7

RESOURCE ACCESS PROTOCOLS

7.1 INTRODUCTION

A *resource* is any software structure that can be used by a process to advance its execution. Typically, a resource can be a data structure, a set of variables, a main memory area, a file, or a set of registers of a peripheral device. A resource dedicated to a particular process is said to be *private*, whereas a resource that can be used by more tasks is called a *shared resource*. A shared resource protected against concurrent accesses is called a *mutually exclusive resource*.

To ensure consistency of the data structures in mutually exclusive resources, any concurrent operating system should use appropriate resource access protocols to guarantee a mutual exclusion among competing tasks. A piece of code executed under mutual exclusion constraints is called a *critical section*.

Any task that needs to enter a critical section must wait until no other task is holding the resource. A task waiting for a mutually exclusive resource is said to be *blocked* on that resource, otherwise it proceeds by entering the critical section and holds the resource. When a task leaves a critical section, the resource associated with the critical section becomes *free*, and it can be allocated to another waiting task, if any.

Operating systems typically provide a general synchronization tool, called a *semaphore* [Dij68, BH73, PS85], that can be used by tasks to build critical sections. A semaphore is a kernel data structure that, apart from initialization, can be accessed only through two kernel primitives, usually called *wait* and *signal*. When using this tool, each mutually exclusive resource R_i must be protected by a different semaphore S_i and each critical section operating on a resource R_i must begin with a $wait(S_i)$ primitive and end with a $signal(S_i)$ primitive.

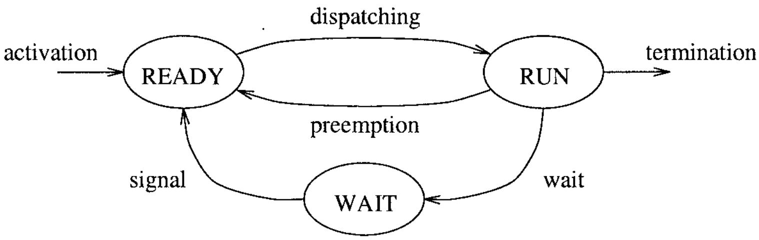

Figure 7.1 Waiting state caused by resource constraints.

All tasks blocked on the same resource are kept in a queue associated with the semaphore that protects the resource. When a running task executes a *wait* primitive on a locked semaphore, it enters a *waiting* state, until another task executes a *signal* primitive that unlocks the semaphore. When a task leaves the waiting state, it does not go in the running state, but in the ready state, so that the CPU can be assigned to the highest-priority task by the scheduling algorithm. The state transition diagram relative to the situation described above is shown in Figure 7.1.

In this chapter, we describe the main problems that may arise in a uniprocessor system when concurrent tasks use shared resources in exclusive mode, and we present some resource access protocols designed to avoid such problems and bound the maximum blocking time of each task. We then show how the schedulability analysis for periodic task sets can be extended to take blocking times into account.

7.2 THE PRIORITY INVERSION PHENOMENON

Consider two tasks J_1 and J_2 that share a mutually exclusive resource R_k (such as a list), on which two operations (such as *insert* and *remove*) are defined. To guarantee the mutual exclusion, both operations must be defined as critical sections. If a binary semaphore S_k is used for this purpose, then each critical section must begin with a *wait*(S_k) primitive and must end with a *signal*(S_k) primitive (see Figure 7.2).

If preemption is allowed and J_1 has a higher priority than J_2, then J_1 can be blocked in the situation depicted in Figure 7.3. Here, task J_2 is activated first, and, after a while, it enters the critical section and locks the semaphore. While J_2 is executing the critical section, task J_1 arrives and, since it has a higher priority, it preempts J_2 and starts executing. However, at time t_1, when attempting to enter its critical section, J_1 is blocked on the semaphore, so J_2 resumes. J_1 has to wait until time t_2, when J_2

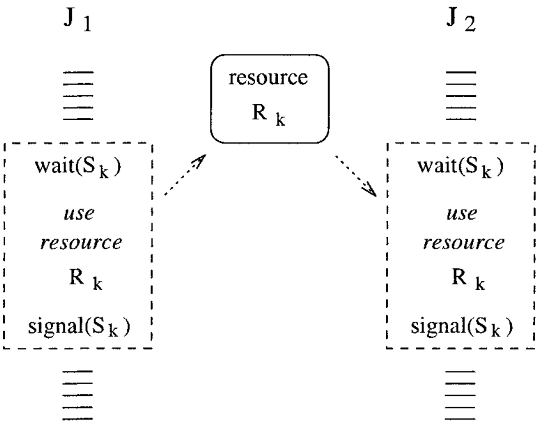

Figure 7.2 Structure of two tasks that share an exclusive resource.

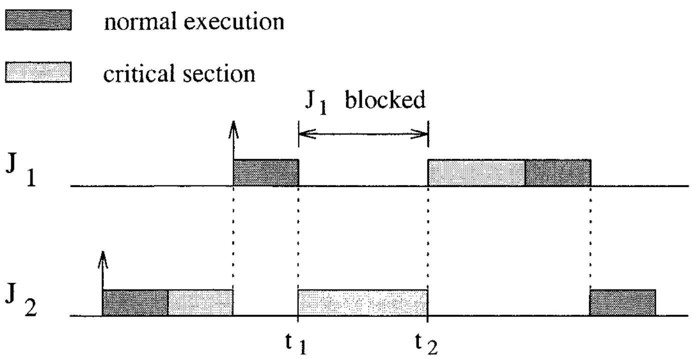

Figure 7.3 Example of blocking on an exclusive resource.

releases the critical section by executing the *signal(S_k)* primitive, which unlocks the semaphore.

In this simple example, the maximum blocking time that J_1 may experience is equal to the time needed by J_2 to execute its critical section. Such a blocking cannot be avoided because it is a direct consequence of the mutual exclusion necessary to protect the shared resource against concurrent accesses of competing tasks.

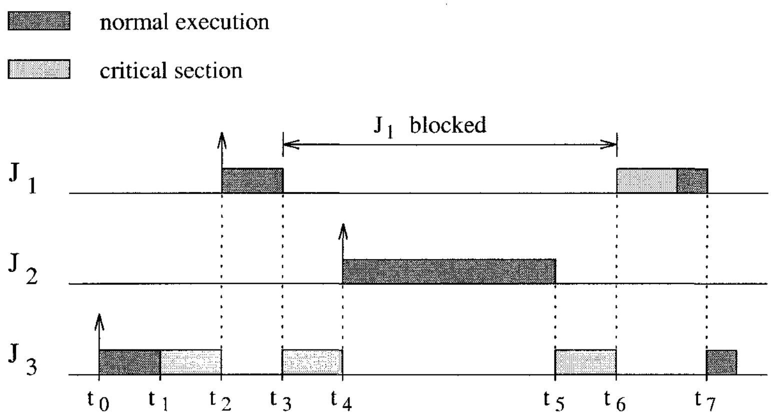

Figure 7.4 An example of priority inversion.

Unfortunately, in the general case, the blocking time of a task on a busy resource cannot be bounded by the duration of the critical section executed by the lower-priority task. In fact, consider the example illustrated in Figure 7.4. Here, three tasks J_1, J_2, and J_3 have decreasing priorities, and J_1 and J_3 share an exclusive resource protected by a binary semaphore S.

If J_3 starts at time t_0, it may happen that J_1 arrives at time t_2 and preempts J_3 inside its critical section. At time t_3, J_1 attempts to use the resource, but it is blocked on the semaphore S; thus, J_3 continues the execution inside its critical section. Now, if J_2 arrives at time t_4, it preempts J_3 (because it has a higher priority) and increases the blocking time of J_1 by all its duration. As a consequence, the maximum blocking time that J_1 may experience does depend not only on the length of the critical section executed by J_3 but also on the worst-case execution time of J_2! This is a situation that, if it recurs with other medium-priority tasks, can lead to uncontrolled blocking and can cause critical deadlines to be missed. A *priority inversion* is said to occur in the interval $[t_3, t_6]$, since the highest-priority task J_1 waits for the execution of lower-priority tasks (J_2 and J_3). In general, the duration of priority inversion is unbounded, since any intermediate-priority task that can preempt J_3 will indirectly block J_1.

Several approaches have been proposed to deal with the problem of scheduling tasks accessing shared resources. A simple solution that avoids the unbounded priority inversion problem is to disallow preemption during the execution of all critical sections. This method, however, is only appropriate for very short critical sections, because it creates unnecessary blocking. Consider, for example, the case depicted in Figure 7.5,

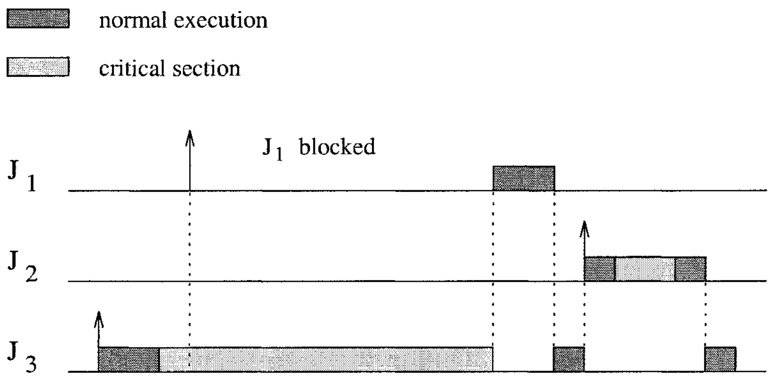

Figure 7.5 Scheduling with non-preemptive critical sections.

where J_1 is the highest priority task that does not use any resource, whereas J_2 and J_3 are low-priority tasks that share a mutually exclusive resource. If the low-priority task J_3 enters a long critical section, J_1 may unnecessarily be blocked for a long period of time.

In other approaches, the priority inversion problem is solved through the use of appropriate protocols that control the accesses to any shared resource. The Priority Inheritance Protocol and the Priority Ceiling Protocol [SRL90] apply to fixed-priority systems,[1] whereas the Stack Resource Policy [Bak91] is suitable both for static and dynamic priority systems. These protocols are described in the following sections.

7.3 PRIORITY INHERITANCE PROTOCOL

The *Priority Inheritance Protocol* (PIP), proposed by Sha, Rajkumar and Lehoczky [SRL90], offers a simple solution to the problem of unbounded priority inversion caused by resource constraints. The basic idea behind this protocol is to modify the priority of those tasks that cause blocking. In particular, when a task J_i blocks one or more higher-priority tasks, it temporarily assumes (*inherits*) the highest priority of the blocked tasks. This prevents medium-priority tasks from preempting J_i and prolonging the blocking duration experienced by the higher-priority tasks. Before describing the protocol in detail, we first introduce the terminology and the basic assumptions made on the system.

[1] The Priority Inheritance Protocol has been extended for EDF by Spuri [Spu95], and the Priority Ceiling Protocol has been extended for EDF by Chen and Lin [CL90].

7.3.1 TERMINOLOGY AND ASSUMPTIONS

Consider a set of n periodic tasks, $\tau_1, \tau_2, \ldots, \tau_n$, which cooperate through m shared resources, $R_1, R_2, \ldots, R_m$. Each task is characterized by a period T_i and a worst-case computation time C_i. The deadline of any periodic instance is assumed to be at the end of its period. Each resource R_k is guarded by a distinct semaphore S_k. Hence, all critical sections on resource R_k begin with a $wait(S_k)$ operation and end with a $signal(S_k)$ operation. The following notation is adopted throughout the discussion:

- J_i denotes a job; that is, a generic instance of task τ_i.
- Since the protocol can modify the priority of the tasks, for each task we distinguish a fixed *nominal* priority P_i (assigned, for example, by the Rate Monotonic algorithm) and an *active* priority p_i ($p_i \geq P_i$), which is dynamic and initially set to P_i.
- $z_{i,j}$ denotes the jth critical section of job J_i.
- $d_{i,j}$ denotes the duration of $z_{i,j}$; that is, the time needed by J_i to execute $z_{i,j}$ without interruption.
- The semaphore guarding the critical section $z_{i,j}$ is denoted by $S_{i,j}$ and the resource associated with $z_{i,j}$ is denoted by $R_{i,j}$.
- We write $z_{i,j} \subset z_{i,k}$ to indicate that $z_{i,j}$ is entirely contained in $z_{i,k}$.

Moreover, the properties of the protocol are valid under the following assumptions:

- Jobs $J_1, J_2, \ldots, J_n$ are listed in descending order of nominal priority, with J_1 having the highest nominal priority.
- Jobs do not suspend themselves (for example, on I/O operations or on explicit synchronization primitives).
- The critical sections used by any task are *properly* nested; that is, given any pair $z_{i,j}$ and $z_{i,k}$, then either $z_{i,j} \subset z_{i,k}$, $z_{i,k} \subset z_{i,j}$, or $z_{i,j} \cap z_{i,k} = \emptyset$.
- Critical sections are guarded by binary semaphores. This means that only one job at a time can be within the critical section corresponding to a particular semaphore S_k.

7.3.2 PROTOCOL DEFINITION

The Priority Inheritance Protocol can be defined as follows:

- Jobs are scheduled based on their active priorities. Jobs with the same priority are executed in a First Come First Served discipline.

- When job J_i tries to enter a critical section $z_{i,j}$ and resource $R_{i,j}$ is already held by a lower-priority job, J_i will be blocked. J_i is said to be blocked by the task that holds the resource. Otherwise, J_i enters the critical section $z_{i,j}$.

- When a job J_i is blocked on a semaphore, it transmits its active priority to the job, say J_k, that holds that semaphore. Hence, J_k resumes and executes the rest of its critical section with a priority $p_k = p_i$. J_k is said to *inherit* the priority of J_i. In general, a task inherits the highest priority of the jobs blocked by it.

- When J_k exits a critical section, it unlocks the semaphore, and the highest-priority job, if any, blocked on that semaphore is awakened. Moreover, the active priority of J_k is updated as follows: if no other jobs are blocked by J_k, p_k is set to its nominal priority P_k, otherwise it is set to the highest priority of the jobs blocked by J_k.

- Priority inheritance is transitive; that is, if a job J_3 blocks a job J_2, and J_2 blocks a job J_1, then J_3 inherits the priority of J_1 via J_2.

EXAMPLES

We first consider the same situation presented in Figure 7.4 and show how the priority inversion phenomenon can be bounded by the Priority Inheritance Protocol. The modified schedule is illustrated in Figure 7.6. Until time t_3 there is no variation in the schedule, since no priority inheritance takes place. At time t_3, J_1 is blocked by J_3, thus J_3 inherits the priority of J_1 and executes the remaining part of its critical section (from t_3 to t_5) at the highest priority. In this condition, at time t_4, J_2 cannot preempt J_3 and cannot create additional interference on J_1. As J_3 exits its critical section, J_1 is awakened and J_3 resumes its original priority. At time t_5, the processor is assigned to J_1, which is the highest-priority task ready to execute, and task J_2 can only start at time t_6, when J_1 has completed. The active priority of J_3 as a function of time is also shown in Figure 7.6 on the lowest timeline.

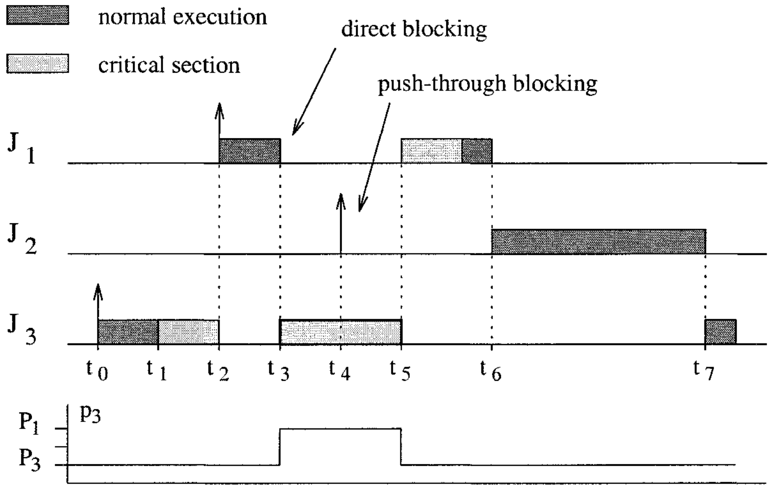

Figure 7.6 Example of Priority Inheritance Protocol.

From this example, we can notice that a high-priority job can experience two kinds of blocking:

- **Direct blocking**. It occurs when a high-priority job tries to acquire a resource already held by a lower-priority job. Direct blocking is necessary to ensure the consistency of the shared resources.
- **Push-through blocking**. It occurs when a medium-priority job is blocked by a lower-priority job that has inherited a higher priority from a job it directly blocks. Push-through blocking is necessary to avoid unbounded priority inversion.

Notice that, in most situations, when a task exits a critical section, it resumes the priority it had when it entered. However, this is not true in general. Consider the example illustrated in Figure 7.7. Here, job J_1 uses a resource R_a guarded by a semaphore S_a, job J_2 uses a resource R_b guarded by a semaphore S_b, and job J_3 uses both resources in a nested fashion (S_a is locked first). At time t_1, J_2 preempts J_3 within its nested critical section; hence, at time t_2, when J_2 attempts to lock S_b, J_3 inherits its priority, P_2. Similarly, at time t_3, J_1 preempts J_3 within the same critical section and, at time t_4, when J_1 attempts to lock S_a, J_3 inherits the priority P_1. At time t_5, when J_3 unlocks semaphore S_b, job J_2 is awakened but J_1 is still blocked; hence, J_3 continues its execution at the priority of J_1. At time t_6, J_3 unlocks S_a and, since no other jobs are blocked, J_3 resumes its original priority P_3.

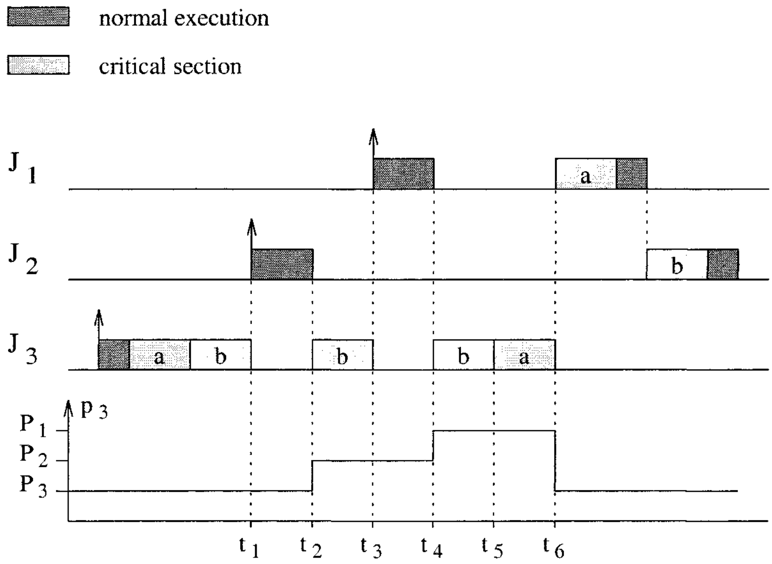

Figure 7.7 Priority inheritance with nested critical sections.

An example of transitive priority inheritance is shown in Figure 7.8. Here, job J_1 uses a resource R_a guarded by a semaphore S_a, job J_3 uses a resource R_b guarded by a semaphore S_b, and job J_2 uses both resources in a nested fashion (S_a protects the external critical section and S_b the internal one). At time t_1, J_3 is preempted within its critical section by J_2, which in turn enters its first critical section (the one guarded by S_a), and at time t_2 it is blocked on semaphore S_b. As a consequence, J_3 resumes and inherits the priority P_2. At time t_3, J_3 is preempted by J_1, which at time t_4 tries to acquire R_a. Since S_a is locked by J_2, J_2 inherits P_1. However, J_2 is blocked by J_3; hence, for transitivity, J_3 inherits the priority P_1 via J_2. When J_3 exits its critical section, no other jobs are blocked by it, thus it resumes its nominal priority P_3. Priority P_1 is now inherited by J_2, which still blocks J_1 until time t_6.

7.3.3 PROPERTIES OF THE PROTOCOL

In this section, the main properties of the Priority Inheritance Protocol are presented. These properties are then used compute the maximum blocking time that each task may experience, in order to analyze the schedulability of a periodic task set.

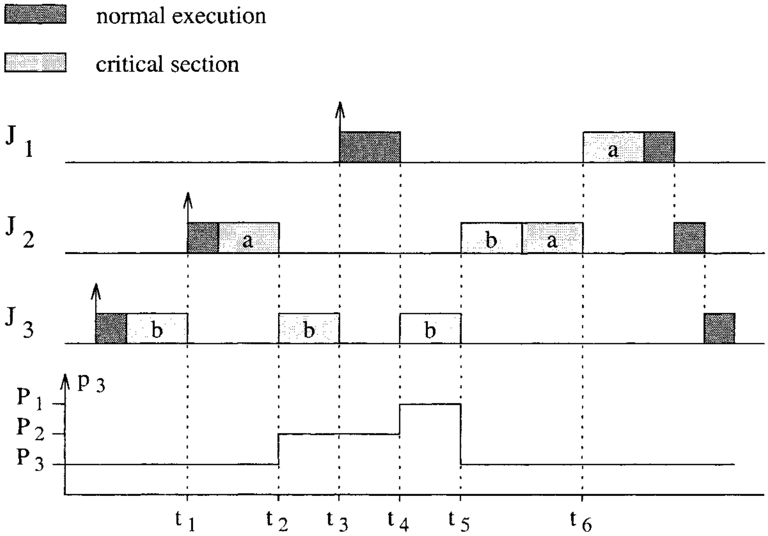

Figure 7.8 Example of transitive priority inheritance.

Lemma 7.1 *A semaphore S_k can cause push-through blocking to job J_i, only if S_k is accessed both by a job with priority lower than P_i and by a job that has or can inherit a priority equal to or higher than P_i.*

Proof. Suppose that semaphore S_k is accessed by a job J_l with priority lower than P_i. If S_k is not accessed by a job that has or can inherit a priority equal to or higher than P_i, then J_l cannot inherit a priority equal to or higher than P_i. Hence, J_l will be preempted by J_i and the lemma follows. □

Lemma 7.2 *Transitive priority inheritance can occur only in the presence of nested critical sections.*

Proof. A transitive inheritance occurs when a high-priority job J_H is blocked by a medium-priority job J_M, which in turn is blocked by a low-priority job J_L (see the example of Figure 7.8). Since J_H is blocked by J_M, J_M must hold a semaphore, say S_a. But J_M is also blocked by J_L on a different semaphore, say S_b. This means that J_M attempted to lock S_b inside the critical section guarded by S_a. The lemma follows.

□

Lemma 7.3 *If there are n lower-priority jobs that can block a job J_i, then J_i can be blocked for at most the duration of n critical sections (one for each of the n lower-priority jobs), regardless of the number of semaphores used by J_i.*

Proof. A job J_i can be blocked by a lower-priority job J_k only if J_k has been preempted within a critical section, say $z_{k,j}$, that can block J_i. Once J_k exits $z_{k,j}$, it can be preempted by J_i; thus, J_i cannot be blocked by J_k again. The same situation may happen for each of the n lower-priority jobs; therefore, J_i can be blocked at most n times. □

Lemma 7.4 *If there are m distinct semaphores that can block a job J_i, then J_i can be blocked for at most the duration of m critical sections, one for each of the m semaphores.*

Proof. Since semaphores are binary, only one of the lower-priority jobs, say J_k, can be within a blocking critical section corresponding to a particular semaphore S_j. Once S_j is unlocked, J_k can be preempted and can no longer block J_i. If all m semaphores that can block J_i are locked by m lower-priority jobs, then J_i can be blocked at most m times. □

Theorem 7.1 (Sha-Rajkumar-Lehoczky) *Under the Priority Inheritance Protocol, a job J can be blocked for at most the duration of $\min(n, m)$ critical sections, where n is the number of lower-priority jobs that could block J and m is the number of distinct semaphores that can be used to block J.*

Proof. It immediately follows from Lemma 7.3 and Lemma 7.4. □

7.3.4 SCHEDULABILITY ANALYSIS

The most important property of the Priority Inheritance Protocol for real-time systems is that it bounds the maximum blocking time of each task. This allows to perform a feasibility analysis and extend the Rate-Monotonic schedulability test for sets of tasks with resource constraints. We recall that, in the absence of blocking, a set of independent periodic tasks is schedulable by the Rate-Monotonic algorithm if

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{1/n} - 1). \tag{7.1}$$

In order to perform a worst-case analysis, let B_i be the maximum blocking time, due to lower-priority jobs, that a job J_i may experience.

Theorem 7.2 *A set of n periodic tasks using the Priority Inheritance Protocol can be scheduled by the Rate-Monotonic algorithm if*

$$\forall i, \ 1 \leq i \leq n, \quad \sum_{k=1}^{i} \frac{C_k}{T_k} + \frac{B_i}{T_i} \leq i(2^{1/i} - 1). \tag{7.2}$$

Proof. Suppose that for each task τ_i equation (7.2) is satisfied. Then equation (7.1) is also satisfied with $n = i$ and C_i replaced by $C_i^* = (C_i + B_i)$. This means that, in the absence of blocking, any job of task τ_i will still meet its deadline even if it executes for $(C_i + B_i)$ units of time. It follows that task τ_i, if it executes for only C_i units of time, can be delayed by B_i and still meet its deadline. Hence, the theorem follows. □

In other words, the schedulability test expressed in equation (7.2) can be interpreted as follows. In order to guarantee a task τ_i, we have to consider the effect of preemptions from all higher-priority tasks ($\sum_{k=1}^{i-1} C_k/T_k$), the execution of τ_i itself (C_i/T_i), and the effect of blocking due to all lower-priority tasks (B_i/T_i).

Suppose, for example, that we want to guarantee the following task set:

	C_i	T_i	B_i
J_1	1	2	1
J_2	1	4	1
J_3	2	8	0

Since the periods of these tasks are harmonic, the utilization bound for Rate Monotonic becomes 100%. Hence, we have to verify the following relations:

$$
\begin{aligned}
\frac{C_1}{T_1} + \frac{B_1}{T_1} &\leq 1 \\
\frac{C_1}{T_1} + \frac{C_2}{T_2} + \frac{B_2}{T_2} &\leq 1 \\
\frac{C_1}{T_1} + \frac{C_2}{T_2} + \frac{C_3}{T_3} &\leq 1.
\end{aligned}
$$

Since all three equations hold, we can conclude that this task set is feasible and all tasks will meet their deadlines. Notice that, if the kth equation should not be satisfied, we would know that task τ_k would miss its deadline. In this case, we could correct the implementation of this task to achieve a feasible schedule.

A simpler but less tight schedulability test can be found by observing that

$$
\frac{B_i}{T_i} \leq \max\left(\frac{B_1}{T_1}, \ldots, \frac{B_n}{T_n}\right) \qquad \text{and} \qquad n(2^{1/n} - 1) \leq i(2^{1/i} - 1).
$$

As a consequence, the feasibility of the schedule can be guaranteed if the following single equation holds:

$$
\sum_{i=1}^{n} \frac{C_i}{T_i} + \max\left(\frac{B_1}{T_1}, \ldots, \frac{B_n}{T_n}\right) \leq n(2^{1/n} - 1). \tag{7.3}
$$

The schedulability test based on tasks' response times can also be extended to take resources into account. In this case, the blocking factor B_i must simply be added to the computation time of each task. Thus, the recurrent equation (4.17) for calculating the response time R_i becomes

$$
R_i = C_i + B_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil C_j. \tag{7.4}
$$

Notice that, when introducing resource constraints, this test becomes only sufficient, since tasks characterized by a long maximum blocking time could actually never experience blocking.

7.3.5 BLOCKING TIME COMPUTATION

The evaluation of the maximum blocking time for each task can be computed based on the result of Theorem 7.1. However, a precise evaluation of the blocking factor B_i is quite complex because each critical section of the lower-priority tasks may interfere with J_i via direct blocking, push-through blocking or transitive inheritance. In this section, we present a simplified algorithm that can be used to compute the blocking factors of tasks that do not use nested critical sections. In this case, in fact, Lemma 7.2 guarantees that no transitive inheritance can occur; thus, the analysis of all possible blocking conditions is simplified. The following notation is used to describe the algorithm:

- σ_i indicates the set of semaphores requested by J_i.
- $\beta_{i,j}$ indicates the set of all critical sections of the lower-priority job J_j that can block J_i.
- $\gamma_{i,k}$ indicates the set of all critical sections guarded by semaphore S_k that can block J_i.
- $Z_{i,k}$ denotes the longest critical section of task τ_i among those guarded by semaphore S_k.
- $D_{i,k}$ denotes the duration of $Z_{i,k}$.

Assuming that all durations $D_{i,k}$ are known (they can be estimated through code analysis), the algorithm for computing the blocking factor B_i of a job J_i can be logically divided into the following steps:

1. For each job J_j with priority lower than P_i, identify the set $\beta_{i,j}$ of all critical sections that can block J_i.
2. For each semaphore S_k, identify the set $\gamma_{i,k}$ of all critical sections guarded by S_k that can block J_i.
3. Sum the duration of the longest critical sections in each $\beta_{i,j}$, for any job J_j with priority lower than P_i; let B_i^l be this sum.
4. Sum the duration of the longest critical sections in each $\gamma_{i,k}$, for any semaphore S_k; let B_i^s be this sum.
5. Compute B_i as the minimum between B_i^l and B_i^s.

The identification of the critical sections that can block a task can be greatly simplified if for each semaphore S_k we define a *ceiling* $C(S_k)$ to be the priority of the highest-priority task that may use it:

$$C(S_k) = \max(P_j : S_k \in \sigma_j).$$

Then, the following lemma holds.

Lemma 7.5 *In the absence of nested critical sections, a critical section $Z_{j,k}$ of J_j guarded by S_k can block J_i only if $P_j < P_i \leq C(S_k)$.*

Proof. If $P_i \leq P_j$, then job J_i cannot preempt J_j; hence, it cannot be blocked by J_j directly. On the other hand, if $C(S_k) < P_i$, by definition of $C(S_k)$, any job that uses S_k cannot have or inherit a priority equal to or higher than P_i. Hence, from Lemma 7.1, $Z_{j,k}$ cannot cause push-through blocking on J_i. Finally, since there are no nested critical sections, Lemma 7.2 guarantees that $Z_{j,k}$ cannot cause transitive blocking. The lemma follows. $\square$

Using the result of Lemma 7.5, the maximum blocking time B_i for each task τ_i can easily be determined as follows:

$$B_i = \min(B_i^l, B_i^s), \tag{7.5}$$

where

$$\begin{aligned} B_i^l &= \sum_{j=i+1}^{n} \max_k [D_{j,k} : C(S_k) \geq P_i] \\ B_i^s &= \sum_{k=1}^{m} \max_{j>i} [D_{j,k} : C(S_k) \geq P_i]. \end{aligned}$$

This computation is performed by the algorithm shown in Figure 7.9. This algorithm assumes that the task set consists of n periodic tasks that use m distinct binary semaphores. Tasks are ordered with decreasing priority, such that $P_i > P_j$ for all $i < j$. Critical sections are nonnested. Notice that the blocking factor B_n is always zero, since there are no tasks with priority lower than P_n that can block τ_n. The complexity of the algorithm is $O(mn^2)$.

```
Blocking_Time(D_{i,k}) {

    for i = 1 to n − 1 {                                  /* for each task J_i */

        B_i^l := 0;
        for j = i + 1 to n {                              /* for each J_j : P_j < P_i */
            D_max := 0;
            for k = 1 to m {                              /* for all semaphores */
                if (C(S_k) ≥ P_i) and (D_{j,k} > D_max) {
                    D_max = D_{j,k};
                }
            }
            B_i^l := B_i^l + D_max;
        }

        B_i^s := 0;
        for k = 1 to m {                                  /* for all semaphores */
            D_max := 0;
            for j = i + 1 to n {                          /* for each J_j : P_j < P_i */
                if (C(S_k) ≥ P_i) and (D_{j,k} > D_max) {
                    D_max = D_{j,k};
                }
            }
            B_i^s := B_i^s + D_max;
        }

        B_i := min(B_i^l, B_i^s);

    }
    B_n := 0;
}
```

Figure 7.9 Algorithm for computing the blocking factors.

This algorithm provides an upper bound for the blocking factors B_i; however, such a bound is not tight, since B_i^l may be computed by considering two or more critical sections guarded by the same semaphore. Obviously, if two critical sections of different jobs are guarded by the same semaphore, they cannot be both blocking (see Lemma 7.4). Similarly, B_i^s may be computed by considering two or more critical sections belonging to the same job. But this cannot happen (see Lemma 7.3). In order to exclude these cases, however, the complexity grows exponentially because the maximum blocking time has to be computed among all possible combinations of blocking critical sections. An algorithm based on exhaustive search is presented in [Raj91]. It can find better bounds than those found by the algorithm presented in this section, but it has an exponential complexity.

EXAMPLE

To illustrate the algorithm presented above, consider the following example, in which four tasks share three semaphores. For each job J_i, the duration of the longest critical section among those that use the same semaphore S_k is denoted by $D_{i,k}$ and it is stored in a table. $D_{i,k} = 0$ means that job J_i does not use semaphore S_k. Suppose to have the following table (semaphore ceilings are indicated in parentheses):

	$S_1(P_1)$	$S_2(P_1)$	$S_3(P_2)$
J_1	1	2	0
J_2	0	9	3
J_3	8	7	0
J_4	6	5	4

According to the algorithm shown in Figure 7.9, the blocking factors of the tasks are computed as follows:

$$B_1^l = 9 + 8 + 6 = 23$$
$$B_1^s = 8 + 9 = 17 \qquad ==> \; B_1 = 17$$

$$B_2^l = 8 + 6 = 14$$
$$B_2^s = 8 + 7 + 4 = 19 \qquad ==> \; B_2 = 14$$

$$B_3^l = 6$$
$$B_3^s = 6 + 5 + 4 = 15 \qquad ==> \; B_3 = 6$$

$$B_4^l = B_4^s = 0 \qquad ==> \; B_4 = 0$$

Note that B_2^l is computed by adding the duration of two critical sections both guarded by semaphore S_1.

7.3.6 IMPLEMENTATION CONSIDERATIONS

The implementation of the Priority Inheritance Protocol requires a slight modification of the kernel data structures associated with tasks and semaphores. First of all, each task must have a nominal priority and an active priority, which need to be stored in the Task Control Block (TCB). Moreover, in order to speed up the inheritance mechanism, it is convenient that each semaphore keeps track of the task holding the lock on it. This can be done by adding in the semaphore data structure a specific field, say *holder*, for storing the identifier of the holder. In this way, a task that is blocked on a semaphore can immediately identify the task that holds its lock for transmitting its priority. Similarly, transitive inheritance can be simplified if each task keeps track of the semaphore on which it is blocked. In this case, this information has to be stored in a field, say *lock*, of the Task Control Block. Assuming that the kernel data structures are extended as described above, the primitives *pi_wait* and *pi_signal* for realizing the Priority Inheritance Protocol can be defined as follows.

pi_wait(s)

- If semaphore s is free, it becomes locked and the name of the executing task is stored in the *holder* field of the semaphore data structure.
- If semaphore s is locked, the executing task is blocked on the s semaphore queue, the semaphore identifier is stored in the *lock* field of the TCB, and its priority is inherited by the task that holds s. If such a task is blocked on another semaphore, the transitivity rule is applied. Then, the ready task with the highest priority is assigned to the processor.

pi_signal(s)

- If the queue of semaphore s is empty (that is, no tasks are blocked on s), s is unlocked.
- If the queue of semaphore s is not empty, the highest-priority task in the queue is awakened, its identifier is stored in $s.holder$, the active priority of the executing task is updated and the ready task with the highest priority is assigned to the processor.

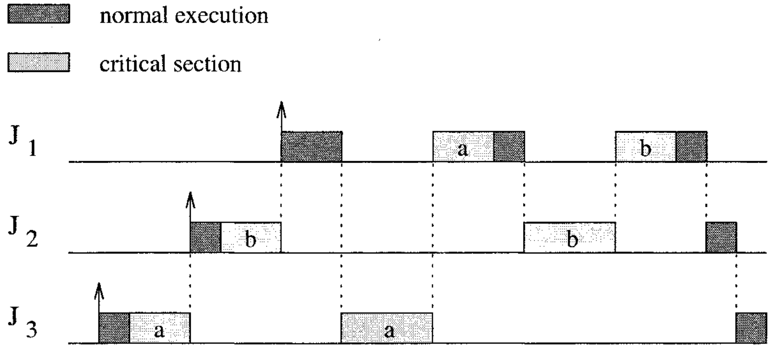

Figure 7.10 Example of chained blocking.

7.3.7 UNSOLVED PROBLEMS

Although the Priority Inheritance Protocol bounds the priority inversion phenomenon, the blocking duration for a job can still be substantial because a chain of blocking can be formed. Another problem is that the protocol does not prevent deadlocks.

CHAINED BLOCKING

Consider three jobs J_1, J_2 and J_3 with decreasing priorities that share two semaphores S_a and S_b. Suppose that J_1 needs to sequentially access S_a and S_b, J_2 accesses S_b, and J_3 S_a. Also suppose that J_3 locks S_a and it is preempted by J_2 within its critical section. Similarly, J_2 locks S_b and it is preempted by J_1 within its critical section. The example is shown in Figure 7.10. In this situation, when attempting to use its resources, J_1 is blocked for the duration of two critical sections, once to wait J_3 to release S_a and then to wait J_2 to release S_b. This is called a *chained blocking*. In the worst case, if J_1 accesses n distinct semaphores that have been locked by n lower-priority jobs, J_1 will be blocked for the duration of n critical sections.

DEADLOCK

Consider two jobs that use two semaphores in a nested fashion but in reverse order, as illustrated in Figure 7.11. Now suppose that, at time t_1, J_2 locks semaphore S_b and enters its critical section. At time t_2, J_1 preempts J_2 before it can lock S_a. At time t_3, J_1 locks S_a, which is free, but then is blocked on S_b at time t_4. At this time, J_2 resumes and continues the execution at the priority of J_1. Priority inheritance does not prevent a deadlock, which occurs at time t_5, when J_2 attempts to lock S_a. Notice,

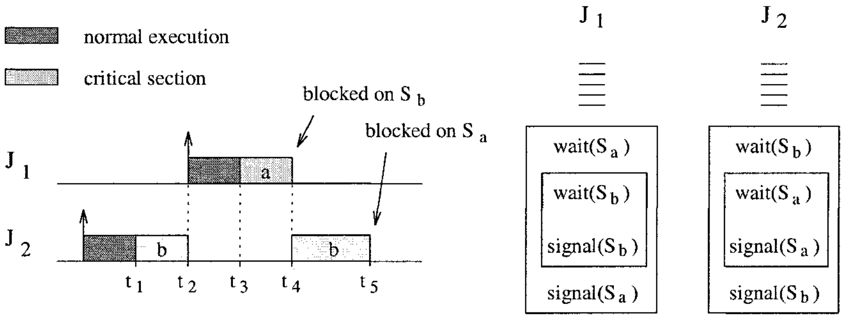

Figure 7.11 Example of deadlock.

however, that the deadlock does not depend on the Priority Inheritance Protocol but is caused by an erroneous use of semaphores. In this case, the deadlock problem can be solved by imposing a total ordering on the semaphore accesses.

7.4 PRIORITY CEILING PROTOCOL

The Priority Ceiling Protocol (PCP) has been introduced by Sha, Rajkumar, and Lehoczky [SRL90] to bound the priority inversion phenomenon and prevent the formation of deadlocks and chained blocking.

The basic idea of this method is to extend the Priority Inheritance Protocol with a rule for granting a lock request on a free semaphore. To avoid multiple blocking, this rule does not allow a job to enter a critical section if there are locked semaphores that could block it. This means that, once a job enters its first critical section, it can never be blocked by lower-priority jobs until its completion.

In order to realize this idea, each semaphore is assigned a *priority ceiling* equal to the priority of the highest-priority job that can lock it. Then, a job J is allowed to enter a critical section only if its priority is higher than all priority ceilings of the semaphores currently locked by jobs other than J.

7.4.1 PROTOCOL DEFINITION

The Priority Ceiling Protocol can be defined as follows:

- Each semaphore S_k is assigned a priority ceiling $C(S_k)$ equal to the priority of the highest-priority job that can lock it. Note that $C(S_k)$ is a static value that can be computed off-line.
- Let J_i be the job with the highest priority among all jobs ready to run; thus, J_i is assigned the processor.
- Let S^* be the semaphore with the highest priority ceiling among all the semaphores currently locked by jobs other than J_i and let $C(S^*)$ be its ceiling.
- To enter a critical section guarded by a semaphore S_k, J_i must have a priority higher than $C(S^*)$. If $P_i \leq C(S^*)$, the lock on S_k is denied and J_i is said to be blocked on semaphore S^* by the job that holds the lock on S^*.
- When a job J_i is blocked on a semaphore, it transmits its priority to the job, say J_k, that holds that semaphore. Hence, J_k resumes and executes the rest of its critical section with the priority of J_i. J_k is said to *inherit* the priority of J_i. In general, a task inherits the highest priority of the jobs blocked by it.
- When J_k exits a critical section, it unlocks the semaphore and the highest-priority job, if any, blocked on that semaphore is awakened. Moreover, the active priority of J_k is updated as follows: if no other jobs are blocked by J_k, p_k is set to the nominal priority P_k; otherwise, it is set to the highest priority of the jobs blocked by J_k.
- Priority inheritance is transitive; that is, if a job J_3 blocks a job J_2, and J_2 blocks a job J_1, then J_3 inherits the priority of J_1 via J_2.

EXAMPLE

In order to illustrate the Priority Ceiling Protocol, consider three jobs J_0, J_1, and J_2 having decreasing priorities. The highest-priority job J_0 sequentially accesses two critical sections guarded by semaphores S_0 and S_1; job J_1 accesses only a critical section guarded by semaphore S_2; whereas job J_2 uses semaphore S_2 and then makes a nested access to S_1. From tasks' resource requirements, all semaphores are assigned the following priority ceilings:

$$\begin{cases} C(S_0) = P_0 \\ C(S_1) = P_0 \\ C(S_2) = P_1. \end{cases}$$

Now suppose that events evolve as illustrated in Figure 7.12.

- At time t_0, J_2 is activated and, since it is the only job ready to run, it starts executing and later locks semaphore S_2.
- At time t_1, J_1 becomes ready and preempts J_2.
- At time t_2, J_1 attempts to lock S_2, but it is blocked by the protocol because P_1 is not greater than $C(S_2)$. Then, J_2 inherits the priority of J_1 and resumes its execution.
- At time t_3, J_2 successfully enters its nested critical section by locking S_1. Note that J_2 is allowed to lock S_1 because no semaphores are locked by other jobs.
- At time t_4, while J_2 is executing at a priority $p_2 = P_1$, J_0 becomes ready and preempts J_2 because $P_0 > p_2$.
- At time t_5, J_0 attempts to lock S_0, which is not locked by any job. However, J_0 is blocked by the protocol because its priority is not higher than $C(S_1)$, which is the highest ceiling among all semaphores currently locked by the other jobs. Since S_1 is locked by J_2, J_2 inherits the priority of J_0 and resumes its execution.
- At time t_6, J_2 exits its nested critical section, unlocks S_1, and, since J_0 is awakened, J_2 returns to priority $p_2 = P_1$. At this point, $P_0 > C(S_2)$; hence, J_0 preempts J_2 and executes until completion.
- At time t_7, J_0 is completed, and J_2 resumes its execution at a priority $p_2 = P_1$.
- At time t_8, J_2 exits its outer critical section, unlocks S_2, and, since J_1 is awakened, J_2 returns to its nominal priority P_2. At this point, J_1 preempts J_2 and executes until completion.
- At time t_9, J_1 is completed; thus, J_2 resumes its execution.

Notice that the Priority Ceiling Protocol introduces a third form of blocking, called *ceiling blocking*, in addition to direct blocking and push-through blocking caused by the Priority Inheritance Protocol. This is necessary for avoiding deadlock and chained blocking. In the previous example, a ceiling blocking is experienced by job J_0 at time t_5.

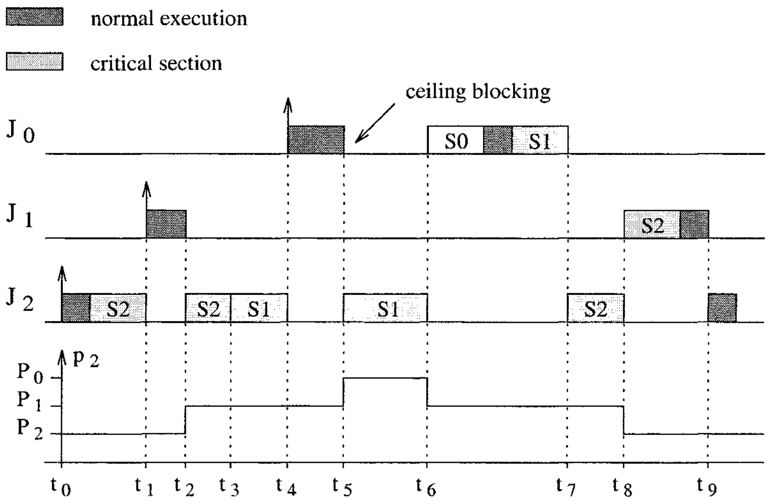

Figure 7.12 Example of Priority Ceiling Protocol.

7.4.2 PROPERTIES OF THE PROTOCOL

The main properties of the Priority Ceiling Protocol are presented in this section. They are used to analyze the schedulability and compute the maximum blocking time of each task.

Lemma 7.6 *If a job J_k is preempted within a critical section Z_a by a job J_i that enters a critical section Z_b, then, under the Priority Ceiling Protocol, J_k cannot inherit a priority higher than or equal to that of job J_i until J_i completes.*

Proof. If J_k inherits a priority higher than or equal to that of job J_i before J_i completes, there must exist a job J_0 blocked by J_k, such that $P_0 \geq P_i$. This situation is shown in Figure 7.13. However, this leads to the contradiction that J_0 cannot be blocked by J_k. In fact, since J_i enters its critical section, its priority must be higher than the maximum ceiling C^* of the semaphores currently locked by all lower-priority jobs. Hence, $P_0 \geq P_i > C^*$. But since $P_0 > C^*$, J_0 cannot be blocked by J_k, and the lemma follows. □

Lemma 7.7 *The Priority Ceiling Protocol prevents transitive blocking.*

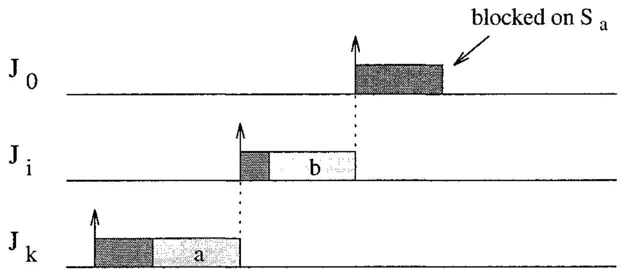

Figure 7.13 An absurd situation that cannot occur under the Priority Ceiling Protocol.

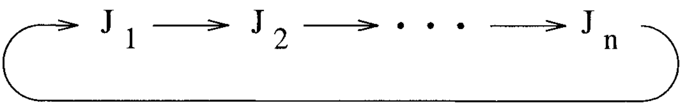

Figure 7.14 Deadlock among n jobs.

Proof. Suppose that a transitive block occurs; that is, there exist three jobs J_1, J_2, and J_3, with decreasing priorities, such that J_3 blocks J_2 and J_2 blocks J_1. By the transitivity of the protocol, J_3 will inherit the priority of J_1. However, this contradicts Lemma 7.6, which shows that J_3 cannot inherit a priority higher than or equal to P_2. Thus, the lemma follows. □

Theorem 7.3 *The Priority Ceiling Protocol prevents deadlocks.*

Proof. Assuming that a job cannot deadlock by itself, a deadlock can only be formed by a cycle of jobs waiting for each other, as shown in Figure 7.14. In this situation, however, by the transitivity of the protocol, job J_n would inherit the priority of J_1, which is assumed to be higher than P_n. This contradicts Lemma 7.6, and hence the theorem follows. □

Theorem 7.4 (Sha-Rajkumar-Lehoczky) *Under the Priority Ceiling Protocol, a job J_i can be blocked for at most the duration of one critical section.*

Proof. Suppose that J_i is blocked by two lower-priority jobs J_1 and J_2, where $P_2 < P_1 < P_i$. Let J_2 enter its blocking critical section first, and let C_2^* be the highest-priority ceiling among all the semaphores locked by J_2. In this situation, if job J_1 enters its critical section we must have that $P_1 > C_2^*$. Moreover, since we assumed that J_i can be blocked by J_2, we must have that $P_i \leq C_2^*$. This means that $P_1 > C_2^* \geq P_i$. This contradicts the assumption that $P_i > P_2$. Thus, the theorem follows. □

7.4.3 SCHEDULABILITY ANALYSIS

The feasibility test for a set of periodic tasks using the Priority Ceiling Protocol can be performed by the same formulae shown for the Priority Inheritance Protocol. The only difference is in the values of each blocking factor B_i, which, for the Priority Ceiling Protocol, corresponds to the duration of the longest critical section among those that can block τ_i.

7.4.4 BLOCKING TIME COMPUTATION

The evaluation of the maximum blocking time for each task can be computed based on the result of Theorem 7.4. According to this theorem, a job J_i can be blocked for at most the duration of the longest critical section among those that can block J_i. The set of critical sections that can block a job J_i is identified by the following lemma.

Lemma 7.8 *Under the Priority Ceiling Protocol, a critical section $Z_{j,k}$ (belonging to job J_j and guarded by semaphore S_k) can block a job J_i only if $P_j < P_i$ and $C(S_k) \geq P_i$.*

Proof. Clearly, if $P_j \geq P_i$, J_i cannot preempt J_j and hence cannot be blocked on $Z_{j,k}$. Now assume $P_j < P_i$ and $C(S_k) < P_i$, and suppose that J_i is blocked on $Z_{j,k}$. We show that this assumption leads to a contradiction. In fact, if J_i is blocked by J_j, its priority must be less than or equal to the maximum ceiling C^* among all semaphores locked by jobs other than J_i. Thus, we have that $C(S_k) < P_i \leq C^*$. On the other hand, since C^* is the maximum ceiling among all semaphores currently locked by jobs other than J_i, we have that $C^* \geq C(S_k)$, which leads to a contradiction and proves the lemma. □

Using the result of Lemma 7.8, the maximum blocking time B_i of job J_i can be computed as the duration of the longest critical section among those belonging to tasks with priority lower than P_i and guarded by a semaphore with ceiling higher than or equal to P_i. If $D_{j,k}$ denotes the duration of the longest critical section of task τ_j among those guarded by semaphore S_k, we can write

$$B_i = \max_{j,k}\{D_{j,k} \mid P_j < P_i,\ C(S_k) \geq P_i\}. \tag{7.6}$$

Consider the same example illustrated for the Priority Inheritance Protocol. For each job J_i, the duration of the longest critical section among those guarded by semaphore S_k is denoted by $D_{i,k}$ and it is stored in a table. $D_{i,k} = 0$ means that job J_i does not use semaphore S_k. Semaphore ceilings are indicated in parentheses:

	$S_1(P_1)$	$S_2(P_1)$	$S_3(P_2)$
J_1	1	2	0
J_2	0	9	3
J_3	8	7	0
J_4	6	5	4

According to equation (7.6), tasks' blocking factors are computed as follows:

$$\begin{cases} B_1 = \max(8,6,9,7,5) = 9 \\ B_2 = \max(8,6,7,5,4) = 8 \\ B_3 = \max(6,5,4) = 6 \\ B_4 = 0. \end{cases}$$

7.4.5 IMPLEMENTATION CONSIDERATIONS

The major implication of the Priority Ceiling Protocol in the kernel data structures is that semaphores queues are no longer needed, since the tasks blocked by the protocol can be kept in the ready queue. In particular, whenever a job J_i is blocked by the protocol on a semaphore S_k, the job J_h that holds S_k inherits the priority of J_i and it is assigned to the processor, whereas J_i returns to the ready queue. As soon as J_h unlocks S_k, p_h is updated and, if p_h becomes less than the priority of the first ready job, a context switch is performed.

To implement the Priority Ceiling Protocol, each semaphore S_k has to store the identifier of the task that holds the lock on S_k and the ceiling of S_k. Moreover, an additional field for storing the task active priority has to be reserved in the task control block. It is also convenient to have a field in the task control block for storing the identifier of the semaphore on which the task is blocked. Finally, the implementation of the protocol can be simplified if the system also maintains a list of currently locked semaphores, order by decreasing priority ceilings. This list is useful for computing the maximum priority ceiling that a job has to overcome to enter a critical section and for updating the active priority of tasks at the end of a critical section.

If the kernel data structures are extended as described above, the primitives *pc_wait* and *pc_signal* for realizing the Priority Ceiling Protocol can be defined as follows.

pc_wait(s)

- Find the semaphore S^* having the maximum ceiling C^* among all the semaphores currently locked by jobs other than the one in execution (J_{exe}).
- If $p_{exe} \leq C^*$, transfer P_{exe} to the job that holds S^*, insert J_{exe} in the ready queue, and execute the ready job (other than J_{exe}) with the highest priority.
- If $p_{exe} > C^*$, or whenever s is unlocked, lock semaphore s, add s in the list of currently locked semaphores and store J_{exe} in $s.holder$.

pc_signal(s)

- Extract s from the list of currently locked semaphores.
- If no other jobs are blocked by J_{exe}, set $p_{exe} = P_{exe}$, else set p_{exe} to the highest priority of the jobs blocked by J_{exe}.
- Let p^* be the highest priority among the ready jobs. If $p_{exe} < p^*$, insert J_{exe} in the ready queue and execute the ready job (other than J_{exe}) with the highest priority.

7.5 STACK RESOURCE POLICY

The Stack Resource Policy (SRP) is a technique proposed by Baker [Bak91] for accessing shared resources. It extends the Priority Ceiling Protocol (PCP) in three essential points:

1. It allows the use of multi-unit resources.
2. It supports dynamic priority scheduling.
3. It allows the sharing of runtime stack-based resources.

From a scheduling point of view, the essential difference between the PCP and the SRP is on the time at which a task is blocked. Whereas under the PCP a task is blocked at the time it makes its first resource request, under the SRP a task is blocked at the time it attempts to preempt. This early blocking slightly reduces concurrency but saves unnecessary context switches, simplifies the implementation of the protocol, and allows the sharing of runtime stack resources.

7.5.1 DEFINITIONS

Before presenting the formal description of the SRP we introduce the following definitions.

PRIORITY

Each task τ_i is assigned a priority p_i that indicates the importance (that is, the urgency) of τ_i with respect to the other tasks in the system. Priorities can be assigned to tasks either statically or dynamically. At any time t, $p_a > p_b$ means that the execution of τ_a is more important than that of τ_b; hence, τ_b can be delayed in favor of τ_a. For example, priorities can be assigned to tasks based on Rate Monotonic (RM) or Earliest Deadline First (EDF).

PREEMPTION LEVEL

Besides a priority p_i, a task τ_i is also characterized by a *preemption level* π_i. The preemption level is a static parameter, assigned to a task at its creation time and associated with all instances of that task. The essential property of preemption levels is that a job J_a can preempt another job J_b only if $\pi_a > \pi_b$. This is also true for priorities. Hence, the reason for distinguishing preemption levels from priorities is that preemption levels are fixed values that can be used to predict potential blocking also in the presence of dynamic priority schemes. The general definition of preemption level used to prove all properties of the SRP requires that

> if J_a arrives after J_b and J_a has higher priority than J_b, then J_a must have a higher preemption level than J_b.

Under EDF scheduling, the previous condition is satisfied if preemption levels are ordered inversely with respect to the order of relative deadlines; that is,

$$\pi_i > \pi_j \Longleftrightarrow D_i < D_j.$$

To better illustrate the difference between priorities and preemption levels, consider the example shown in Figure 7.15. Here we have two jobs J_1 and J_2, with relative deadlines $D_1 = 10$ and $D_2 = 5$, respectively. Being $D_2 < D_1$, we define $\pi_1 = 1$ and $\pi_2 = 2$. Since $\pi_1 < \pi_2$, J_1 can never preempt J_2; however, J_1 may have a priority higher than that of J_2. In fact, under EDF, the priority of a job is dynamically assigned based on its absolute deadline. For example, in the case illustrated in Figure 7.15a, the absolute deadlines are such that $d_2 < d_1$; hence, J_2 will have higher priority than J_1. On the other hand, as shown in Figure 7.15b, if J_2 arrives a time $r_1 + 6$, absolute deadlines are such that $d_2 > d_1$; hence, J_1 will have higher priority than J_2.

Notice that, in the case of Figure 7.15b, although J_1 has priority higher than J_2, J_2 cannot be preempted. This happens because, when $d_1 < d_2$ and $D_1 > D_2$, J_1 always starts before J_2; thus, it does not need to preempt J_2.

RESOURCE CEILING

Each resource R is required to have a *current ceiling* C_R, which is a dynamic value computed as a function of the units of R that are currently available. If n_R denotes the number of units of R that are currently available and $\mu_R(J)$ denotes the maximum requirement of job J for R, the current ceiling of R is defined to be

$$C_R(n_R) = \max[\{0\} \cup \{\pi(J) : n_R < \mu_R(J)\}].$$

In other words, if all units of R are available, then $C_R = 0$. However, if the units of R that are currently available cannot satisfy the requirement of one or more jobs, then C_R is equal to the highest preemption level of those jobs that could be blocked on R.

To better clarify this concept, consider the following example, where three tasks (J_1, J_2, J_3) share three resources (R_1, R_2, R_3), consisting of three, one, and three units,

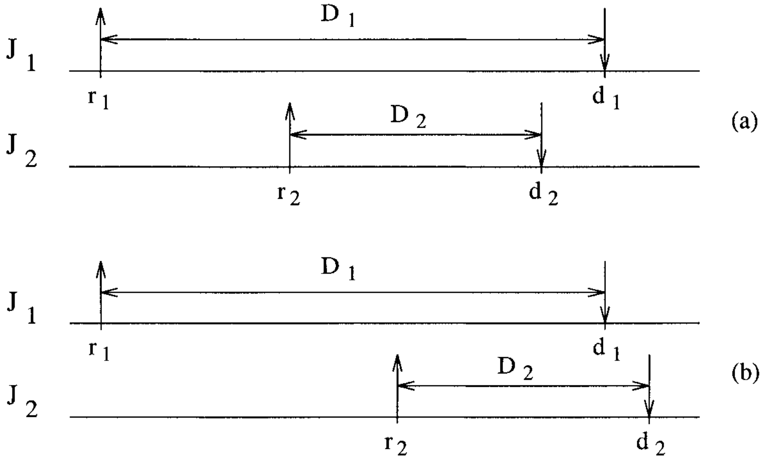

Figure 7.15 Although $\pi_2 > \pi_1$, under EDF p_2 can be higher than p_1 (a) or lower than p_1 (b).

	D_i	π_i	μ_{R1}	μ_{R2}	μ_{R3}
J_1	5	3	1	0	1
J_2	10	2	2	1	3
J_3	20	1	3	1	1

Figure 7.16 Task parameters and resource requirements.

	$C_R(3)$	$C_R(2)$	$C_R(1)$	$C_R(0)$
R_1	0	1	2	3
R_2	-	-	0	2
R_3	0	2	2	3

Figure 7.17 Resource ceilings as a function of the number of available units. Dashes identify impossible cases.

respectively. All tasks parameters – relative deadlines, preemption levels, and resource requirements – are shown in Figure 7.16.

Based on these requirements, the current ceilings of the resources as a function of the number n_R of available units are reported in Figure 7.17 (dashes identify impossible cases).

Let us compute, for example, the ceiling of resource R_1 when only two units (out of three) are available. From Figure 7.16, we see that the only job that could be blocked in this condition is J_3 because it requires three units of R_1; hence, $C_{R1}(2) = \pi_3 = 1$. If only one unit of R_1 is available, the jobs that could be blocked are J_2 and J_3; hence, $C_{R1}(1) = \max(\pi_2, \pi_3) = 2$. Finally, if none of the units of R_1 is available, all three jobs could be blocked on R_1; hence, $C_{R1}(0) = \max(\pi_1, \pi_2, \pi_3) = 3$.

Notice that, in the specific case of resources having a single unit (binary resources), the definition of current ceiling can be simplified as follows:

$$C_R = \max(\{0\} \cup \{\pi(J) : R \textit{ could block } J\}).$$

This means that, if R is free, its ceiling is zero, whereas if R is busy, its ceiling is equal to the highest preemption level of the jobs that require R.

SYSTEM CEILING

The resource access protocol adopted in the SRP also requires a *system ceiling*, Π_s, defined as the maximum of the current ceilings of all the resources; that is,

$$\Pi_s = \max(C_{R_i} : i = 1, \ldots, m).$$

Notice that Π_s is a dynamic parameter that can change every time a resource is accessed or released by a job.

7.5.2 PROTOCOL DEFINITION

The key idea of the SRP is that, when a job needs a resource that is not available, it is blocked at the time it attempts to preempt, rather than later. Moreover, to prevent multiple priority inversions, a job is not allowed to start until the resources currently available are sufficient to meet the maximum requirement of every job that could preempt it. Using the definitions introduced in the previous paragraph, this is achieved by the following preemption test:

> A job is not permitted to preempt until its priority is the highest among those of all the jobs ready to run, and its preemption level is higher than the system ceiling.

If the ready queue is ordered by decreasing priorities, the preemption test can be simply performed by comparing the preemption level $\pi(J)$ of the ready job with the highest priority (the one at the head of the queue) with the system ceiling. If $\pi(J) > \Pi_s$, job J is executed, otherwise it is kept in the ready queue until Π_s becomes less than $\pi(J)$. The condition $\pi(J) > \Pi_s$ has to be tested every time Π_s may decrease; that is, every time a resource is released.

OBSERVATIONS

The implications that the use of the SRP has on tasks' execution can be better understood through the following observations:

- Passing the preemption test for job J ensures that the resources that are currently available are sufficient to satisfy the maximum requirement of job J and the maximum requirement of every job that could preempt J. This means that, once J starts executing, it will never be blocked for resource contention.
- Although the preemption test for a job J is performed before J starts to execute, resources are not allocated at this time but only when requested.
- A task can be blocked by the preemption test even though it does not require any resource. This is needed to avoid unbounded priority inversion.
- Blocking at preemption time, rather than at access time, decreases the number of context switches, reduces the run-time overhead, and simplifies the implementation of the protocol.
- The preemption test has the effect of imposing priority inheritance; that is, an executing job that holds a resource modifies the system ceiling and resists preemption as though it inherits the priority of any jobs that might need that resource. Note that this effect is accomplished without modifying the priority of the job.

EXAMPLE

In order to illustrate how the SRP works, consider the task set already described in Figure 7.16. The structure of the tasks is shown in Figure 7.18, where $wait(R_i, n)$ denotes the request of n units of resource R_i, and $signal(R_i)$ denotes their release. The current ceilings of the resources have already been shown in Figure 7.17, and a possible EDF schedule for this task set is depicted in Figure 7.19. In this figure, the fourth timeline reports the variation of the system ceiling, whereas the numbers along the schedule denote resource indexes.

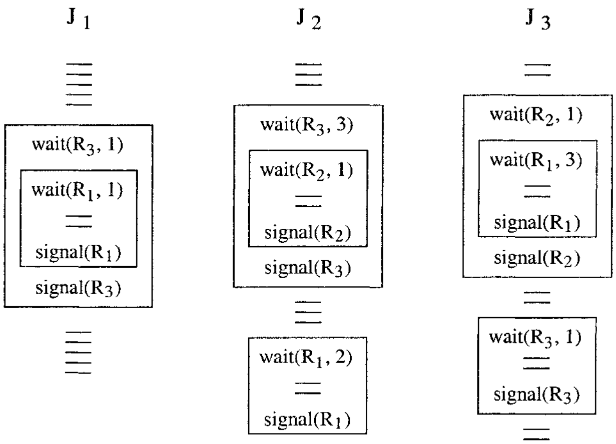

Figure 7.18 Structure of the tasks in the SRP example.

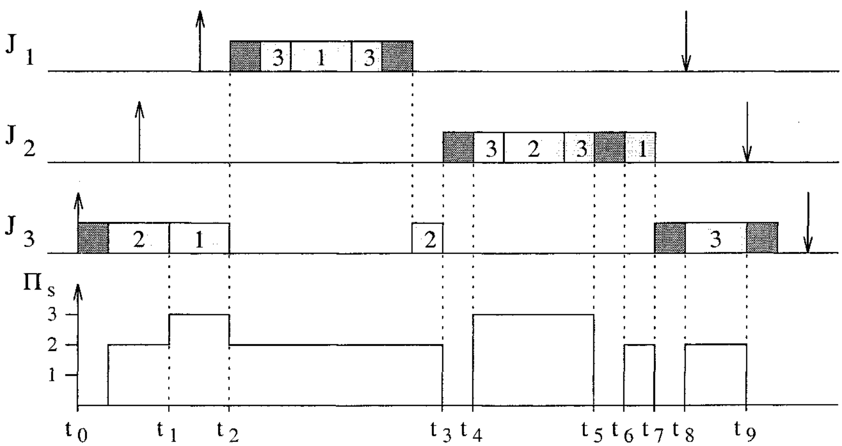

Figure 7.19 Example of a schedule under EDF and SRP. Numbers on tasks execution denote the resource indexes.

At time t_0, J_3 starts executing and the system ceiling is zero because all resources are completely available. When J_3 enters its first critical section, it takes the only unit of R_2; thus, the system ceiling is set to the highest preemption level among the tasks that could be blocked on R_2 (see Figure 7.17); that is, $\Pi_s = \pi_2 = 2$. As a consequence, J_2 is blocked by the preemption test and J_3 continues to execute. Note that when J_3 enters its nested critical section (taking all units of R_1), the system ceiling is raised to $\Pi_s = \pi_1 = 3$. This causes J_1 to be blocked by the preemption test.

As J_3 releases R_1 (at time t_2), the system ceiling becomes $\Pi_s = 2$; thus, J_1 preempts J_3 and starts executing. Note that, once J_1 is started, it is never blocked during its execution because the condition $\pi_1 > \Pi_s$ guarantees that all the resources needed by J_1 are available. As J_1 terminates, J_3 resumes the execution and releases resource R_2. As R_2 is released, the system ceiling returns to zero and J_2 can preempt J_3. Again, once J_2 is started, all the resources it needs are available; thus, J_2 is never blocked.

7.5.3 PROPERTIES OF THE PROTOCOL

The main properties of the Stack Resource Policy are presented in this section. They will be used to analyze the schedulability and compute the maximum blocking time of each task.

Lemma 7.9 *If the preemption level of a job J is greater than the current ceiling of a resource R, then there are sufficient units of R available to*

1. *Meet the maximum requirement of J and*
2. *Meet the maximum requirement of every job that can preempt J.*

Proof. Assume $\pi(J) > C_R$, but suppose that the maximum request of J for R cannot be satisfied. Then, by definition of current ceiling of a resource, we have $C_R \geq \pi(J)$, which is a contradiction.

Assume $\pi(J) > C_R$, but suppose that there exists a job J_H that can preempt J such that the maximum request of J_H for R cannot be satisfied. Since J_H can preempt J, it must be $\pi(J_H) > \pi(J)$. Moreover, since the maximum request of J_H for R cannot be satisfied, by definition of current ceiling of a resource, we have $C_R \geq \pi(J_H)$. Hence, we derive that $\pi(J) < C_R$, which contradicts the assumption. $\square$

Theorem 7.5 (Baker) *If no job J is permitted to start until $\pi(J) > \Pi_s$, then no job can be blocked after it starts.*

Proof. Let N be the number of tasks that can preempt a job J and assume that no job is permitted to start until its preemption level is greater than Π_s. The thesis will be proved by induction on N.

If $N = 0$, there are no jobs that can preempt J. If J is started when $\pi(J) > \Pi_s$, Lemma 7.9 guarantees that all the resources required by J are available when J preempts; hence, J will execute to completion without blocking.

If $N > 0$, suppose that J is preempted by J_H. If J_H is started when $\pi(J_H) > \Pi_s$, Lemma 7.9 guarantees that all the resources required by J_H are available when J_H preempts. Since any job that preempts J_H also preempts J, the induction hypothesis guarantees that J_H executes to completion without blocking, as will any job that preempts J_H, transitively. When all the jobs that preempted J complete, J can resume its execution without blocking, since the higher-priority jobs released all resources and when J started the resources available were sufficient to meet the maximum request of J. $\square$

Theorem 7.6 (Baker) *Under the Stack Resource Policy, a job J_i can be blocked for at most the duration of one critical section.*

Proof. Suppose that J_i is blocked for the duration of two critical sections shared with two lower-priority jobs, J_1 and J_2. Without loss of generality, assume $\pi_2 < \pi_1 < \pi_i$. This can happen only if J_1 and J_2 hold two different resources (such as R_1 and R_2) and J_2 is preempted by J_1 inside its critical section. This situation is depicted in Figure 7.20. This immediately yields to a contradiction. In fact, since J_1 is not blocked by the preemption test, we have $\pi_1 > \Pi_s$. On the other hand, since J_i is blocked, we have $\pi_i \leq \Pi_s$. Hence, we obtain that $\pi_i < \pi_1$, which contradicts the assumption. $\square$

Theorem 7.7 (Baker) *The Stack Resource Policy prevents deadlocks.*

Proof. By Theorem 7.5, a job cannot be blocked after it starts. Since a job cannot be blocked while holding a resource, there can be no deadlock. $\square$

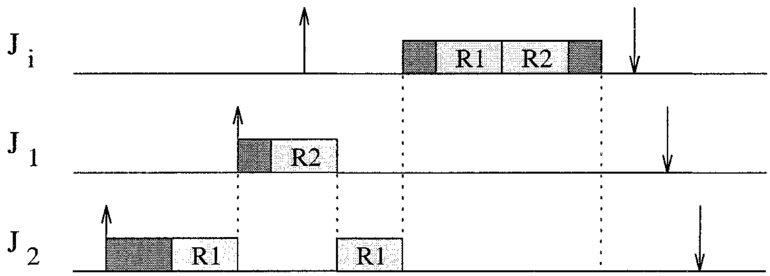

Figure 7.20 An absurd situation that cannot occur under SRP.

7.5.4 SCHEDULABILITY ANALYSIS

As far as the schedulability analysis is concerned, the considerations done for the Priority Ceiling Protocol are also valid for the Stack Resource Policy, since the general result does not depend on the time on which a job is blocked. However, if the SRP is used along with the EDF scheduling algorithm, the guarantee test has to be modified by considering that under EDF the least upper bound of the processor utilization factor is 1.

As a result, a set of n periodic tasks using the Stack Resource Policy can be scheduled by the EDF algorithm if

$$\forall i, \;\; 1 \leq i \leq n, \qquad \left(\sum_{k=1}^{i} \frac{C_k}{T_k} \right) + \frac{B_i}{T_i} \leq 1. \tag{7.7}$$

As for the PCP, C_i denotes the worst-case execution time of task τ_i, T_i denotes its period, and B_i its maximum blocking time. For each task τ_i, the sum in parentheses represents the utilization factor due to τ_i itself and to all tasks having a preemption level higher than π_i, whereas the term B_i/T_i considers the blocking time caused by tasks having preemption level lower than π_i. Condition (7.7) can easily be extended to periodic tasks with deadlines less than periods. In this case, the schedulability test is modified as follows:

$$\forall i, \;\; 1 \leq i \leq n, \qquad \left(\sum_{k=1}^{i} \frac{C_k}{D_k} \right) + \frac{B_i}{D_i} \leq 1. \tag{7.8}$$

A more precise schedulability condition can be achieved through a processor demand approach [BRH90, JS93]. In particular, equation (4.22) has been extended in [BL97, Lip97], where it is proved that a set of periodic tasks that use shared resources with

SRP is schedulable by EDF if $U < 1$ and for all $L \geq 0$ and for all $1 \leq i \leq n$

$$B_i + \sum_{k=1}^{i} \left\lfloor \frac{L + T_k - D_k}{T_k} \right\rfloor C_k \ \leq \ L. \tag{7.9}$$

It is worth noticing that this test can be run more efficiently by checking condition (7.9) on the following set of intervals:

$$\mathcal{D} = \{d_k \mid D_i \leq d_k \leq \min(L_i^*, D_n)\}$$

where

$$L_i^* \ = \ B_i + \frac{\sum_{k=1}^{i}(T_k - D_k)U_k}{1 - \sum_{k=1}^{i} U_k}.$$

7.5.5 BLOCKING TIME COMPUTATION

The maximum blocking time that a job can experience with the SRP is the same as the one that can be experienced with the Priority Ceiling Protocol. Theorem 7.6, in fact, guarantees that under the SRP a job J_i can be blocked for at most the duration of one critical section among those that can block J_i. Lemma 7.8, proved for the PCP, can be easily extended to the SRP, thus a critical section $Z_{j,k}$ belonging to job J_j and guarded by semaphore S_k can block a job J_i only if $\pi_j < \pi_i$ and $\max(C_{S_k}) \geq \pi_i$. Notice that, under the SRP, the ceiling of a semaphore is a dynamic variable, so we have to consider its maximum value, that is the one corresponding to a number of units currently available equal to zero.

Hence, the maximum blocking time B_i of job J_i can be computed as the duration of the longest critical section among those belonging to tasks with preemption level lower than π_i and guarded by a semaphore with maximum ceiling higher than or equal to π_i. If $D_{j,k}$ denotes the duration of the longest critical section of task τ_j among those guarded by semaphore S_k, we can write

$$B_i = \max_{j,k}\{D_{j,k} \mid \pi_j < \pi_i, \ C_{S_k}(0) \geq \pi_i\}. \tag{7.10}$$

7.5.6 SHARING RUNTIME STACK

Another interesting implication deriving from the use of the SRP is that it supports stack sharing among tasks. This is particularly convenient for those applications consisting of a large number of tasks, dedicated to acquisition, monitoring, and control activities. In conventional operating systems, each process must have a private stack space, sufficient

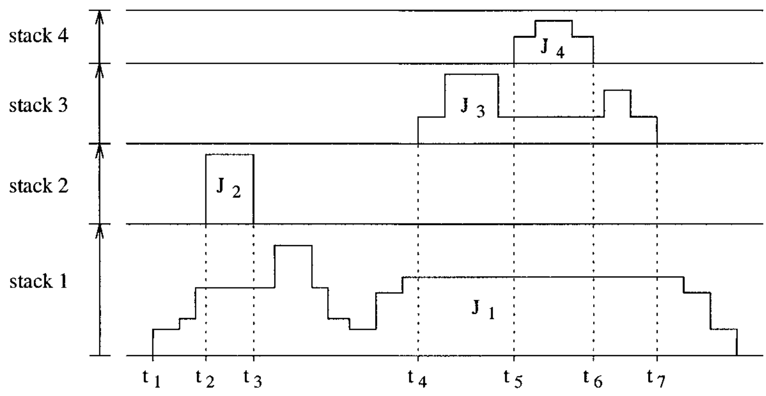

Figure 7.21 Possible evolution with one stack per task.

to store its context (that is, the content of the CPU registers) and its local variables. A problem with these systems is that, if the number of tasks is large, a great amount of memory may be required for the stacks of all the tasks.

For example, consider four jobs J_1, J_2, J_3, and J_4, with preemption levels $1, 2, 2$, and 3, respectively (3 being the highest preemption level). Figure 7.21 illustrates a possible evolution of the stacks, assuming that each job is allocated its own stack space, equal to its maximum requirement. At time t_1, J_1 starts executing; J_2 preempts at time t_2 and completes at time t_3, allowing J_1 to resume. At time t_4, J_1 is preempted by J_3, which in turn is preempted by J_4 at time t_5. At time t_6, J_4 completes and J_3 resumes. At time t_7, J_3 completes and J_1 resumes.

Note that the top of each process stack varies during the process execution, while the storage region reserved for each stack remains constant and corresponds to the distance between two horizontal lines. In this case, the total storage area that must be reserved for the application is equal to the sum of the stack regions dedicated to each process.

However, if all tasks are independent or use the SRP to access shared resources, then they can share a single stack space. In this case, when a job J is preempted by a job J', J maintains its stack and the stack of J' is allocated immediately above that of J. Figure 7.22 shows a possible evolution of the previous task set when a single stack is allocated to all tasks.

Under the SRP, stack overlapping without interpenetration is a direct consequence of Theorem 7.5. In fact, since a job J can never be blocked once started, its stack can

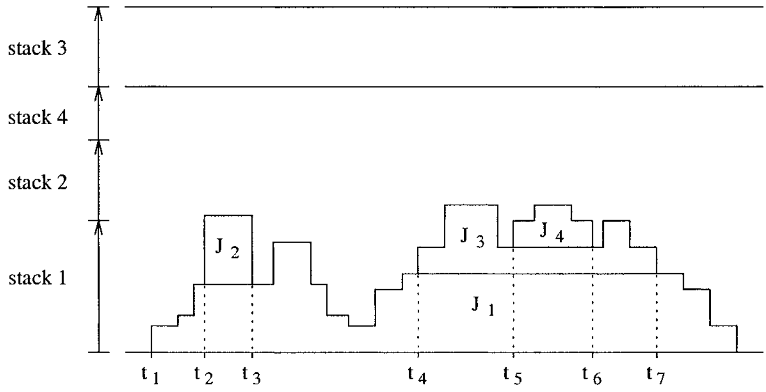

Figure 7.22 Possible evolution with a single stack for all tasks.

never be penetrated by the ones belonging to jobs with lower preemption levels, which can resume only after J is completed.

Note that the stack space between the two upper horizontal lines (which is equivalent to the minimum stack between J_2 and J_3) is no longer needed, since J_2 and J_3 have the same preemption level, so they can never occupy stack space at the same time. In general, the higher the number of tasks with the same preemption level, the larger stack saving.

For example, consider an application consisting of 100 jobs distributed on 10 preemption levels, with 10 jobs for each level, and suppose that each job needs up to 10 Kbytes of stack space. Using a stack per job, 1000 Kbytes would be required. On the contrary, using a single stack, only 100 Kbytes would be sufficient, since no more than one job per preemption level could be active at one time. Hence, in this example we would save 900 Kbytes; that is, 90%. In general, when tasks are distributed on k preemption levels, the space required for a single stack is equal to the sum of the largest request on each level.

7.5.7 IMPLEMENTATION CONSIDERATIONS

The implementation of the SRP is similar to that of the PCP, but the locking operations (*srp_wait* and *srp_signal*) are simpler, since a job can never be blocked when attempting to lock a semaphore. When there are no sufficient resources available to satisfy the maximum requirement of a job, the job is not permitted to preempt and is kept in the ready queue.

To simplify the preemption test, all the ceilings of the resources (for any number of available units) can be precomputed and stored in a table. Moreover, a stack can be used to keep track of the system ceiling. When a resource R is allocated, its current state, n_R, is updated and, if $C_R(n_R) > \Pi_s$, then Π_s is set to $C_R(n_R)$. The old values of n_R and Π_s are pushed onto the stack. When resource R is released, the values of Π_s and n_R are restored from the stack. If the restored system ceiling is lower than the previous value, the preemption test is executed on the ready job with the highest priority to check whether it can preempt. If the preemption test is passed, a context switch is performed; otherwise, the current task continues its execution.

7.6 SUMMARY

The concurrency control protocols presented in this chapter can be compared with respect to several characteristics. Figure 7.23 provides a qualitative evaluation of the algorithms in terms of priority assignment, number of blockings, instant of blocking, programming transparency, deadlock prevention, implementation, and complexity for computing the blocking factors. Notice that the Priority Inheritance Protocol (PIP), although not so effective in terms of performance, is the only one that is transparent at the programming level. The other protocols, in fact, require the user to specify the list of resources used by each task, in order to compute the ceiling values. This feature of PIP makes it actractive for commercial operating systems (like VxWorks), where predictability can be improved without introducing new kernel primitives.

	priority assignment	number of blocking	blocking instant	transp-arency	deadlock prevention	implem-entation	B_i computation
PIP	fixed	min(n,m)	on resource access	YES	NO	hard	hard
PCP	fixed	1	on resource access	NO	YES	medium	easy
SRP	fixed or dynamic	1	on preemption	NO	YES	easy	easy

Figure 7.23 Evaluation summary of resource access protocols.

Exercises

7.1 Verify whether the following task set is schedulable by the Rate-Monotonic algorithm. Apply the processor utilization approach first, and then the Response Time Analysis:

	C_i	T_i	B_i
τ_1	4	10	5
τ_2	3	15	3
τ_3	4	20	0

7.2 Consider three periodic tasks τ_1, τ_2, and τ_3 (having decreasing priority), which share three resources, A, B, and C, accessed using the Priority Inheritance Protocol. Compute the maximum blocking time B_i for each task, knowing that the longest duration $D_i(R)$ for a task τ_i on resource R is given in the following table (there are no nested critical sections):

	A	B	C
τ_1	2	0	2
τ_2	2	3	0
τ_3	3	2	5

7.3 Solve the same problem described in Exercise 7.2 when the resources are accessed by the Priority Ceiling Protocol.

7.4 For the task set described in Exercise 7.2, illustrate the situation produced by RM + PIP in which task τ_2 experiences its maximum blocking time.

7.5 Consider four periodic tasks τ_1, τ_2, τ_3, and τ_4 (having decreasing priority), which share five resources, A, B, C, D, and E, accessed using the Priority Inheritance Protocol. Compute the maximum blocking time B_i for each task, knowing that the longest duration $D_i(R)$ for a task τ_i on resource R is given in the following table (there are no nested critical sections):

	A	B	C	D	E
τ_1	2	5	9	0	6
τ_2	0	0	7	0	0
τ_3	0	3	0	7	13
τ_4	6	0	8	0	10

7.6 Solve the same problem described in Exercise 7.5 when the resources are accessed by the Priority Ceiling Protocol.

7.7 For the task set described in Exercise 7.5, illustrate the situation produced by RM + PIP in which task τ_2 experiences its maximum blocking time.

7.8 Consider three tasks τ_1, τ_2, and τ_3, which share three multi-unit resources, A, B, and C, accessed using the Stack Resource Policy. Resources A and B have three units, whereas C has two units. Compute the ceiling table for all the resources based on the following task characteristics:

	D_i	μ_A	μ_B	μ_C
τ_1	5	1	0	1
τ_2	10	2	1	2
τ_3	20	3	1	1