THE PRIORITY CEILING PROTOCOL: A METHOD FOR MINIMIZING THE BLOCKING OF HIGH PRIORITY ADA TASKS

John B. Goodenough
Software Engineering Institute
Carnegie-Mellon University
Pittsburgh, PA 15213, USA

Lui Sha
Software Engineering Institute
Carnegie-Mellon University
Pittsburgh, PA 15213, USA

1. INTRODUCTION

At the First International Workshop on Real-Time Ada Issues, it was reported that a high-priority Ada task can be delayed indefinitely by lower priority tasks under certain conditions [1].[1] The delay of a high-priority task by a lower priority task is called *priority inversion*. Priority inversion occurs because of task synchronization requirements and, therefore, cannot be completely eliminated; but it can be minimized. Two recently defined protocols for scheduling real-time processes provide solutions to the priority inversion problem [4]. Both protocols assume the use of binary semaphores to synchronize access to shared resources. In this paper, we extend these protocols to Ada.

The *basic priority inheritance protocol*, as defined in [4], has the following important property:

> If m distinct semaphores are used by n lower priority processes, a process can be blocked for at most the duration of $min(m, n)$ critical sections [4].

Hence, the basic priority inheritance protocol places an upper bound on the blocking delay that a process can encounter. The *priority ceiling protocol* [4] improves the basic priority inheritance protocol by minimizing the blocking time of a process to at most the duration of a single (outermost) critical section of a lower priority process. It also prevents nontrivial forms of deadlock.

The basic priority inheritance and ceiling protocols can be applied to Ada 1) if Ada tasks are written following certain rules and 2) if the rules governing the scheduling of tasks are changed slightly. In essence, the idea is to represent each semaphore as a *server* task. Each critical region is represented as an entry of the task. For example, consider two processes, J_1 and J_2, that access a single semaphore, S:

J_1 = {..., **P**(S), ..., **V**(S), ..., **P**(S), ..., **V**(S), ...}
J_2 = {..., **P**(S), ..., **V**(S), ...}

The corresponding Ada task structure is shown in Figure 1.

To apply the priority ceiling protocol in Ada, a programmer must obey the following restrictions on the use of Ada tasking features:

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. (Such a task structure models the notion of critical regions guarded by a semaphore, thus allowing us to apply 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 contains at least

[1]These conditions are not unique to Ada. They can arise in any system in which the need to access shared resources causes a low-priority task to block the execution of a higher priority task.

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

```
task body T1 is              -- corresponds to J1
begin
    ...
    Server.E1;               -- corresponds to first critical region of J1
    ...
    Server.E2;               -- corresponds to second critical region of J1
    ...
end T;

task body T2 is              -- corresponds to J2
begin
    ...
    Server.E3;               -- corresponds to critical region of J2
    ...
end T2;

task body Server is          -- contains critical regions guarded by semaphore S
begin
    loop
        select
            accept E1 do ... end E1;        -- J1's first critical region
        or  accept E2 do ... end E2;        -- J1's second critical region
        or  accept E3 do ... end E3;        -- J2's critical region
        or  terminate;
        end select;
    end loop;
end Server;
```

Figure 1: Critical Regions Mapped as Ada Tasks

one entry call.)[2]

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; they are excluded to simplify our application of the theory to Ada.)

3. Each task must be assigned a priority. (Hence, the execution of one task can be preempted by the execution of a higher priority task.)

4. A server task must have a priority lower than that of any of its client tasks.[3] (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.)

Further theoretical work may allow the priority ceiling protocol to be extended to a wider variety of Ada tasking structures, but the purpose of this paper is to indicate the advantages of the protocol when it is applied directly to Ada. Given these restrictions, using the priority ceiling protocol on a single processor guarantees the following properties [4]:

1. The queue for a particular entry can have at most one calling task. (Hence, priority queues are not needed.)

---

[2]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 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.

2. If a task is queued for one alternative of a select statement, no tasks are queued for any other alternatives. (Hence, at most one entry call is queued for a given server task and there is no need to specify that in a selective wait, the highest priority queued task is selected for execution; there is at most one waiting task.)

3. Execution of a high-priority non-server task can be delayed by at most one lower priority client task for at most the duration of the longest entry call made by that task. (This is a bounded form of priority inversion.)

4. Deadlock cannot occur as long as a task does not call itself.[4]

5. Blocked calls to the same server are serviced in order of priority.

The following section defines the priority ceiling protocol for Ada.

2. THE PRIORITY CEILING PROTOCOL FOR ADA

A *server* task is a task 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 version of the priority ceiling protocol. A *client* task is a non-server task that contains at least one entry call. A server task is said to be *executing on behalf of* client task T if the server has been called[5] either by T or by a server task that is executing on behalf of 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. For example, suppose client task T_1 calls server 1. In addition, client task T_2 calls server 2, and server 2 calls server 1 during its rendezvous with task T_2. The priority ceiling of server 1 is the maximum priority of tasks T_1 and T_2.

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.[6]

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,

    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.

Definition 1 is used later when defining how blocked calls are processed. Definition 2 is used later to define the execution priority of a server task. Definition 2b defines a form of blocking that can occur when a task is not making or executing an entry call.

Given the restricted usage of Ada constructs and the definitions of blocking, the priority ceiling protocol is defined as follows:

1. When an attempted entry call is blocked, the call is not made and the calling task's execution is

---

[4]Given the mapping from critical regions to Ada tasks, having a task call itself would be equivalent to trying to lock a semaphore while inside a critical region guarded by that semaphore. The priority ceiling protocol prevents deadlock caused by mutual waiting; e.g., under this protocol, it cannot be the case that server task S1 is attempting to call server task S2 while S2 is attempting to call S1.

[5]The calling task is either queued on an entry or the server is in rendezvous with the caller.

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

suspended.[7] (This ensures that at most one task is queued for a server task.)

2. A server executes at its assigned priority except when it is executing on 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. (This rule means a server task can execute at higher than its assigned priority even if it is not in a rendezvous. 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.)

3. If a server blocks the execution of one or more tasks, the server executes with the highest priority of the tasks it blocks[8] 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. 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. This is a form of priority inversion.)

Note that the priority of a server task changes depending on which tasks it is serving or blocking, and that a server can block tasks that are not requesting service; i.e., a calling task can be blocked even though the called server is not executing any call.[9] Moreover, when a rendezvous is completed, the set of blocked tasks can change; in particular, (at most) one task can be unblocked.

The basic priority inheritance protocol is like the priority ceiling protocol except for the definition of blocking: an attempted call is blocked (Definition 1) and a server blocks a calling task (Definition 2a) only if the called task is executing on behalf of some task, i.e., blocking does not take into account priority ceilings or whether other servers are executing.

In [4], the following theorem was proved:[10]

Theorem 1: A set of n periodic non-server tasks using the priority ceiling protocol can be scheduled by the rate monotonic algorithm [2] if the following conditions are satisfied:

$$\forall\, i,\ 1 \le i \le n,\quad \frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_i}{T_i} + \frac{B_i}{T_i} \le i(2^{1/i} - 1),$$

where,

- C_i is the execution time of non-server task τ_i (the execution time includes the time required to execute an entry call when there is no preemption or blocking).
- T_i is the period of non-server task τ_i.
- B_i is the worst-case blocking time of non-server task τ_i, i.e., the longest entry call that can be made by a lower priority client task that calls a server whose priority ceiling is greater than or

[7]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 [4]. 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.

[8]The priority of the blocked tasks will always be higher than the priority of the server's client task.

[9]For example, suppose server S1 has been called by a task at priority 1 and then a task at priority 2 attempts to call server S2. If S1's priority ceiling is equal to or higher than 2, task 2's call will be blocked even though S2 is able to accept it. This seemingly unnecessary blocking is in fact the key to the success of the priority ceiling protocol. The protocol ensures that when a task preempts a server and attempts to call another server, the new server will never block a higher priority task if the call is successful.

[10]The theorem has been reworded to apply to Ada.

equal to τ_i's priority.[11]

Task τ_i has a higher priority than task τ_j if $i < j$.[12]

For example, suppose that we have three non-server tasks. For highest priority task τ_1, we first check if equation $C_1/T_1 + B_1/T_1 \leq 1$ holds. Next, we check if equation $C_1/T_1 + C_2/T_2 + B_2/T_2 \leq 2(2^{1/2} - 1)$ holds for task τ_2. Finally, we check if equation $C_1/T_1 + C_2/T_2 + C_3/T_3 + B_3/T_3 \leq 3(2^{1/3} - 1)$ holds for task τ_3. If all three equations hold, then the tasks will meet all their deadlines; i.e., each task will complete its work before the start of its next period.

The first i terms in the theorem's inequality constitute the effect of preemptions from all higher priority tasks and τ_i's own execution time, while B_i in the last term represents the worst-case delay caused by the execution of lower priority tasks. Blocking delays occur because of task synchronization requirements. The theorem specifies a sufficient (worst-case) condition that characterizes the rate-monotonic schedulability of a given periodic task set. Tighter bounds were also given in [4].

3. SUMMARY

Both the basic priority inheritance protocol and the priority ceiling protocol correct the unbounded priority inversion problem caused by existing Ada rules. However, the priority ceiling protocol minimizes the blocking of high-priority tasks better than the basic priority inheritance protocol. Under the priority ceiling protocol, a task can be blocked by lower priority tasks at most once. In addition, this protocol prevents mutual deadlocks as long as each task, when executing alone, does not deadlock with itself. Finally, the implementation of this protocol may well be simpler than the current rules since at most one task can be queued per server task and only one priority-ordered queue need be kept—the queue of tasks that are ready to run or blocked. It also seems likely that a useful set of real-time applications can be programmed using the limited form of server tasks currently allowed by the protocol (see the companion paper [3]). Further work is underway to extend the protocol and verify its utility.

4. ACKNOWLEDGEMENT

We are grateful to Jan Storbank Pedersen for suggestions that helped to clarify and correct earlier drafts of this paper.

This paper has also been published by the Software Engineering Institute as CMU/SEI-88-SR-4.

REFERENCES

1. Cornhill, D. Tasking Session Summary. Proceedings of ACM International Workshop on Real-Time Ada Issues, 1987, pp. 29-32.

2. Liu, C. L. and Layland, J. W. "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment". *JACM* **20** (1973), 46-61.

3. Locke, D. and Goodenough, J. B. A Practical Application of the Ceiling Protocol in a Real-Time System. Tech. Rept. CMU/SEI-88-SR-3, Software Engineering Institute, Carnegie-Mellon University, March, 1988. (see also these proceedings).

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

---

[11]Note that task τ_i can have a non-zero blocking time (i.e., can be delayed by the execution of a lower priority task) even if it makes no entry calls, since blocking time is determined by entry calls made by lower priority tasks. This is an example of push-through blocking (see the Appendix).

[12]Note that for purposes of applying this theorem, task τ_1 has the highest priority, although in Ada a task with priority n would have the highest priority. In the scheduling literature, the notational convention is that task τ_i's priority is higher than τ_j if $i < j$. In addition, use of the rate monotonic algorithm means that $T_i < T_j$ (i.e., higher priority tasks have shorter periods).

APPENDIX

A given task's execution can be delayed in two ways: either it is preempted by the execution of a higher priority non-server task (or the execution of a server on behalf of such a task), or it is blocked by a server executing on behalf of a lower priority task. The priority ceiling protocol limits the amount of blocking of a task to at most one server call made by a lower priority task, whereas the amount of blocking for the basic priority inheritance protocol can be more. Blocking can occur in three ways:

- *Direct* blocking: the called task either has a queued task or is executing a rendezvous. (This is the normal form of Ada blocking and is the price paid for consistency in shared data.)
- *Push-through* blocking: a medium-priority task is unable to execute because a server task executing on behalf of a lower priority task is blocking the execution of a high-priority task and, therefore, is executing instead of the medium-priority task. (This is the price paid to avoid indefinite blocking of high-priority tasks.)
- *Ceiling* blocking: although the called task would normally be able to accept the call (because it is not executing on behalf of any other task), some server task is executing with a priority ceiling higher than or equal to that of the calling task. (This is the price paid to avoid deadlock and chained blocking. Although it sometimes introduces extra delays in servicing a call, the overall effect is to reduce the worst-case blocking delays for high-priority tasks.)

An example showing the effect of the priority ceiling protocol is given in Figure 2. This figure shows the execution of seven tasks, including two server tasks. The non-server tasks are labeled with their priority, e.g., task 5 has the highest priority. The server tasks have the lowest priority: S1 has priority 0 and S2 has priority -1. The calling relationships among the tasks are as follows:

- Task 5 calls an entry of server task S2. The entry call takes one unit of time.
- Task 4 calls an entry of server task S1. The entry call takes one unit of time.
- Task 3 makes no entry calls.
- Task 2 calls an entry of server task S2; during this rendezvous, S2 calls an entry of task S1. (This illustrates the effect of the protocol for nested entry calls.) The entry call takes 4 units of time, including the time used by the rendezvous with S1.
- Task 1 calls an entry of server task S1. The entry executes using 4 units of time.

Since S1 is called on behalf of tasks 1, 2, and 4, S1's priority ceiling is 4. Since S2 is called on behalf of tasks 2 and 5, its priority ceiling is 5.

The actions illustrated by Figure 2 are explained below, but the overall effect to note is that the higher priority tasks complete their execution much earlier under the priority ceiling protocol than under the basic priority inheritance protocol (illustrated in Figure 3).

- At time t_1, all tasks are activated, but only tasks 1, S1, and S2 are ready to run. Since task 1 has the highest priority, its execution begins.
- At time t_2, task 1 attempts to call server S1. Since no server tasks are executing on behalf of any task, 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. Eventually its select statement is executed and the waiting call is accepted; the rendezvous starts. Execution continues at priority 1.
- At time t_3, task 2 starts its execution. Since task 2 has higher priority than the current priority of server tasks S1, S2, and task 1, the execution of these tasks is preempted.

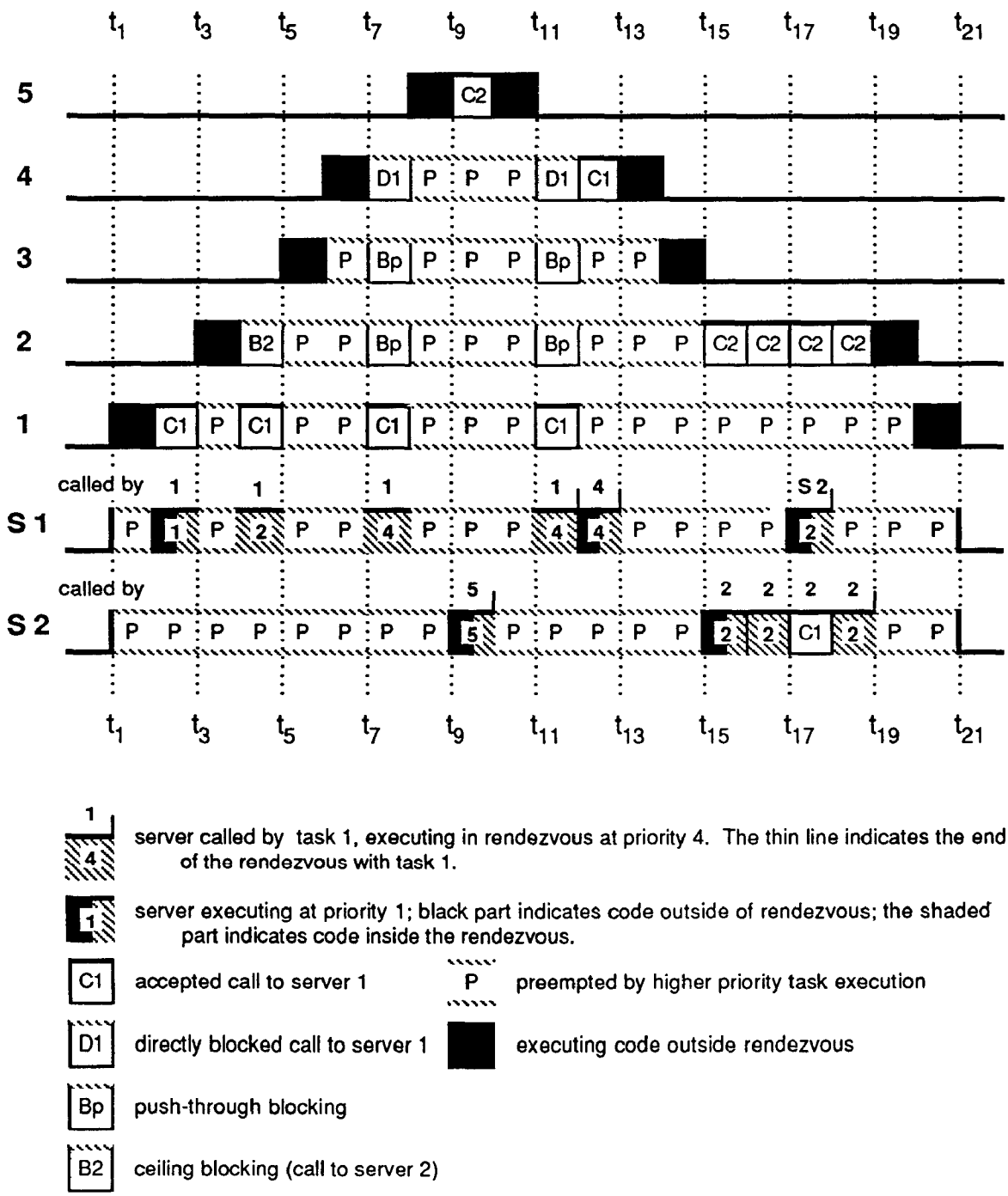


Figure 2: The Priority Ceiling Protocol

- At time t_4, task 2 attempts to call server task S2. We now examine the priority ceilings of all server tasks that are executing on behalf of some task other than task 2. There is only one (server task S1) and its priority ceiling is 4. Since this priority ceiling exceeds task 2's current priority, task 2 is blocked; i.e., its execution is suspended and the entry call is not made. In particular, task 2 is *not* queued for S2. Since S1 blocks the execution of task 2, S1 inherits task 2's priority. S1 is now the highest priority task that can run.

  The blocking of the call to S2 is an example of ceiling blocking. This kind of blocking occurs because of the priority ceiling protocol. Although it may seem counter-intuitive to block task 2 even though its call can otherwise be accepted, the example given here shows that the overall effect on timing behavior is beneficial.

- At time t_5, task 3 starts its execution. Since its priority is higher than the current priority of any task that is ready to run, it preempts the execution of tasks S2, S1, 1, and 2.

- At time t_6, task 4 starts its execution. Since its priority is higher than the execution priority of any other task that is ready to run, it preempts the execution of tasks S2, S1, 1, 2, and 3.

- At time t_7, task 4 attempts to call server task S1. We examine the priority ceiling of all server tasks that are executing on behalf of some task other than task 4. S1 is the only such task (it is executing on behalf of task 1). S1's priority ceiling is 4, which is equal to the priority of the calling task, so task 4's execution is blocked. In this case, since task 4 is calling S1 and S1 is already in rendezvous with another task, the call could not be accepted in any case. This is, therefore, an example of *direct* blocking.

  Since S1 is blocking task 4, its execution priority is increased to 4. S1 is now the highest priority task able to execute. Tasks 2 and 3 are now blocked from execution because S1 is executing with a higher priority. Since S1 is executing on behalf of task 1, its execution priority would normally be 1. If it did not inherit the priority of the task it is blocking, task 4 would still be blocked, but task 3 would be the highest priority task able to run and would preempt the execution of tasks 2, 1, S1, and S2. The effect would be to delay task 4 even longer. The blocking of tasks 2 and 3 is an example of *push-through* blocking.

- At time t_8, task 5 starts its execution. Since its priority is higher than the current priority of any executing task, S1's execution is preempted, and task 5 starts to execute.

- At time t_9, task 5 attempts to call S2. We examine the priority ceiling of all server tasks that are executing on behalf of some task other than task 5. S1 is the only such task (executing on behalf of task 1). S1's priority ceiling is 4, which is less than task 5's priority, so the call can be accepted. Since S2 has not yet had an opportunity to execute its select statement, task 5 is queued. S2 is given the priority of its queued task and its execution starts. When its select statement is executed, the waiting call is accepted, and the rendezvous starts; execution continues at priority 5.

  Task 5's call to S2 can be accepted because task 2's attempted call (at time t_4) was blocked. This is where the blocking of task 2 pays off—a high-priority task that would otherwise be blocked while S2 is executing can now be serviced. Comparison with the basic priority inheritance protocol (Figure 3) is worthwhile.

- At time t_{10}, the rendezvous with S2 is completed. S2 reverts to its assigned priority, so its execution is preempted by the execution of task 5.

- At time t_{11}, task 5 completes its execution. Task 4's call to S1 is still blocked since S1 has not yet finished its rendezvous. S1 has priority 4 and therefore continues its execution.

- At time t_{12}, task S1 completes its rendezvous. Its priority returns to normal. S1 has been blocking the execution of tasks 2, 3, and 4. Completion of the rendezvous means tasks 2 and 4 are no longer blocked because S1 is no longer executing on behalf of any task. In addition, S1's low-priority means it no longer blocks task 3. So all tasks are eligible to run. Since task 4 has the highest priority, its execution resumes.

  Task 4 was suspended just before making the call to S1. Now that its execution has resumed, it again attempts to call S1. Since there are no server tasks executing on behalf of any task, the call succeeds.

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.

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.

Note that if task 4 had attempted to call S2 instead of S1, its call would have been serviced before task 2's call, even though task 2 made its call first. In effect, calls are automatically serviced in order of priority under the priority ceiling (and the basic priority inheritance) protocol.

- At time t_{13}, task S1 completes its rendezvous. Its priority returns to normal. All tasks are now eligible to run. Since task 4 has completed its call to S1 and has the highest priority, its execution resumes.
- At time t_{14}, task 4 completes its execution. Task 3 now has the highest priority and resumes its execution.
- At time t_{15}, task 3 completes its execution. Task 2 now has the highest priority and resumes its execution by attempting to call server S2. Since no server tasks are executing on behalf of any tasks, the call succeeds. 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.

  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. S2 is executing on behalf of task 2. We check to see if there are any other server tasks that are executing on behalf of some task other than task 2. Since there are no such tasks, it doesn't matter that S2's current priority is less than S1's priority ceiling; the call succeeds.

  The call is queued, and S1 starts executing at priority 2, the current priority of its caller, S2. Eventually the call is accepted and the rendezvous begins.

  If during this execution of S1, task 5 started execution and attempted to call S2, then S2 and S1 would block the execution of task 5, and S1 would therefore resume execution at priority 5. Note that priority 5 is higher than S1's priority ceiling. The priority ceiling is only determined by the priority of tasks that call a server directly or indirectly; it is not affected by the priority of tasks that a server can block. Therefore, a server's priority ceiling is not its maximum execution priority but, instead, reflects the maximum priority of its client tasks.
- At time t_{18}, S1 completes execution of the rendezvous. S1's priority returns to normal and its execution is preempted by S2, which is still executing at priority 2. S2 continues executing its rendezvous, at the priority of its calling task.
- At time t_{19}, S2 completes execution of its rendezvous. Its priority returns to normal, so its execution is preempted by task 2.
- At time t_{20}, task 2 completes its execution. Task 1 can now resume its execution.
- At time t_{21}, task 1 completes its execution.

We now determine the worst-case blocking time for each task. Under the priority ceiling protocol, a server S can block a non-server task J:

- if the priority ceiling of S is higher than or equal to the priority of J, and
- if there exists a lower priority task which may call S.

In this example, the maximal entry call duration of both S_1 and S_2 is 4. Hence, if a task can be blocked once by either S_1 or S_2, the blocking time is 4. Finally, recall that the priority ceilings of S_1 and S_2 are 4 and 5, respectively.

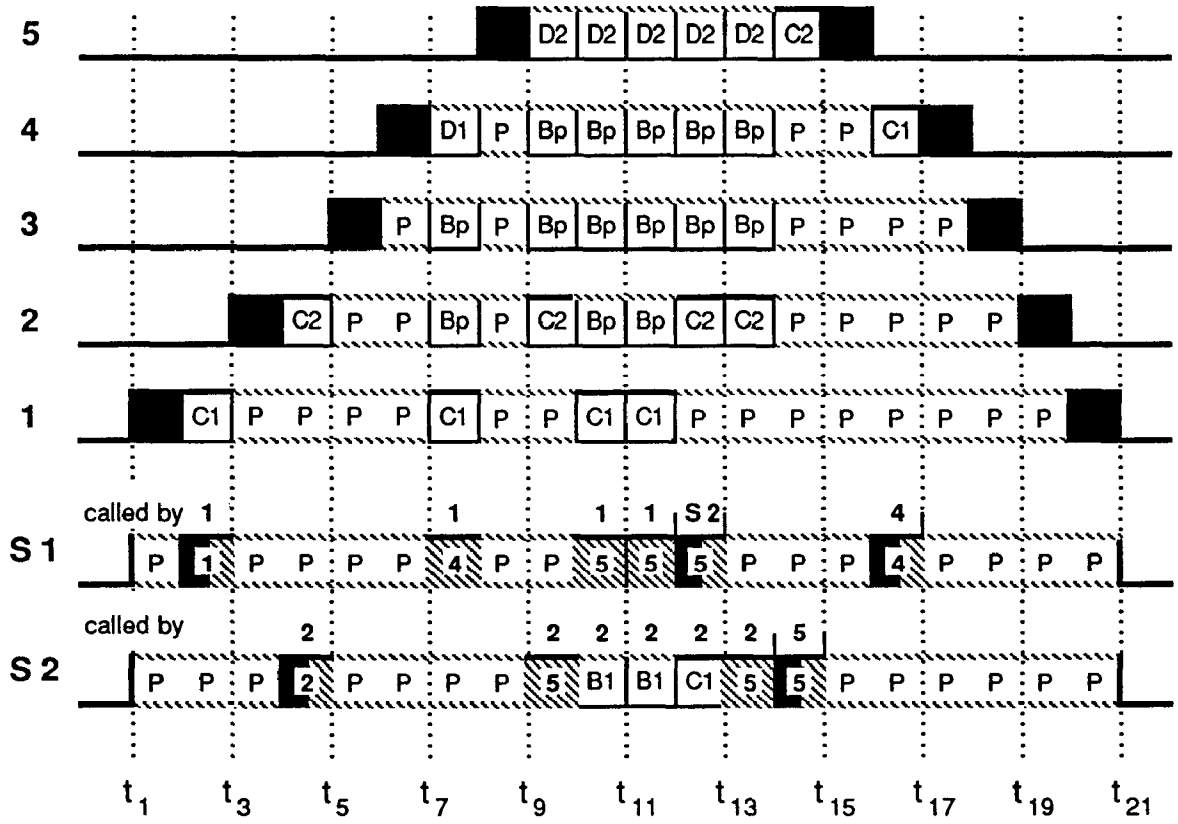
The worst-case blocking time for each task is as follows.

- The worst-case blocking time for task 1, B_1, is zero since there are no lower priority client tasks that can invoke a server.
- The worst-case blocking time for task 2, B_2, is 4 because task 2's priority is lower than the ceiling of server S_1. In addition, S_1 can be called by lower priority task 1. Task 2's priority is also lower than the priority ceiling of S_2, but since no lower priority client task calls S_2, task 2's blocking time is independent of the time taken for any entry calls to S_2.
- The worst-case blocking time for task 3, B_3, is 4 because task 3's priority is lower than the priority ceilings of S_1 and S_2. In addition, either server can be called by a lower priority task, and the maximum duration of each such call is 4. Note that task 3 has a non-zero blocking time even though it makes no entry calls.
- The worst-case blocking time for task 4, B_4, is 4 because task 4's priority is not higher than the ceilings of servers S_1 and S_2. Both servers can be called by lower priority tasks, and the maximum duration of each such call is 4.
- The worst-case blocking time for task 5, B_5, is 4 because server S_2's priority ceiling is equal to the priority of task 5, S_2 can be called by task 2, and task 2's entry call takes 4 units of time.

Note that to minimize the effect of the blocking times, entry calls of low-priority tasks should be short.

The effect of the basic priority inheritance protocol is indicated in Figure 3.

- At time t_1, S1 and S2 are preempted and task 1's execution begins.
- At time t_2, task 1 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, task 2 starts its execution. Since task 2 has higher priority than the current priority of tasks S1, S2, and task 1, the execution of these tasks is preempted.
- At time t_4, task 2 attempts to call server task S2. Since S2 is not executing on behalf of any task, the call is accepted (unlike the case of the priority ceiling protocol). 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, task 3 starts its execution. Since its priority is higher than the current priority of any task that is ready to run, it preempts the execution of tasks S2, S1, 1, and 2.
- At time t_6, task 4 starts its execution. Since its priority is higher than the execution priority of any other task that is ready to run, it preempts the execution of tasks S2, S1, 1, 2, and 3.
- At time t_7, task 4 attempts to call server task S1. Since S1 is executing on behalf of task 1, task 4 is blocked; i.e., its execution is suspended and the entry call is not made. Since S1 now blocks the execution of task 4, S1 inherits task 4's priority and resumes execution.
- At time t_8, task 5 starts its execution. Since its priority is higher than the current priority of any executing task, S1's execution is preempted and task 5 starts to execute.
- At time t_9, task 5 attempts to call S2. S2 is executing on behalf of task 2, so task 5 is blocked. Since S2 now blocks the execution of task 5, it inherits task 5's priority and is now the highest priority task ready to run. Note that under the priority ceiling protocol, task 2's earlier call to S2 had been blocked so that task 5's call could be accepted.
- At time t_{10}, S2 attempts to call S1. Since S1 is executing on behalf of task 1, this call is blocked and S1 inherits the current priority of S2. S1 is now the highest priority task ready to run; it resumes



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 3: The Basic Inheritance Protocol

execution at priority 5.

The situation at t_{10} illustrates how chained blocking can arise under the basic priority inheritance protocol. Task 5's call cannot be accepted by S2 until both S2 and S1 complete their execution on behalf of tasks 1 and 2. Under the priority ceiling protocol, server calls on behalf of at most one task need to be completed. Under the 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. Its priority returns to normal. S1 has been blocking tasks 4 and 5. Since S2 is still blocking task 5, 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 task 4 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}, task S1 completes its rendezvous. Its priority returns to normal. All tasks are now eligible to run. Since S2 is still blocking task 5, its execution resumes at priority 5.
- At time t_{14}, S2 completes its rendezvous. Its priority returns to normal and no longer blocks task 5. Task 5 is now the highest priority ready to run, and its call to S2 succeeds. S2 resumes execution at priority 5.
- At time t_{15}, S2 completes task 5's call. Its priority returns to normal. Task 5 continues its execution.
- At time t_{16}, task 5 completes its execution. Task 4 is now the highest priority task ready to execute. Its call to S1 succeeds, and S1 starts executing at priority 4.
- At time t_{17}, S1 completes task 4's call. Its priority returns to normal, and task 4 continues its execution.
- At times $t_{18\text{-}21}$, tasks 4, 3, 2, and 1 complete their execution.