting client is serviced first. Any required priority inheritance for *S* would have occurred when the client tasks on the entry queues made their entry calls. Hence, priority inheritance does not need to be performed here.

---

[11]The assumption is that *Current_Task* and *Head_Of_Job_Queue* point to the same task. This may not be the case for a real runtime implementation. To make our intent as clear as possible in the pseudo code, the *Head_Of_Job_Queue* variable is used as a reference point to any tasks other than the *Current_Task* currently in the Job Queue.

[12]**Note**. All bold italic subprograms in the program listings are elaborated upon further in Appendix A.

2.2.3.4. Entry Queue Insertion

The modifications to Ada's task entry queue semantics necessary to implement the basic priority inheritance protocol algorithm are captured in the pseudo-code of Figure 2-2.

```
----------
--  Prioritized insert of the Calling Task into appropriate
--  entry queue of the Called Task.
----------
  Priority_Add_To_Entry_Queue(Of => Current_Task,
                              Onto => Called_Task,
                               For => Entry_Number);

  for all tasks in the Called Task's call chain
  loop
    Transmit_Callers_Priority(From => Current_Task,
                              To => Next_Called_Task_In_Chain);
    ----------
    --  If Next Called Server Task in call chain is on the Job Queue,
    --  remove it and  re-insert it into the Job Queue with its
    --  new priority.
    ----------
      if On_The_Job_Queue(Next_Called_Task_In_Chain) then
        Remove_From_Job_Queue(Next_Called_Task_In_Chain);
        LIFO_Add_To_Job_Queue(Next_Called_Task_In_Chain);
      end if;
  end loop;

  Continue normal processing including a call to Check_Job_Queue;
```

Figure 2-2: Basic Inheritance: Entry Queue Insertion

Entry queues are priority ordered. Hence, when a task *T* makes an entry call to a server *S* and needs to be inserted into *S*'s entry queue, *T*'s position in *S*'s entry queue will depend upon its priority. Also, *S* and any of its callees inherit the caller's priority. We assume that the ***Priority_Add_To_Entry_Queue*** routine can obtain the called entry of the server from the current task's TCB.

2.2.3.5. Rendezvous Completion

The modifications to Ada's task rendezvous completion semantics necessary to implement the basic priority inheritance protocol algorithm can be summarized as follows. When a rendezvous is completed, the server task's priority is set to its base priority if the server's entry queues are empty or to the priority of the highest priority task in its entry queues. One need only look at the first task in each of the servers' entry queues to determine the highest priority caller because the entry calls are maintained in priority order.

2.3. Optimization Issues

This section discusses viable optimizations that can be performed for improving the performance of a runtime system that implements the basic priority inheritance protocol. Optimizations identified to date are summarized below.

1. To avoid unnecessary queuing and dequeuing of server tasks which have been called by multiple clients, a check can be made upon the completion of every rendezvous with a server to see if the next highest priority task eligible to run is blocked by the server.
2. If entry queues are managed in priority order, finding the highest priority client across all open alternatives for a server involves merely checking the first task in each queue.

3. Priority Ceiling Protocol

This chapter presents the definition of the priority ceiling protocol in Ada terms and discusses the implementation of the protocol in an Ada runtime system. See Appendix C for detailed examples.

3.1. Definition of the Priority Ceiling Protocol for Ada

A *server* task is one whose accept statements are all contained in a single select statement that is the only statement in the body of an endless loop. Server tasks are the only form of task allowed to contain an accept statement under the current implementation of the priority ceiling protocol. A *client* task is a non-server task that contains at least one entry call. A server task *S* is said to be *executing on behalf of* client task *T* if *S* is in rendezvous either with *T* or with a server task that is in rendezvous with *T*. The *priority ceiling* of a server task is defined as the highest priority of its client tasks, i.e., the highest priority of tasks that can call the server directly or indirectly.

The priority ceiling protocol uses the following definitions:

1. Let T be a client task attempting to call a server. The attempted call is *blocked* unless T's priority is greater than the priority ceiling of each server task that is executing on behalf of some task other than T.[13]
2. A server task S is said to *block the execution of* non-server task T if S is executing on behalf of some other client task U, T's priority is greater than U's, and either
    a. T is attempting to call a server (not necessarily S), and S has a priority ceiling greater than or equal to T's priority, or
    b. S is called by a server that blocks the execution of T, or
    c. S is blocking the execution of some task whose priority is higher than T's priority.

The priority ceiling protocol consists of the following rules [6]:

1. When an attempted entry call is blocked as defined above, the call is not made and the calling task's execution is suspended.[14]
2. A server executes at its assigned priority except when it is executing on

[13] If a server task attempts to call another server, the priority ceiling protocol guarantees that this call will never be blocked.

[14] Note that a call is blocked if the server task is executing a rendezvous or if another task is queued on an entry, since the server's priority ceiling will necessarily be greater than or equal to the caller's priority [10]. Thus the above rule includes the usual condition under which a calling task is blocked. However, the difference here is that the calling task is blocked *before* the call is actually made and placed in an entry queue.

behalf of a client task. In this case, it executes at the priority of its client unless Rule 3, below, requires execution at a higher priority.

3. If a server blocks the execution of one or more tasks, the server executes with the highest priority of the tasks it blocks[15] until the server has completed execution of an accept statement, at which point its execution priority becomes the higher of either its assigned priority or its priority as determined by these rules. Since it will usually be the case that a higher priority task is ready to run when the rendezvous is completed, the server task is usually preempted after completing a rendezvous.

Rule 1 guarantees that a server task will have at most one client on any of its entry queues by not queuing (i.e., suspending the task prior to making the entry call) any attempted entry call that is blocked. Rule 2 means a server task can execute at higher than its assigned priority even if it is not in a rendezvous.[16] Although the priority of a server task is increased by this rule, the server is *not* said to inherit its client's priority; instead, it is considered to be executing as part of its client and therefore executes at its client's priority. Under Rule 3, the server is said to *inherit* the priority of the highest priority blocked task. This rule allows the execution of a lower priority client task to delay the execution of a medium-priority task when the lower priority task has called a server and the server is blocking the execution of a high-priority task. The medium-priority task pays this price to avoid blocking the high-priority task.

Example

Suppose that a client task T requests the services of server S. Let S^* be defined as the server with the highest priority ceiling executing on behalf of some task other than T. If T's priority is not higher than S^*'s *priority ceiling*, the runtime system suspends T's execution without queuing the call and elevates S^*'s priority to that of T. In this example, when server S^* completes its current rendezvous, it resumes executing at its original priority, allowing the highest priority task blocked by the protocol to resume.

Under the priority ceiling protocol, a client task can be blocked in one of three ways: (1) directly, when the called server task either has a queued task or is executing a rendezvous; (2) indirectly, when a server task inherits a higher priority; or (3) indirectly, when, although the called task would normally be able to accept the call, some executing server task exists with a priority ceiling higher than or equal to that of the calling task. These cases of blocking are referred to as *direct*, *push-through*, and *ceiling* blocking, respectively.

[15]The priority of the blocked tasks will always be higher than the priority of the server's client task since they would not have otherwise been able to execute and become blocked.

[16]If the server task has no assigned priority, its effective priority can be increased according to priority inheritance rules [7].

3.2. Implementing the Priority Ceiling Protocol in an Ada Runtime

The priority ceiling protocol can be implemented within the semantics of Ada as follows. No server task is assigned a priority using the *pragma* PRIORITY. In addition, each server must register its priority ceiling with the runtime system, either through a pragma or a runtime service call. Since Ada rules do not specify any particular priority or scheduling discipline for tasks with undefined priorities, a system-dependent pragma (or the runtime system implicitly) can assign the priority of server tasks to be lower than that of their clients. The execution priority of server tasks gets modified when client tasks make entry calls to or complete their rendezvous with the server tasks. Furthermore, when a high-priority client wants to call a server, the call may be blocked by the runtime system in accordance with the rules listed in Section 3.1 because these rules specify when it is "sensible" to allow the execution of the high-priority task to continue [7]. Thus, the priority ceiling protocol can be implemented within the current Ada semantics for tasking.

As with the basic inheritance protocol, the priority ceiling protocol requires priorities to be potentially modified when tasks make entry calls. Since priority inheritance is transitive, the runtime system needs to maintain in each task control block (TCB) the state information regarding any caller and callee tasks. Additionally, to support the ceiling blocking rule, each task's TCB must contain information regarding any server tasks called by the task, the task's blocking status, and its priority ceiling, if applicable. This information is necessary for properly controlling the runtime behavior of client tasks whose server calls may have to be retried because of priority ceiling blocking rules.

An implementation of the priority ceiling protocol thus requires minor modifications to the data structures used by the runtime system and support for manipulating these data structures. The runtime data structures required and the modifications to be made to an Ada runtime system to implement the ceiling protocol are presented below. This discussion focuses on only those modules in the runtime system that are affected by the priority ceiling protocol. As presented, these modifications should be applied incrementally after the basic inheritance protocol has been successfully implemented. The proposed list of modifications should be treated as guidelines since alternative implementations are also possible.

3.2.1. Runtime Data Structures

The implementation of the priority ceiling protocol requires the following additions to the runtime data structures. **Note**. It is assumed that these runtime data structures already exist.

1. *Task Control Block* - the runtime task control block associated with every Ada task must contain the following information to support the implementation of the priority ceiling protocol.

    a. *Base_Priority* - the task's default execution priority assigned by the runtime system or by using pragma PRIORITY.

b. *Current_Priority* - the priority at which the task can currently run.

c. *Is_A_Server* - Boolean flag indicating whether or not the task is a server. This Boolean is used to distinguish server tasks that meet the conditions specified in Section 1.3.1. This field has to be set either by the runtime system or by the application code.

d. *Priority_Ceiling* - the highest priority of its client tasks, i.e., the highest priority of the tasks that call this server directly or indirectly.

e. *Called_Task* - the server task that was called by this client task, if any.

f. *Blocking_Task* - the server task that is presently blocking the task.

g. *TCB_F_Link* - a pointer to the TCB of the next task after this one on the Job Queue.

h. *Wakeup_Status* - information regarding the called task and specific entry that caused a ceiling blocking of this task.

It is assumed that a task's TCB is initialized when the task is first created by the runtime system and remains available throughout the lifetime of the task. It is further assumed that the values of the *Is_A_Server* and *Priority_Ceiling* fields of the TCB are provided to the runtime by the application code either via a compiler directive or a runtime service call.

2. *Prioritized Job Queue* - a prioritized, linked Job Queue of the tasks ready to execute. The task at the head of the Job Queue (i.e., the one with the highest (inherited) priority) will always be the currently executing task. Tasks of equal priority will be managed under a FIFO policy, although server tasks that inherit the priority of a task they block must be inserted and removed from the Job Queue in LIFO order to guarantee that the task at the head of the queue is always the highest priority ready-to-execute task.

The implementation of the priority ceiling protocol also requires the following runtime data structure, which is assumed not to already exist.

1. *Called Server Stack* - a prioritized LIFO stack of called server tasks ordered according to their priority ceiling. A new server can be pushed onto this stack only if its priority ceiling is higher than that of the server already at the top of the stack. Along with the server's priority ceiling, additional information must be maintained in this stack. In particular, the calling client task and the server's priority ceiling need to be saved. This information is used to enforce the priority ceiling blocking rule (see Section 3.1).

3.2.2. Implementation Mechanisms

The implementation of the priority ceiling protocol in an Ada runtime system requires support for the following mechanisms:

1. *Setting Priority Ceilings* - some mechanism for the runtime system to know the priority ceiling of each server task is needed. This could be done automatically by the compiler front end, via implementation-dependent prag-

mas, or through a runtime interface to be called by each server to register its own priority ceiling.

2. *Called Server Stack Management* - the traditional stack operations of push, pop, and top of stack will suffice for managing this data structure in the implementation of the priority ceiling protocol. Also, an operation to check a client task's priority against the priority ceiling of S^* (see Section 3.1) is needed to support the priority ceiling protocol. This operation defaults to checking the priority ceiling of the server at the top of the Called Server Stack in the absence of nested server calls. For the case of nested server calls, the ceiling blocking check need not be done since once a chain of nested server calls is initiated, the ceiling protocol guarantees that none of the server calls will be blocked. However, the server push operation may still be required in this situation.

3. *Job Queue Management* - the simple queue operations of adding (in priority order) to and removing TCBs from the Job Queue are necessary to support the priority ceiling protocol. Also, a priority insertion operation in which tasks of equal priority are treated in a LIFO manner is necessary for properly handling nested server calls under the priority ceiling protocol.

4. *Entry Queue Management* - queuing must be done on an entry only if the priority ceiling protocol allows an entry call to be made; otherwise the calling task must be *suspended* and not queued. If the attempted entry call is blocked in this manner, a runtime mechanism for retrying that call when the client becomes unblocked is necessary. The consequence of this rule is that at most one caller will be queued to a server at any given time, but the blocked caller must re-attempt the entry call when it becomes the current executing task. This action will require the saving (for the blocked caller) of state information indicating which server and specific entry should be subsequently called.

5. *Priority Management* - the priority of the server tasks changes depending on which client tasks are blocked. From Rule 3 above (Section 3.1), a server's priority can be inherited either directly or indirectly. Upon completion of a rendezvous, the server reverts to the maximum of its base priority and the priorities of any blocked client tasks.

3.2.3. Runtime Modifications

This section contains the details of how to implement the priority ceiling protocol given the above runtime data structures and mechanisms. The pseudo-code provides *only* the additional support that is required to implement the priority ceiling protocol and assumes that the other steps required for each operation as defined by the language are already in place. The following are assumptions about implementing the priority ceiling protocol in an Ada runtime system.

1. Rendezvous execute within the context of the called task.

2. The Job Queue contains tasks that are either ready to run or have been blocked by priority ceiling rules. In either case, when a task reaches the head of the Job Queue, it will be ready to execute. The task at the head of the Job Queue will always be the currently executing task denoted as the

Current_Task. Furthermore, a task switch occurs when the *Current_Task* changes, i.e., a new task has been inserted at the head of the Job Queue. This task switch normally is performed by a routine such as the ***Check_Job_Queue*** in Appendix A.c.

3. When a client *T*'s entry call to a server *S* is blocked by the priority ceiling protocol (PCP) rules, *T* is requeued on the Job Queue in LIFO order based on tasks of equal priority. This assumption guarantees that when a called server task S has inherited the priority of a client task T at the head of the Job Queue, S will be inserted ahead of T in the Job Queue.

4. The application code will indicate, either through a pragma or a call to the runtime system, the priority ceiling of each server task.

5. If a server task attempts to call another server, the priority ceiling protocol guarantees that this call will never be blocked. In this case, a conditional push (i.e., a new server is pushed onto this stack if its priority ceiling is higher than that of the server already at the top of the stack) onto Server Stack can be used so that the priority order of the Server Stack is preserved.

6. Server tasks do not execute any statements on behalf of their clients that would cause them to suspend (e.g., synchronous I/O operations). If server tasks were able to become suspended during a service call, client requests could be queued in their order of arrival (FIFO). For the sake of simplicity, this is assumed; however, it is unnecessary and can be handled as follows. When a server task *S* suspends, i.e., relinquishes the CPU while it is the highest priority task eligible to run, it along with all of the clients it has blocked must be taken off the Job Queue as a family of tasks. The list of client tasks blocked by any given server can be maintained by adding another field to each task's TCB. Should another client T_2 call *S* while it is suspended, *S* would still inherit T_2's priority and T_2 would be blocked and therefore inserted immediately after *S* in the family list. Upon resumption of *S*'s execution, each task in the family of blocked clients would be re-inserted onto the Job Queue.

3.2.3.1. Global Data

The following objects must be defined and globally accessible from the Ada runtime source code.

1. *Current_Task* or *Head_Of_Job_Queue* - a pointer to the currently executing Ada task. This also represents a pointer to the head of the Job Queue.

2. *Top_Of_Server_Stack* - a pointer to the top of the Called Server Stack.

3.2.3.2. Entry Calls

The modifications to Ada's task-calling semantics necessary to implement the ceiling protocol algorithm are captured in the pseudo-code of Figure 3-1. When a task *T* makes an entry call to a server task *S*, the execution priority of *S* changes. Since *S* would have already been executing at the highest priority of its currently calling clients, *T* must have higher priority than those clients; otherwise, *T* would not have be able to run and make

```
Current_Task.Called_Task := Called_Task;
if Called_Task.Is_A_Server then
  ----------
  --  Server inherits caller's current priority and is inserted at the
  --  head of the Job Queue.
  ----------
    Transmit_Callers_Priority(To => Called_Task, From => Current_Task);
  ----------
  --  Check for Ceiling Blocking situation
  ----------
    if Called_Task not in nested entry call then
      if Top_Of_Server_Stack = null or else
         (Current_Task.Current_Priority >
          Top_Of_Server_Stack.Priority_Ceiling)
      then
        Push_Onto_Server_Stack(Server => Called_Task,
                               Client => Current_Task);
      else
      ----------
      --  Client Call is blocked by PCP rules.
      --  Save necessary wakeup state for retrying entry call later.
      --  Mark the Current Task as "blocked by PCP".
      --  Insert the Current Task in the Job Queue
      ----------
        Save_Wakeup_Status(Current_Task, Called_Task, Called_Entry);
        Mark_As_Blocked_By_Protocol(Current_Task, Called_Task);
        LIFO_Add_To_Job_Queue(Current_Task);
        Determine_Blocking_Task(Blocking_Task);
        Transmit_Callers_Priority(To => Blocking_Task, From => Current_Task);
      end if;

    elsif Called_Task.Priority_Ceiling >
          Top_Of_Server_Stack.Priority_Ceiling
    then
        Push_Onto_Server_Stack(Server => Called_Task,
                               Client => Current_Task);
    end if;
    Check_Job_Queue;
else
  Continue normal processing;
end if;
```

Figure 3-1: Priority Ceiling: Entry Calls

the call to *S*. Hence, *T*'s priority must be inherited by *S* and its callees, if any, via the ***Transmit_Callers_Priority*** subprogram. If this entry call is nested (e.g., a server calling another server), then the priority ceiling protocol guarantees that the call can proceed. In this case, a push onto the Server Stack is necessary only if *T*'s priority ceiling is greater than that of the server at the head of the Server Stack. If this entry call is not nested, then the priority ceiling blocking check must be made. The priority of the current (i.e., calling) task *T* is compared with the priority ceiling of S^*, where S^* is always the server task at the head of the Server Stack (see Section 3.1 for the definition of S^*). If *T*'s priority is higher than S^*'s ceiling, the call can go through and the called server task *S* must be pushed onto the Server Stack. Otherwise the call is blocked[17] and the nec-

[17] In this case, the Current Task is marked as "blocked by the protocol" before the actual entry call is made. It then relinquishes the CPU to the task that caused it to block.

essary state information must be saved for a subsequent retry of the entry call to *S*. If *T* is blocked, it must be requeued on the Job Queue using an insertion operation for which tasks of equal priority are treated in a LIFO manner. Furthermore, the server task that is blocking *T*'s call must be identified so that it along with all of its callees can inherit *T*'s priority via the ***Transmit_Callers_Priority*** subprogram. Finally, for all cases, after the task priorities have been adjusted accordingly and the Job Queue manipulations are completed, the ***Check_Job_Queue*** subprogram makes a scheduling decision to determine the current executing task.[18]

3.2.3.3. Rendezvous Completion

The modifications to Ada's task rendezvous completion semantics necessary to implement the ceiling protocol algorithm are captured in the pseudo-code of Figure 3-2.

```
if Current_Task.Is_A_Server then
  ----------
  --  Are we in a nested rendezvous chain of calls from servers?
  ----------
    if Current_Task not in nested rendezvous then
      Pop_Off_Of_Server_Stack;
    else
      if Current_Task = Top_Of_Server_Stack then
        Pop_Off_Of_Server_Stack;
      end if;
    end if;

    ----------
    --  Check whether next highest task on the Job Queue is blocked
    --  on a call to Current_Task.
    ----------
      if Current_Task = Head_Of_Job_Queue.TCB_F_Link.Blocking_Task then
        Push_Onto_Server_Stack(Server => Current_Task,
                               Client => Head_Of_Job_Queue.TCB_F_Link);
        Current_Task.Current_Priority :=
            Head_Of_Job_Queue.TCB_F_Link.Current_Priority;
      else
        ----------
        --  Set the Current Task's priority to the maximum of its base
        --  priority and the current priority of its highest priority
        --  (calling) client task (in any of its entry queues).
        ----------
          Set_Priority_To_Max_Of_Base_And_Client(For => Current_Task);
      end if;

    Check_Job_Queue;
else
  Continue normal processing;
end if;
```

Figure 3-2: Priority Ceiling: Rendezvous Completion

When a simple (unnested) rendezvous is completed, the Server Stack is popped uncon-

[18]For a successful entry call, the called server task will become the *Current_Task*, whereas in the case of a blocked client entry call, the server task causing the blocking becomes the *Current_Task*.

ditionally, whereas when a rendezvous within a nested calling chain is completed, the Server Stack is popped only if *S* (i.e., *Current_Task*) is at the head of the Server Stack. In both cases, the adjusted executing priority for the server task *S* must be determined. If the task immediately after *S* in the Job Queue (call it T_b) is being blocked by *S*, T_b's call can go through now, so *S* is pushed back onto the Server Stack and *S* retains its current executing priority.[19] If this is not the case, *S*'s current priority becomes its base priority if its entry queues are empty or the priority of the highest priority task in its entry queues. Finally, after the task priorities have been adjusted accordingly, the ***Check_Job_Queue*** subprogram makes a scheduling decision to make *Current_Task* consistent with the task at the head of the Job Queue.

3.3. Optimization Issues

This section discusses viable optimizations that can be performed for improving the performance of a runtime system that implements the priority ceiling protocol. Optimizations identified to date are summarized below.

1. A task cannot be queued on a server's entry queue when either direct or ceiling blocking would occur. However, such blocking can occur only for the sequentially *first* rendezvous of a task's instance. That is, if an entry call is blocked, it has to be the *first* entry call made by this task's instance. This means that if an entry call is made and this entry call is *not* the first, it could immediately be queued without any problems. The call has to be successful.

   Moreover, this optimization does not need compiler support. The runtime can maintain runtime information of whether this is the task's first rendezvous or not. This information would be reset whenever the task blocks on external events like suspension for I/O and/or delays itself voluntarily.

   Note. The above optimization appears to hold for push-through blocking, too. However, when a rendezvous is successful, S^* has to be maintained for the benefit of other tasks, and it seems that this cannot be optimized.

2. In general, when a server task S completes a rendezvous, it is assigned a new executing priority, removed from the head of the queue, and reinserted in the Job Queue. If a server task S at the head of the Job Queue is blocking the next task in the Job Queue, an optimization can be performed to avoid unnecessary queue manipulations as follows. When a server S completes a rendezvous, consider the next task T in the Job Queue and check whether T is blocked on a call to S; if it is blocked by S, then S inherits T's priority and no queue manipulations are necessary.

[19] *S* will continue running at an inherited priority and be able to loop around to execute its selective wait statement. At this point, *S* will suspend and be requeued on the Job Queue. Consequently, T_b will become the current executing task because it is now at the head of the Job Queue. T_b will retry its call to *S* that now can succeed. T_b will make the entry call and a task switch to *S* will be made for the rendezvous code to run in *S*'s execution context.

References

1. Borger, M.W., Klein, M.H., Sha, L., and Weiderman, N. A Testbed for Investigating Real-Time Ada Issues. *Proceedings of the International Workshop of Real-Time Ada Issues*, June, 1988.

2. Burns, A., Wellings, A.J. Real-Time Ada Issues. *Proceedings of the International Workshop of Real-Time Ada Issues*, November, 1987.

3. Burns, A. "Using Large Families for Handling Priority Requests". *Ada Letters 7*, 1 (January 1987), 97-104.

4. Cornhill, D., Sha, L., Lehoczky, J.P., and Rajkumar, R., and Tokuda, H. Limitations of Ada for Real-Time Scheduling. *Proceedings of the International Workshop of Real-Time Ada Issues*, November, 1987.

5. Cornhill, D., and Sha, L. "Priority Inversion in Ada". *Ada Letters 7*, 7 (November 1987), 30-32.

6. Goodenough, J. B., and Sha, L. The Priority Ceiling Protocol: A Method for Minimizing the Blocking of High-Priority Ada Tasks. *Proceedings of the International Workshop of Real-Time Ada Issues*, June, 1988.

7. Goodenough, J. B., and Sha, L. Real-Time Scheduling Theory and Ada. Tech. Rept. SEI-88-TR-33, Software Engineering Institute, November, 1988.

8. Locke, C.D., and Vogel, D.R. Problems is Ada Runtime Task Scheduling. *Proceedings of the International Workshop of Real-Time Ada Issues*, November, 1987.

9. McCormick, F. Scheduling Difficulties of Ada in the Hard Real-Time Environment. *Proceedings of the International Workshop of Real-Time Ada Issues*, November, 1987.

10. Sha, L., Lehoczky, J.P., and Rajkumar, R. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. Tech. Rept. CMU-CS-87-181, Carnegie Mellon University, Computer Science Department, 1987.

Appendix A: Runtime Support Routines

This appendix further elaborates the details of various support routines that were used in the pseudo-code of Sections 2.2 and 3.2.

A.a. Transmit_Callers_Priority Subprogram

```
procedure Transmit_Callers_Priority (To_Task : in TCB;
                                     From_Task : in TCB ) is
 TCB_Ptr : TCB := To_Task;
begin
  while TCB_Ptr /= Null
  loop
    if From_Task.Current_Priority >= TCB_Ptr.Current_Priority then
      TCB_Ptr.Current_Priority := From_Task.Current_Priority;
      Requeue TCB_Ptr if currently on Job Queue;
      TCB_Ptr := TCB_Ptr.Called_Task;
    end if;
  end loop;
end Transmit_Callers_Priority;
```

A.b. Determine_Blocking_Task Subprogram

```
procedure Determine_Blocking_Task(Blocking_Task : in out TCB) is
begin
  if Server_Stack_Head = null then
    Blocking_Task := Current_Task.Called_Task;
  else
    Blocking_Task := Server_Stack_Head;
    ----------
    -- Find the originator of the calling server chain of
    -- the stack head. Then, find the first server in the
    -- chain which blocks the current task.
    ----------
    while Blocking_Task.Calling_Task /= null and then
          Blocking_Task.Calling_Task.Is_A_Server
    loop
       Blocking_Task := Blocking_Task.Calling_Task;
    end loop;

    while Blocking_Task.Priority_Ceiling < Current_Task.Current_Priority
    loop
      Blocking_Task := Blocking_Task.Called_Task;
    end loop;

  end if;

end Determine_Blocking_Task;
```

A.c. Check_Job_Queue

```
procedure Check_Job_Queue is
begin
  if Current_Task.Current_Priority < Head_Of_Job_Queue.Current_Priority then
     Requeue Current Task;
     Remove Job Queue Head and Make it Current Task;
     Task switch to new Current Task;
  end if;
end Check_Job_Queue;
```

Appendix B: Basic Priority Inheritance Implementation Examples

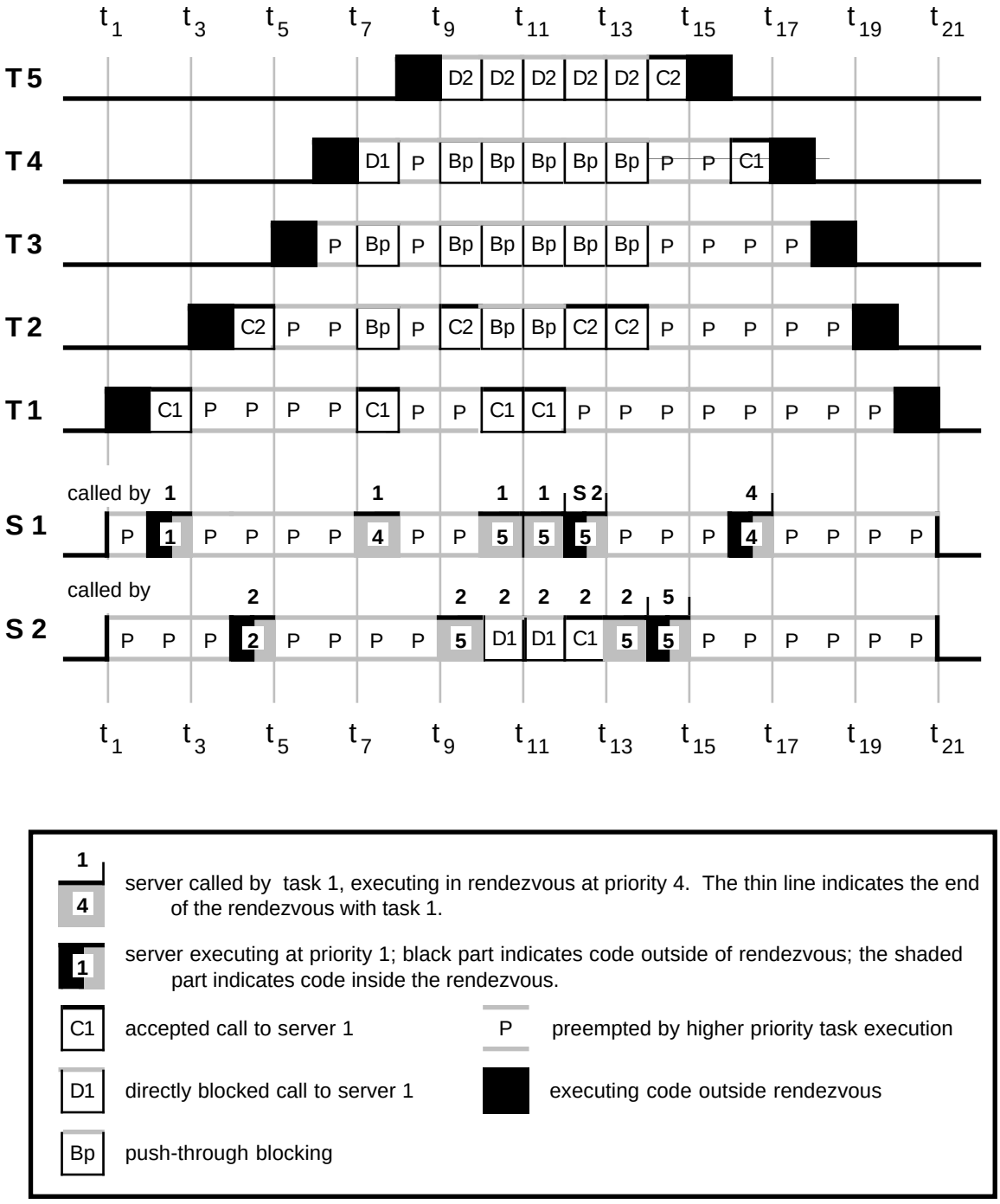
Two examples demonstrating how the proposed implementation will work are presented in this appendix. The first example appears in [6]. The second example includes more than two servers to illustrate longer blocking times. Further test cases are in preparation and will appear in a future technical report published by the Real-Time Scheduling in Ada Project.

B.a. Basic Priority Inheritance Protocol — Example #1

An example showing the effect of the basic priority inheritance protocol is given in Figure B-1. The conventions used in Figure B-1 are the same as used in [6].

Figure B-1 shows the execution of seven tasks, including two server tasks. The non-server tasks are labeled with a "T" followed by their priority, e.g., T5 has the highest priority. The server tasks have no assigned priority, however, it is assumed that effectively their priority is zero, i.e., lower than all non-servers. The calling relationships among the tasks are as follows:

- T1 becomes ready to execute at time t_1. T1 executes for 1 unit of time before and after a call to server S1. The entry call to S1 executes for 4 units of time.
- T2 becomes ready to execute at time t_3. T2 executes for 1 unit of time before and after a call to server S2. During T2's rendezvous with S2, S2 calls an entry of S1. (This illustrates the effect of the protocol for nested entry calls.) S2 executes 2 units of time before its nested call to S1 and 1 unit of time after the call. The nested call to S1 takes 1 unit of time, and thus, T2 is in rendezvous with S2 for 4 units of time.
- T3 becomes ready to execute at time t_5. T3 executes for 2 units of time and does not make any entry calls.
- T4 becomes ready to execute at time t_6. T4 executes for 1 unit of time before and after a call to server S1. The entry call takes 1 unit of time.
- T5 becomes ready to execute at time t_8. T5 executes for 1 unit of time before and after a call to server S2. The entry call takes 1 unit of time.



1 / **4** server called by task 1, executing in rendezvous at priority 4. The thin line indicates the end of the rendezvous with task 1.

1 server executing at priority 1; black part indicates code outside of rendezvous; the shaded part indicates code inside the rendezvous.

C1 accepted call to server 1

P preempted by higher priority task execution

D1 directly blocked call to server 1

executing code outside rendezvous

Bp push-through blocking

Figure B-1: Basic Priority Inheritance Protocol — Example #1

Now consider the runtime system behavior of the basic inheritance protocol under the circumstances illustrated by Figure B-1.

- At time t_0, the *Job_Queue* is empty.
- At time t_1, all tasks are activated, but only tasks T1, S1, and S2 are ready to run and therefore on the *Job_Queue*. Since T1 has the highest priority, its execution begins.
- At time t_2, T1 attempts to call server task S1. Since S1 is not executing on behalf of any task, the call succeeds; since S1 has not yet had time to execute its select statement, the call is queued; server S1 starts executing at priority 1, the priority of its queued task. Eventually its select statement is executed and the waiting call is accepted; the rendezvous starts. Execution continues at priority 1.
- At time t_3, T2 becomes ready to run and is inserted into the *Job_Queue*. Since T2 has higher priority than the current priority of tasks S1, S2, and T1, the execution of these tasks is preempted and T2 is at the head of the *Job_Queue*; therefore, T2 starts its execution.
- At time t_4, T2 attempts to call server task S2. Since S2 is not executing on behalf of any task, the call is accepted. Since S2 has not yet had time to execute its select statement, the call is queued; server S2 starts executing at priority 2, the priority of its queued task. Eventually its select statement is executed and the waiting call is accepted; the rendezvous starts. Execution continues at priority 2.
- At time t_5, T3 becomes ready to run and is inserted into the *Job_Queue*. Since its priority is higher than the current priority of any task that is ready to run, it is at the head of the *Job_Queue* and therefore preempts the execution of tasks S2, T2, S1, and T1.
- At time t_6, T4 becomes ready to run and is inserted into the *Job_Queue*. Since its priority is higher than the execution priority of any other task that is ready to run, it is at the head of the *Job_Queue* and preempts the execution of tasks T3, S2, T2, S1, and T1.
- At time t_7, T4 attempts to call server S1. Since S1 is executing on behalf of T1, T4 is directly blocked. Since S1 is blocking T4, its execution priority is increased to 4 and is re-inserted into the *Job_Queue*. S1 is now the highest priority task so it resumes execution on behalf of T1. Note that T2 and T3 have been preempted by S1 due to priority inheritance. This form of preemption by a task with a lower base priority is known as *push-through* blocking and is denoted as B_p in Figure B-1.
- At time t_8, T5 becomes ready to run and is inserted into the *Job_Queue*. Since its priority is higher than the current priority of any executing task, S1's execution is preempted, and T5 starts to execute. In this case, T4 is not blocked by a lower priority task, but rather is preempted by T5.
- At time t_9, T5 attempts to call S2. S2 is executing on behalf of T2, so T5 is directly blocked. Since S2 now blocks the execution of T5, it inherits T5's priority and is now the highest priority task ready to run. This effect causes T4 to experience push-through blocking.

- At time t_{10}, S2 attempts to call S1. Since S1 is executing on behalf of T1, this call is (directly) blocked and S1 inherits the current priority of S2. S1 is now the highest priority task ready to run; it resumes execution at priority 5.

  The situation at t_{10} illustrates how chained blocking can arise under the basic priority inheritance protocol. T5's call at t_9 cannot be accepted by S2 until both S2 and S1 complete their execution on behalf of T2 and T1, respectively. Under the priority ceiling protocol, server calls on behalf of at most one task need to be completed. However, under the basic inheritance protocol, chained blocking can occur even when nested server calls are not made.

- At time t_{11}, execution of S1 continues at priority 5.

- At time t_{12}, S1 completes its rendezvous with T1. S1 has been blocking T4 and T5. Since S2 is still blocking T5, it executes at priority 5. Its execution is resumed and its call to S1 is now accepted. Note that S2's call succeeds even though T4 called S1 first—the effect of the blocking rule is to ensure calls are accepted in order of priority rather than in order of time.

- At time t_{13}, S1 completes its rendezvous with S2. Its priority returns to 4 (that of T4), since T4 is still on S1's entry queue. S1 gets requeued and is inserted ahead of T4 in the Job Queue. Since S2 is still blocking T5, it is at the head of the *Job_Queue* so its execution resumes at priority 5.

- At time t_{14}, S2 completes its rendezvous. Its priority returns to normal and it is requeued. Since it no longer blocks T5, T5 is now the highest priority ready to run, and its call to S2 succeeds. S2 inherits T5's priority and gets requeued. Because it is the highest priority task ready to run, S2 resumes execution at priority 5.[20]

- At time t_{15}, S2 completes T5's call. S2's priority returns to normal and it is requeued. T5 continues its execution.

- At time t_{16}, T5 completes its execution. S1 is now the highest priority task ready to execute. S1 starts executing at priority 4 on behalf of T4.

- At time t_{17}, S1 completes T4's call. Its priority returns to normal, and T4 continues its execution.

- At times $t_{18\text{-}21}$, T4, T3, T2, and T1 complete their execution.

The runtime system's state over time with respect to the *Job_Queue*, the priority of the current task, and blocked calls to server tasks is presented in Figure B-2.

[20] A simple optimization can be performed here to avoid the unnecessary queuing and dequeuing operations.

Time	(Head) Job_Queue (Tail)	Priority	Directly Blocked Calls
t_0			
t_1	T1, S1, S2	1	
t_2	S1, T1, S2	1	
t_3	T2, S1, T1, S2	2	
t_4	S2, T2, S1, T1	2	
t_5	T3, S2, T2, S1, T1	3	
t_6	T4, T3, S2, T2, S1, T1	4	
t_7	S1, T4, T3, S2, T2, T1	4	T4/S1
t_8	T5, S1, T4, T3, S2, T2, T1	5	T4/S1
t_9	S2, T5, S1, T4, T3, T2, T1	5	T5/S2, T4/S1
t_{10}	S1, S2, T5, T4, T3, T2, T1	5	S2/S1, T5/S2, T4/S1
t_{11}	S1, S2, T5, T4, T3, T2, T1	5	S2/S1, T5/S2, T4/S1
t_{12}	S1, S2, T5, T4, T3, T2, T1	5	T5/S2, T4/S1
t_{13}	S2, T5, S1, T4, T3, T2, T1	5	T5/S2, T4/S1
t_{14}	S2, T5, S1, T4, T3, T2, T1	5	T4/S1
t_{15}	T5, S1, T4, T3, T2, T1, S2	5	T4/S1
t_{16}	S1, T4, T3, T2, T1, S2	4	
t_{17}	T4, T3, T2, T1, S2, S1	4	
t_{18}	T3, T2, T1, S2, S1	3	
t_{19}	T2, T1, S2, S1	2	
t_{20}	T1, S2, S1	1	
t_{21}	S2, S1	0	

Figure B-2: Runtime System State Information for Example #1

B.b. Basic Priority Inheritance Protocol — Example #2

This example consists of nine tasks, including four server tasks. It illustrates longer blocking times dues to more client/server interactions. The non-server tasks are labeled with a "T" followed by their priority. The server tasks have no assigned priority, however, it is assumed that effectively their priority is zero. The calling relationships among the tasks are as follows:

- T1 becomes ready to execute at time t_1. T1 executes for 1 unit of time before and after a call to server S1. S1 makes a nested entry call to S2. The outer entry call to S1 takes 2 units of time and the nested entry call to S2 costs 3 units of time. Hence, the net cost of the nested entry call is 5 units of time.
- T2 becomes ready to execute at time t_4. T2 executes for 1 unit of time before and after a call to server S2. The entry call takes 1 unit of time.
- T3 becomes ready to execute at time t_6. T3 executes for 1 unit of time before and after a call to server S3. S3 again calls an entry of server task S4. The outer entry call to S3 takes 3 units of time (2 units of time before

and 1 unit of time after the inner entry call), and the inner entry call to S4 consumes 3 units of time. That is, the nested entry call consumes a total of 5 units of time.

- T4 becomes ready to execute at time t_8. T4 executes for 1 unit of time before and after a call to server S4. The entry call takes 1 unit of time.
- T5 becomes ready to execute at time t_{11}. T5 executes for 1 unit of time before and after a call to server S3. S3, in turn, calls an entry of server task S4. (That is, during T5's rendezvous with S3, S3 calls an entry of server task S4). The inner entry call to S4 by S3 takes 1 unit of time while the outer entry call to S3 by T5 consumes 1 unit of time both before and after the inner entry call. In other words, the nested entry call consumes 3 units of time totally.

Now consider the actions taken by the basic inheritance protocol illustrated by Figure B-3 under the sequence of events described below.

- At time t_0, the *Job_Queue* is empty.
- At time t_1, all tasks are activated, but only tasks T1, S1, S2, S3, and S4 are ready to run and therefore on the *Job_Queue*. Since T1 has the highest priority, its execution begins.
- At time t_2, T1 attempts to call server S1. Since S1 is not executing on behalf of any task, the call succeeds; since S1 has not yet had time to execute its select statement, the call is queued; server S1 starts executing at priority 1, the priority of its queued task. Eventually its select statement is executed and the waiting call is accepted; the rendezvous starts. Execution continues at priority 1.
- At time t_3, S1 attempts to call S2. Because S2 is not executing on behalf of any task, the call succeeds; since S2 has not yet had time to execute its select statement, the call is queued; server S2 starts executing at priority 1, the priority of its queued task. Eventually S2's select statement is executed and the waiting call is accepted; the rendezvous starts. Execution continues at priority 1.
- At time t_4, T2 becomes ready to run and is inserted into the *Job_Queue*. Since T2 has higher priority than the current priority of tasks S2, S1, and T1, the execution of these tasks is preempted, and T2 is at the head of the *Job_Queue*; therefore, T2 starts its execution.
- At time t_5, T2 attempts to call server S2. Since S2 is executing on behalf of T1 through S1, T2 is directly blocked. Because S2 is blocking T2, its execution priority is increased to 2 and is re-inserted at the head of the *Job_Queue*. S2 is now the highest priority task so it resumes execution on behalf of T1.
- At time t_6, T3 becomes ready to run and is inserted into the *Job_Queue*. Since its priority is higher than the current priority of any task that is ready to run, it is at the head of the *Job_Queue* and therefore preempts the execution of tasks S2, T2, S1, and T1.

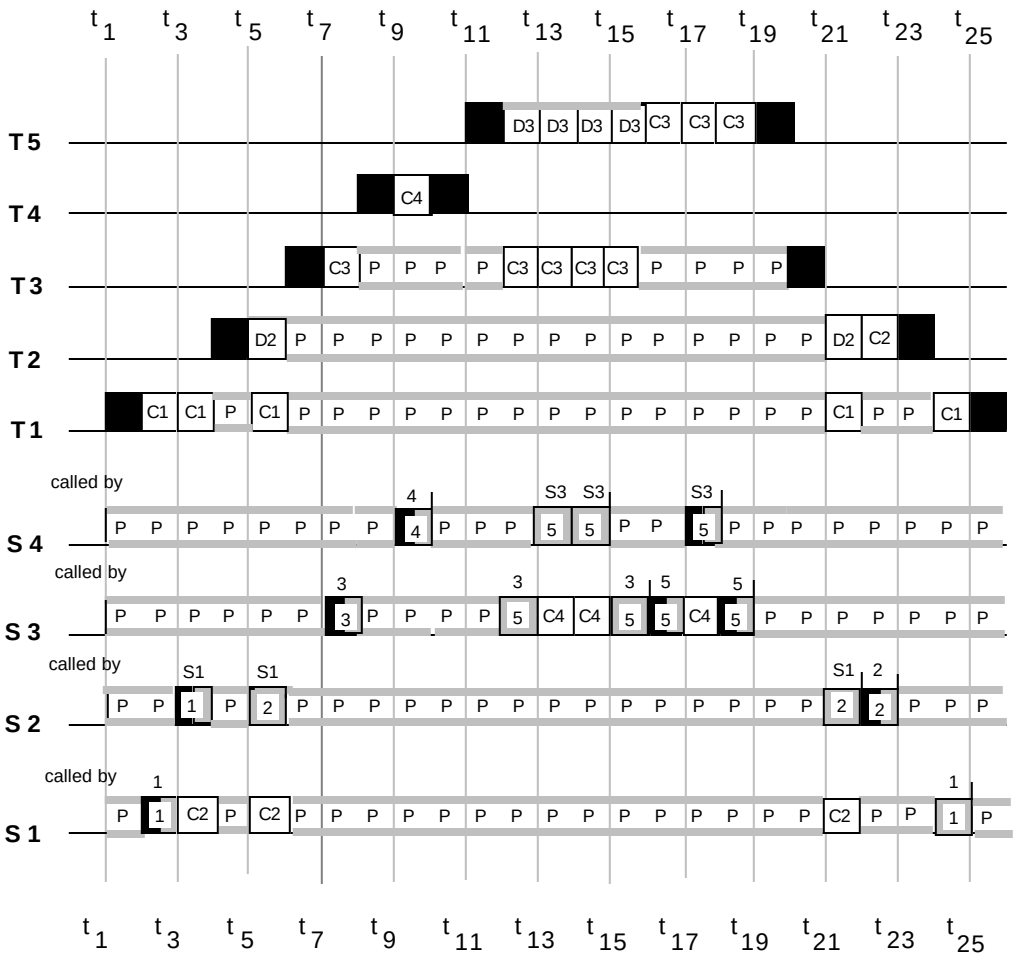


Figure B-3: Basic Priority Inheritance Protocol — Example #2

- At time t_7, T3 attempts to call server task S3. Because S3 is not executing on behalf of any task, the call succeeds; since S3 has not yet had time to execute its select statement, the call is queued; server S3 starts executing at priority 3, the priority of its queued task. Eventually S3's select statement is executed and the waiting call is accepted; the rendezvous starts. Execution continues at priority 3.
- At time t_8, T4 becomes ready to run and is inserted into the *Job_Queue*. Since its priority is higher than the current priority of any executing task, S3's execution is preempted, and T4 starts to execute.
- At time t_9, T4 attempts to call server task S4. Because S4 is not executing on behalf of any task, the call succeeds; since S4 has not yet had time to execute its select statement, the call is queued; server S4 starts executing at priority 4, the priority of its queued task. Eventually S4's select statement is executed and the waiting call is accepted; the rendezvous starts. Execution continues at priority 4.
- At time t_{10}, T4 completes its rendezvous with S4. S4 returns to its normal priority and is requeued. T4 continues executing.
- At time t_{11}, T4 has completed its execution and T5 becomes ready to run and is inserted into the *Job_Queue*. Since its priority is higher than the current priority of any executing task, T5 starts to execute.
- At time t_{12}, T5 attempts to call S3. S3 is executing on behalf of T3, so T5 is directly blocked. Since S3 now blocks the execution of T5, it inherits T5's priority and is now the highest priority task ready to run.
- At time t_{13}, server S3 makes a call to S4. Since this is a nested entry call, S4 now begins execution at the inherited priority of T5.
- At time t_{14}, server S4 continues to execute on behalf of S3.
- At time t_{15}, S4 completes its rendezvous with S3 and resumes its original priority. S3 is at the head of the *Job_Queue* and resumes execution on behalf of T3 at the inherited priority of task T5.
- At time t_{16}, S3 completes its rendezvous with T3. Its priority returns to normal and it is requeued. Since it no longer blocks T5, T5 is now the highest priority ready to run, and its call to S3 succeeds. S3 inherits T5's priority and gets requeued.[21] Because it the highest priority task ready to run S3 resumes execution at priority 5.
- At time t_{17}, server S3 makes a call to S4. Since this is a nested entry call, S4 now begins execution at the inherited priority of T5.
- At time t_{18}, S4 completes its rendezvous with S3 and resumes its original priority. S3 is at the head of the *Job_Queue* and resumes execution on behalf of T5 at its inherited priority.

[21] In this case the requeuing operations of S3 can be optimized away.

- At time t_{19}, S3 completes its rendezvous with T5 and regains its original priority. Task T5, which is at the head of the *Job_Queue*, resumes execution at its original priority of 5.
- At time t_{20}, task T5 completes execution and is removed from *Job_Queue*. T3 is at the head of the *Job_Queue* and resumes its execution.
- At time t_{21}, task T3 completes execution and is removed from *Job_Queue*. S2 is at the head of the *Job_Queue* so its continues executing on behalf of S1.
- At time t_{22}, S2 completes its rendezvous with S1, but T2 is ready to call S2. S2 remains at the head *Job_Queue* and begins executing on behalf of T2.
- At time t_{23}, S2 completes its rendezvous with T2. S2's priority returns to normal and it is requeued. T2 continues its execution.
- At time t_{24}, T2 completes its execution. S1 is now at the head of the *Job_Queue* and resume its execution on behalf of T1.
- At time t_{25}, S1 completes its rendezvous with T1. S1's priority returns to normal and it is requeued. T1 continues its execution.
- At time t_{26}, T1 completes its execution. S1 is now at the head of the *Job_Queue*.

The runtime system's state over time with respect to the *Job_Queue*, the priority of the current task, and blocked calls to server tasks is presented in Figure B-4.

Time	(Head) Job_Queue (Tail)	Priority	Directly Blocked Calls
t_0			
t_1	T1, S1, S2, S3, S4	1	
t_2	S1, T1, S2, S3, S4	1	
t_3	S2, S1, T1, S3, S4	1	
t_4	T2, S2, S1, T1, S3, S4	2	
t_5	S2, T2, S1, T1, S3, S4	2	T2/S2
t_6	T3, S2, T2, S1, T1, S3, S4	3	T2/S2
t_7	S3, T3, S2, T2, S1, T1, S4	3	T2/S2
t_8	T4, S3, T3, S2, T2, S1, T1, S4	4	T2/S2
t_9	S4, T4, S3, T3, S2, T2, S1, T1	4	T2/S2
t_{10}	T4, S3, T3, S2, T2, S1, T1, S4	4	T2/S2
t_{11}	T5, S3, T3, S2, T2, S1, T1, S4	5	T2/S2
t_{12}	S3, T5, T3, S2, T2, S1, T1, S4	5	T5/S3, T2/S2
t_{13}	S4, S3, T5, T3, S2, T2, S1, T1	5	T5/S3, T2/S2
t_{14}	S4, S3, T5, T3, S2, T2, S1, T1	5	T5/S3, T2/S2
t_{15}	S3, T5, T3, S2, T2, S1, T1, S4	5	T5/S3, T2/S2
t_{16}	S3, T5, T3, S2, T2, S1, T1, S4	5	T2/S2
t_{17}	S4, S3, T5, T3, S2, T2, S1, T1	5	T2/S2
t_{18}	S3, T5, T3, S2, T2, S1, T1, S4	5	T2/S2
t_{19}	T5, T3, S2, T2, S1, T1, S4, S3	5	T2/S2
t_{20}	T3, S2, T2, S1, T1, S4, S3	3	T2/S2
t_{21}	S2, T2, S1, T1, S4, S3	2	
t_{22}	S2, T2, S1, T1, S4, S3	2	
t_{23}	T2, S1, T1, S4, S3, S2	2	
t_{24}	S1, T1, S4, S3, S2	1	
t_{25}	T1, S4, S3, S2, S1	1	
t_{25}	S4, S3, S2, S1	0	

Figure B-4: Runtime System State Information for Example #2

Appendix C: Priority Ceiling Protocol Implementation Examples

Two examples demonstrating how the proposed implementation will work are presented in this appendix. These examples are exactly those task sets described in Appendix B. Note that for the priority ceiling protocol, since the second example includes more than two servers, it will be useful in demonstrating the behavior of the Called Server Stack and identifying potential optimizations when a server completes a rendezvous and its priority must be adjusted downwards.

C.a. Priority Ceiling Protocol — Example #1

An example showing the effect of the priority ceiling protocol is given in Figure C-1. Since S1 is called by T1, T2, and T4, S1's priority ceiling is 4. Since S2 is called by tasks T2 and T5, its priority ceiling is 5.

Note. The data structure *Server_Stack* is a LIFO stack, and the *insertion* and *deletion* operations on it are *push* and *pop* operations, respectively. Though the members of *Server_Stack* are listed as pairs, say S2/T5, one need only store a pointer to the task control block of S and have one of its fields to point to the TCB of T5.

- At time t_0, initially the *Job_Queue* and Server Stack are empty.
- At time t_1, all tasks are activated, but only tasks T1, S1, and S2 are ready to run and therefore on the *Job_Queue*. Since T1 has the highest priority, its execution begins.
- At time t_2, T1 attempts to call server S1. Since no server tasks are executing on behalf of any task (i.e., the Server Stack is empty), the call succeeds. Since the server has not yet had time to execute its select statement, the call is queued, and server S1 starts executing at priority 1, the priority of its queued task. The server/client task pair S1/T1 is inserted into the Server Stack.
- At time t_3, T2 is inserted into the *Job_Queue*. Since T2 has higher priority than the current priority of tasks S1, S2, and T1, the execution of these tasks is preempted and T2 is at the head of the *Job_Queue*; therefore, T2 starts its execution. Since server S1 is preempted by T2, S^* is updated to S1, which is at the head of *Server_Stack*.
- At time t_4, T2 attempts to call server task S2. We now examine the priority ceiling of S^* which is 4. Since this priority ceiling exceeds T2's current priority, T2 is blocked, i.e., its execution is suspended and the entry call is not made. In particular, T2 is *not* queued for S2. Since S1 blocks the execution of T2, S1 inherits task T2's priority, i.e., is taken off the *Job_Queue* and re-inserted given its new inherited priority. S1 is now the highest priority task so it resumes execution on behalf of T1. Hence, S^* is reset to *NULL* since there is no server currently executing on behalf of a task other than task T1.

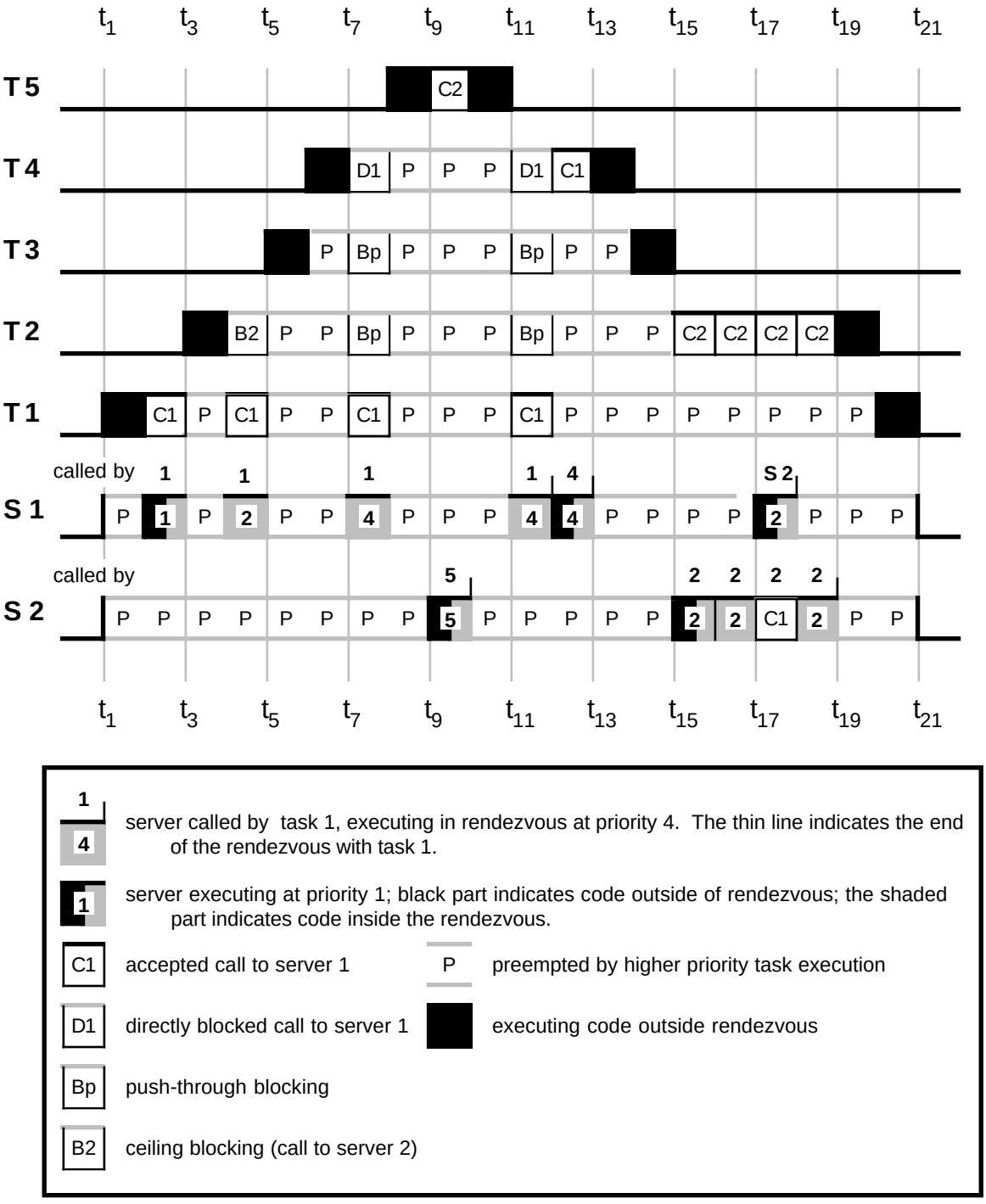


Figure C-1: Priority Ceiling Protocol — Example #1

- At time t_5, T3 is inserted into the *Job_Queue*. Since its priority is higher than the current priority of any task that is ready to run, it is at the head of the *Job_Queue* and therefore preempts the execution of tasks S2, S1, T1, and T2. Since server S1 is preempted by T3, S^* is updated to S1, which is at the head of *Server_Stack*.

- At time t_6, T4 is inserted into the *Job_Queue*. Since its priority is higher than the execution priority of any other task that is ready to run, it is at the head of the *Job_Queue* and preempts the execution of tasks S2, S1, T1, T2, and T3. Both *Server_Stack* and S^* remain the same.

- At time t_7, T4 attempts to call server task S1. The priority ceiling of S^* is 4, which is equal to the priority of the calling task, so T4's execution is blocked.[22] Since S1 is blocking T4, its execution priority is increased to 4 and is re-inserted into the *Job_Queue*. S1 is now the highest priority task so it resumes execution on behalf of T1. Hence, S^* is reset to *NULL* since there is no server currently executing on behalf of a task other than task T1.

- At time t_8, T5 is inserted into the *Job_Queue*. Since its priority is higher than the current priority of any executing task, S1's execution is preempted, and T5 starts to execute. Since server S1 is preempted by T5, S^* is updated to S1, which is at the head of *Server_Stack*.

- At time t_9, T5 attempts to call S2. The priority ceiling of S^* is 4, which is less than T5's priority, so the call can be accepted. The server/client task pair S2/T5 is inserted into the Server Stack. Since S2 has not yet had an opportunity to execute its select statement, T5 is queued. S2 is given the priority of its queued task and re-inserted into the *Job_Queue*. S2 is now the highest priority task able to execute.

- At time t_{10}, the rendezvous with S2 is completed. The server/client task pair S2/T5 is removed from the Server Stack. S2 reverts to its assigned priority (i.e., it is requeued on the *Job_Queue*), and so its execution is preempted by the execution of T5.

- At time t_{11}, T5 completes its execution and is popped off the *Job_Queue*. T4's call to S1 is still blocked since S1 has not yet finished its rendezvous. S1 has priority 4 currently at the head of the *Job_Queue* and therefore continues its execution.

- At time t_{12a}, task S1 completes its rendezvous. Its priority returns to normal and is re-inserted in to the *Job_Queue*; furthermore, the task server/client pair S1/T1 is removed from the Server Stack. S1 has been blocking the execution of tasks T2, T3, and T4. Completion of the rendezvous means tasks T2 and T4 are no longer blocked because S1 is no longer executing on behalf of any task. In addition, its low priority means it no longer blocks T3. So all tasks are eligible to run. Since task T4 has the highest priority, its execution resumes.

- At time t_{12b}, since T4 was suspended just before making the call to S1, now

[22] **Note**. In this case T4's call to S1 is also directly blocked since S1 has already been called by T1.

that its execution has resumed, it again attempts to call S1. Since the Server Stack is empty, the call succeeds and the S1/T4 task pair is inserted into the Server Stack. Since S1 just completed its rendezvous, it is not yet ready to accept the call (it is not yet waiting at an accept statement), so the call is queued. Since the call is queued, S1's current priority is raised to the current priority of the calling task and it gets inserted at the head of the *Job_Queue*. S1 now has the highest execution priority. It begins execution and eventually executes its select statement. The waiting call is accepted and the rendezvous begins. Execution continues at priority 4. S^* remains *NULL* since no other other servers are executing on behalf of other clients.

- At time t_{13}, task S1 completes its rendezvous. Its priority returns to normal and is re-inserted in to the *Job_Queue*; furthermore, the task server/client pair S1/T4 is removed from the Server Stack. All tasks are now eligible to run. Since T4 has completed its call to S1 and has the highest priority, its execution resumes.

- At time t_{14}, T4 completes its execution and is popped off the *Job_Queue*. T3 now is at the head of the *Job_Queue* since it has the highest priority and therefore resumes its execution.

- At time t_{15}, T3 completes its execution and is popped off the *Job_Queue*. T2 now has the highest priority and resumes its execution by attempting to re-call server S2. **Note**. T2 has been blocked on its call to S2 since t_4. Since the Server Stack is empty, the call succeeds and the S2/T2 task pair is inserted into the Server Stack; S^* remains *NULL*. Since S2 has just completed a rendezvous, it is not yet ready to accept the call, so the call is queued. Since the call is queued, S2's execution priority is raised to the execution priority of the calling task and it gets inserted at the head of the *Job_Queue*. S2 now has the highest execution priority. It begins execution and eventually executes its select statement. The waiting call is accepted and the rendezvous begins. Execution continues at priority 2, the current priority of the calling task.

- At time t_{16}, execution of the rendezvous continues.

- At time t_{17}, S2 attempts to call S1. Since this is a nested entry call, the call succeeds and S2 is queued on S1, which inherits S2's inherited priority of 2. Since the priority ceiling of S1 is lower than the priority ceiling of the head of *Server_Stack*, S2, the pair S1/T2 is *not* added to *Server_Stack*.

- At time t_{18}, S1 completes execution of the rendezvous. S1's priority returns to normal and it is re-inserted in to the *Job_Queue*. S2 is at the head of the *Job_Queue* and it continues executing its rendezvous at the priority of its calling task, T2.

- At time t_{19}, S2 completes execution of its rendezvous. Its priority returns to normal and it is re-inserted in to the *Job_Queue*; furthermore, the task server/client pair S2/T2 is removed from the Server Stack. T2 is now the highest priority task so its execution resumes.

- At time t_{20}, T2 completes its execution and is popped off the *Job_Queue*. T1 is at the head of the *Job_Queue* and can now resume its execution.

- At time t_{21}, T1 completes its execution and is popped off the *Job_Queue*. S1 is now at the head of the *Job_Queue*.

Now consider the actions taken by the ceiling protocol illustrated by Figure C-1 under the sequence of events described below. The system actions on the runtime data structures *Job_Queue*, *Server_Stack* and S^* are presented in Figure C-2.

Time	(Head) Job_Queue (Tail)	Priority	Server_Stack	S^*/Ceiling
t_0				
t_1	T1, S1, S2	1		
t_2	S1, T1, S2	1	S1/T1	
t_3	T2, S1, T1, S2	2	S1/T1	S1/4
t_4	S1, T2, T1, S2	2	S1/T1	
t_5	T3, S1, T2, T1, S2	3	S1/T1	S1/4
t_6	T4, T3, S1, T2, T1, S2	4	S1/T1	S1/4
t_7	S1, T4, T3, T2, T1, S2	4	S1/T1	
t_8	T5, S1, T4, T3, T2, T1, S2	5	S1/T1	S1/4
t_9	S2, T5, S1, T4, T3, T2, T1	5	S2/T5, S1/T1	
t_{10}	T5, S1, T4, T3, T2, T1, S2	5	S1/T1	
t_{11}	S1, T4, T3, T2, T1, S2	4	S1/T1	
t_{12a}	T4, T3, T2, T1, S2, S1	4		
t_{12b}	S1, T4, T3, T2, T1, S2	4	S1/T4	
t_{13}	T4, T3, T2, T1, S2, S1	4		
t_{14}	T3, T2, T1, S2, S1	3		
t_{15}	S2, T2, T1, S1	2	S2/T2	
t_{16}	S2, T2, T1, S1	2	S2/T2	
t_{17}	S1, S2, T2, T1	2	S2/T2	
t_{18}	S2, T2, T1, S1	2	S2/T2	
t_{19}	T2, T1, S1, S2	2		
t_{20}	T1, S1, S2	1		
t_{21}	S1, S2	0		

Figure C-2: Runtime System State Information for Example #1

C.b. Priority Ceiling Protocol — Example #2

Reconsider the task set defined in Section B.b. Since S1 is called only by T1 (priority 1), S1's priority ceiling is 1. Since S2 is called by T2 (priority 2), S2's priority ceiling is 2. Likewise, the priority ceiling of S3 is 5, since S3 is called by T5 (priority 5). S4 is also called on behalf of T5 (through S3) and hence S4's priority ceiling is also 5.

Now consider the actions taken by the ceiling protocol illustrated by Figure C-3 under the sequence of events described below.

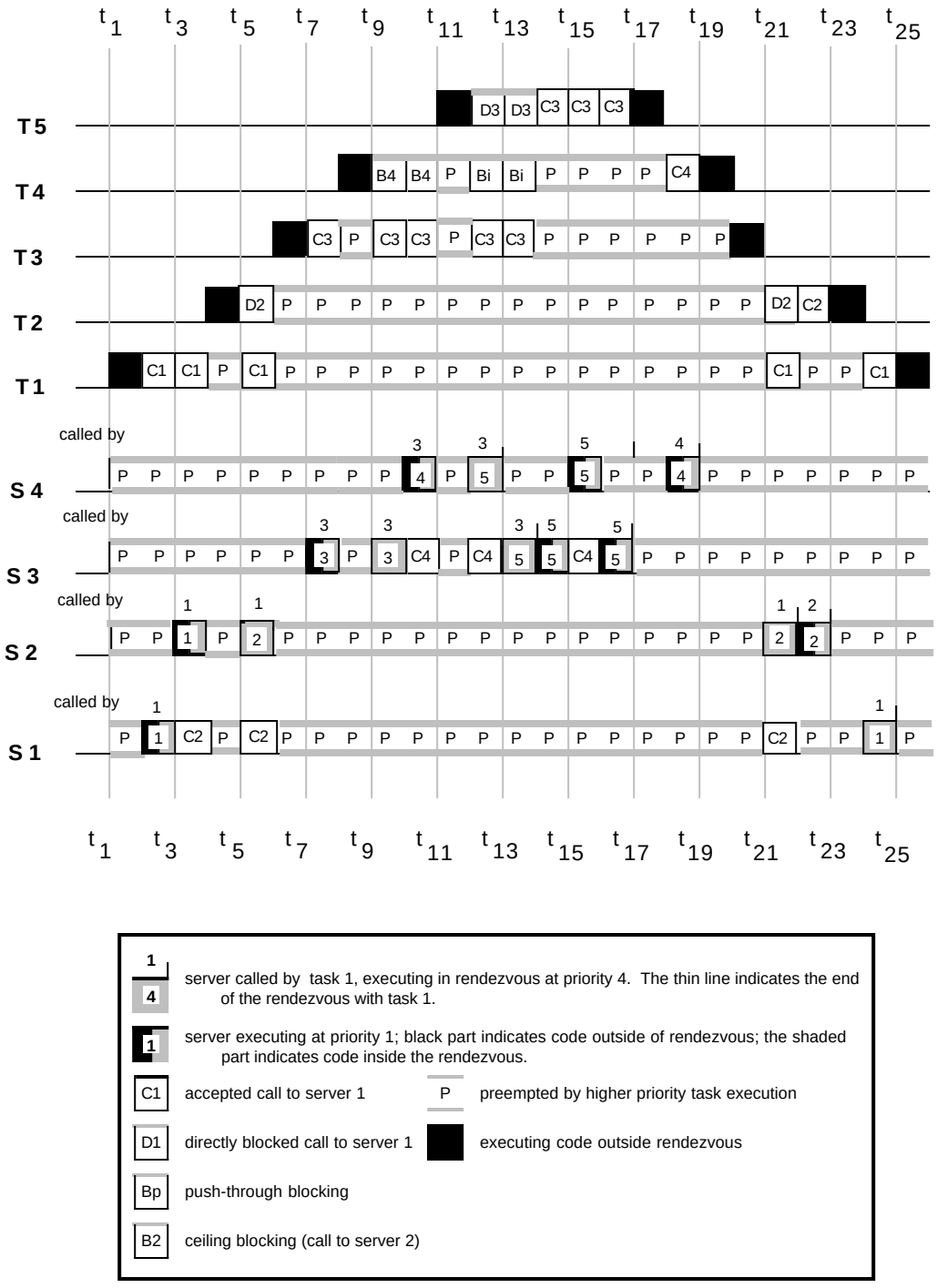


Figure C-3: Priority Ceiling Protocol — Example #2

- At time t_0, *Job_Queue* and *Server_Stack* are empty.
- At time t_1, all tasks are activated, but only tasks T1, S1, S2, S3, and S4 are ready to run and hence are on the *Job_Queue*.
- At time t_2, T1 attempts to call S1. Since *Server_Stack* is empty, no other servers are executing on behalf of any other task. Hence, the call succeeds and is queued. The server S1, which has not yet executed its *accept* statement, begins its execution at priority 1, the priority of its caller T1. The client/server task pair S1/T1 is inserted into *Server_Stack*.
- At time t_3, S1 attempts to call S2. Since nested server calls can always succeed, the call is queued. The server S2 now begins execution at the executing priority of its caller S1, namely at the priority of T1. Insertion into *Server_Stack* for nested calls is carried out under special conditions only. The server/caller pair of S2/S1 is added to *Server_Stack* since S2's priority ceiling is higher than the priority ceiling of its caller S1.
- At time t_4, task T2 becomes ready to run and immediately preempts S2. Thus, T2 is at the head of the *Job_Queue*. Since server S2 is preempted by T2, S^* is updated to S2, which is at the head of *Server_Stack*.
- At time t_5, task T2 attempts to call S_2. Since T2's priority is not higher than the priority ceiling of S^* (S2), the call fails and S2 inherits the priority of T2.
- At time t_6, task T3 becomes ready to execute, and having higher priority than the inherited priority of S2, it preempts S2 and begins to execute. S^* needs to be updated but remains S2, which is still the head of *Server_Stack*.
- At time t_7, task T3 tries to call S3. Since T3's priority is higher than the priority ceiling of S^* (S2), the call is successful and is queued on S3. The server/client pair of S3/T3 is inserted into *Server_Stack*. S3 now begins execution at its inherited priority of T3's (3).
- At time t_8, task T4 with priority 4 becomes eligible to run and preempts S3 running at a priority of 3. Since a server has been preempted, S^* is updated to S3 which is now at the head of *Server_Stack*.
- At time t_9, task T4 attempts to call S4. Since T4's priority of 4 is less than the priority ceiling 5 of S^* (S3), the call is not successful and S3 inherits T4's priority. Since S3 is executing on behalf of T3, S^* is set to the server with the highest priority ceiling called by a task other than T3. Thus, S^* is reset to S2.
- At time t_{10}, server S3 makes a call to S4. Since this is a nested entry call, the call is successful and the call is queued on S4. S4 now begins execution at the inherited priority of 4. Since this is a nested entry call, S4 needs to be added to *Server_Stack* only if S4's priority ceiling is higher than the priority

ceiling of S3, which is at the head of the stack.[23] Since this is *not* the case, the pair S4/S3 is *not* added to *Server_Stack*.

- At time t_{11}, task T5 with priority 5 becomes eligible to run and preempts server S4 running at priority 4. Since the server S4 was preempted by task T5, S^* is updated to the head of *Server_Stack*, namely S3.
- At time t_{12a}, task T5 attempts to call S3. However, T5's priority of 5 is not higher than the priority ceiling 5 of S^* (S3) and the call is unsuccessful. Instead, S3 inherits T5's priority of 5. Since S3 is executing on behalf of T3, S^* is reset to S2.[24] However, S3 is itself blocked waiting for S4 to complete its rendezvous. Hence, at time t_{12b}, S4 inherits S3's executing priority, namely that of T5.
- At time t_{13}, S4 completes its rendezvous with S3 and resumes execution at its base priority. S3 is at the head of *Job_Queue* and resumes execution at its inherited priority of task T5. Note that S^* still remains S2, since S3 is executing on behalf of T3.
- At time t_{14a}, S3 completes its rendezvous and resumes execution at its base priority. Hence, the server/client pair of S3/T3 is removed from *Server_Stack*. T5, the task at the head of *Job_Queue*, needs to rendezvous with S3, S3 reinherits T5's priority at time t_{14b} and regains its position at the head of *Job_Queue*. The server/client pair S3/T5 is now inserted into *Server_Stack*. Since S3 is being called by the currently executing job T5, S^* still remains S2.
- At time t_{15}, S3 makes an entry call to S4. Since this is a nested entry call, the call succeeds and S3 is queued on S4, which inherits S3's inherited priority of task T5. Since the priority ceiling of S4 is lower than the priority ceiling of the head of *Server_Stack*, S3, the pair S4/T5 is *not* added to *Server_Stack*.
- At time t_{16}, S4 completes its rendezvous with S3 and resumes its original priority. S3 is now at the head of *Job_Queue* and resumes its execution at its inherited priority of task T5. Both *Server_Stack* and S^* remain the same.
- At time t_{17}, S3 completes its rendezvous with T5 and regains its original base priority. Hence the server/client pair of S3/T5 is deleted from the *Server_Stack*. Task T5, which is at the head of *Job_Queue*, resumes execution at its original priority of 5.
- At time t_{18a}, task T5 completes execution and is removed from *Job_Queue*. Task T4 is at the head of *Job_Queue* but needs to call S4. Hence, at time t_{18b}, server task S4 inherits T4's priority, reaches the head of *Job_Queue*

[23] Nested entry calls only need guarantee that the largest priority ceiling associated with any of the server tasks in the calling chain up to this point is represented on the Server Stack. Since the Server Stack is managed as a prioritized stack, there is no need to push the S4/S3 pair in this situation.

[24] The repetitive setting and resetting of S^*, for instance at t_9 and t_{12}, can be "optimized away" by allowing nested entry calls to go through freely.

and begins execution. The server/client pair of S4/T4 is inserted into *Server_Stack* but S^* remains as S2.

- At time t_{19}, server task S4 completes its rendezvous with T4 and resumes its original priority. The pair S4/T4 is deleted from *Server_Stack*. Task T4 resumes execution at its own priority.
- At time t_{20}, task T4 completes execution and is removed from *Job_Queue*. Task T3 is at the head of the queue and resumes execution of its code outside the rendezvous.
- At time t_{21}, task T3 completes execution and is removed from *Job_Queue*. Server task S2, which is at the head of the queue, resumes execution on behalf of T1. Hence, S^* is reset to *NULL* since there is no server currently executing on behalf of a task other than task T1.
- At time t_{22a}, S2 completes its rendezvous with T1 and resumes its original priority of 2. Task T2, which is at the head of *Job_Queue*, needs to rendezvous with S2. Hence, at time t_{22b}, S2 inherits task T2's priority and takes the position at the head of *Job_Queue*. The pair S2/T2 is added to *Server_Stack* and S^* becomes S1.
- At time t_{23}, S2 completes its rendezvous with task T2 and regains its original priority. Hence, the pair S2/T1 is removed from *Server_Stack*. Task T2 is at the head of *Job_Queue* and resumes execution at its own priority.
- At time t_{24}, task T2 completes execution and is removed from *Job_Queue*. Server task S1 resumes its rendezvous with T1 at T1's priority. Hence S^* becomes *NULL*.
- At time t_{25}, S1 completes its rendezvous with T1, thereby regaining its original priority of 1. Subsequently, task T1 resumes execution at its base priority. The pair S1/T1 is removed from *Server_Stack* and since *Server_Stack* becomes empty, S^* obviously remains *NULL*.
- At time t_{26}, task T1 completes execution and is removed from *Job_Queue*. Server task S4 is now at the head of *Job_Queue*.

The system actions on the run-time data structures *Job_Queue*, *Server_Stack* and *S_Star* are presented in Figure C-4.

Time	(Head) Job_Queue (Tail)	Priority	Server_Stack	S^*/Ceiling
t_0				
t_1	T1, S4, S3, S2, S1	1		
t_2	S1, T1, S4, S3, S2	1	S1/T1	
t_3	S2, S1, T1, S4, S3	1	S2/S1, S1/T1	
t_4	T2, S2, S1, T1, S4, S3	2	S2/S1, S1/T1	S2/2
t_5	S2, T2, S1, T1, S4, S3	2	S2/S1, S1/T1	S2/2
t_6	T3, S2, T2, S1, T1, S4, S3	3	S2/S1, S1/T1	S2/2
t_7	S3, T3, S2, T2, S1, T1, S4	3	S3/T3, S2/S1, S1/T1	S2/2
t_8	T4, S3, T3, S2, T2, S1, T1, S4	4	S3/T3, S2/S1, S1/T1	S3/5
t_9	S3, T4, T3, S2, T2, S1, T1	4	S3/T3, S2/S1, S1/T1	S2/2
t_{10}	S4, S3, T4, T3, S2, T2, S1,T1	4	S3/T3, S2/S1, S1/T1	S3/5
t_{11}	T5, S4, S3, T4, T3, S2, T2, S1, T1	5	S3/T3, S2/S1, S1/T1	S3/5
t_{12a}	S3, T5, S4, T4, T3, S2, T2, S1, T1	5	S3/T3, S2/S1, S1/T1	S2/2
t_{12b}	S4, S3, T5, T4, T3, S2, T2, S1, T1	5	S3/T3, S2/S1, S1/T1	S2/2
t_{13}	S3, T5, T4, T3, S2, T2, S1, T1, S4	5	S3/T3, S2/S1, S1/T1	S2/2
t_{14a}	T5, T4, T3, S2, T2, S1, T1, S4, S3	5	S2/S1, S1/T1	S2/2
t_{14b}	S3, T5, T4, T3, S2, T2, S1, T1, S4	5	S3/T5, S2/S1, S1/T1	S2/2
t_{15}	S4, S3, T5, T4, T3, S2, T2, S1, T1	5	S3/T5, S2/S1, S1/T1	S2/2
t_{16}	S3, T5, T4, T3, S2, T2, S1, T1, S4	5	S3/T5, S2/S1, S1/T1	S2/2
t_{17}	T5, T4, T3, S2, T2, S1, T1, S4, S3	5	S2/S1, S1/T1	S2/2
t_{18a}	T4, T3, S2, T2, S1, T1, S4, S3	4	S2/S1, S1/T1	S2/2
t_{18b}	S4, T4, T3, S2, T2, S1, T1, S3	4	S4/T4, S2/S1, S1/T1	S2/2
t_{19}	T4, T3, S2, T2, S1, T1, S3, S4	4	S2/S1, S1/T1	S2/2
t_{20}	T3, S2, T2, S1, T1, S3, S4	3	S2/S1, S1/T1	S2/2
t_{21}	S2, T2, S1, T1, S3, S4	2	S2/S1, S1/T1	
t_{22a}	T2, S1, T1, S3, S4, S2	2	S1/T1	S1/1
t_{22b}	S2, T2, S1, T1, S3, S4	2	S2/T2, S1/T1	S1/1
t_{23}	T2, S1, T1, S3, S4, S2	2	S1/T1	S1/1
t_{24}	S1, T1, S3, S4, S2	1	S1/T1	
t_{25}	T1, S3, S4, S2, S1	1		
t_{26}	S3, S4, S2, S1	4		

Figure C-4: Runtime System State Information for Example #2

Table of Contents

1. Introduction **1**
1.1. Background 1
1.2. Priority Inversion 2
1.3. Real-Time Scheduling Protocols 3
1.3.1. Use of Ada Features 4
1.3.2. Coding Restrictions 4
1.4. Strategies for Implementing the Protocols 5

2. Basic Priority Inheritance Protocol **7**
2.1. Definition of the Basic Priority Inheritance Protocol for Ada 7
2.2. Implementing the Basic Priority Inheritance Protocol in an Ada Runtime 8
2.2.1. Runtime Data Structures 9
2.2.2. Implementation Mechanisms 9
2.2.3. Runtime Modifications 10
2.2.3.1. Global Data 11
2.2.3.2. Entry Calls 11
2.2.3.3. Selective Wait 11
2.2.3.4. Entry Queue Insertion 12
2.2.3.5. Rendezvous Completion 12
2.3. Optimization Issues 13

3. Priority Ceiling Protocol **15**
3.1. Definition of the Priority Ceiling Protocol for Ada 15
3.2. Implementing the Priority Ceiling Protocol in an Ada Runtime 17
3.2.1. Runtime Data Structures 17
3.2.2. Implementation Mechanisms 18
3.2.3. Runtime Modifications 19
3.2.3.1. Global Data 20
3.2.3.2. Entry Calls 20
3.2.3.3. Rendezvous Completion 22
3.3. Optimization Issues 23

References **25**

Appendix A. Runtime Support Routines **27**
A.a. Transmit_Callers_Priority Subprogram 27
A.b. Determine_Blocking_Task Subprogram 27
A.c. Check_Job_Queue 28

Appendix B. Basic Priority Inheritance Implementation Examples **29**
B.a. Basic Priority Inheritance Protocol — Example #1 29
B.b. Basic Priority Inheritance Protocol — Example #2 33

Appendix C. Priority Ceiling Protocol Implementation Examples **39**
C.a. Priority Ceiling Protocol — Example #1 39
C.b. Priority Ceiling Protocol — Example #2 43

List of Figures

Figure 2-1: Basic Priority Inheritance: Entry Calls 11
Figure 2-2: Basic Inheritance: Entry Queue Insertion 12
Figure 3-1: Priority Ceiling: Entry Calls 21
Figure 3-2: Priority Ceiling: Rendezvous Completion 22
Figure B-1: Basic Priority Inheritance Protocol — Example #1 30
Figure B-2: Runtime System State Information for Example #1 33
Figure B-3: Basic Priority Inheritance Protocol — Example #2 35
Figure B-4: Runtime System State Information for Example #2 38
Figure C-1: Priority Ceiling Protocol — Example #1 40
Figure C-2: Runtime System State Information for Example #1 43
Figure C-3: Priority Ceiling Protocol — Example #2 44
Figure C-4: Runtime System State Information for Example #2 48