CHAPTER 8

Resources and Resource Access Control

In Section 3.1, we briefly mentioned the fact that in addition to a processor, each job may require some other resource in order to execute, but thus far we have ignored this requirement in order to focus on the problems in processor scheduling. We are now ready to take into account the resource requirements.

This chapter first extends our workload model to include resources and introduces additional notations needed in later sections. It then discusses how resource contention affects the execution behavior and schedulability of jobs, how various resource access-control protocols work to reduce the undesirable effect of resource contention, and how well these protocols succeed in achieving this goal. We focus on priority-driven systems. Clock-driven systems do not have these problems as we can avoid resource contention among jobs by scheduling them according to a cyclic schedule that keeps jobs' resource accesses serialized.

8.1 ASSUMPTIONS ON RESOURCES AND THEIR USAGE

We continue to focus on the case where the system contains only one processor. In addition, the system also contains ρ types of serially reusable resources named $R_1, R_2, \ldots, R_\rho$. There are ν_i indistinguishable units of resource (of type) R_i, for $1 \leq i \leq \rho$. Serially reusable resources are typically granted (i.e., allocated) to jobs on a nonpreemptive basis and used in a *mutually exclusive* manner. In other words, when a unit of a resource R_i is granted to a job, this unit is no longer available to other jobs until the job frees the unit. Again, examples of such resources are mutexes, reader/writer locks, connection sockets, printers, and remote servers. A binary semaphore is a resource (type) that has only 1 unit while a counting semaphore has many units. A system containing five printers has 5 units of the printer resource. There is only 1 unit of an exclusive write-lock.

A resource that has an infinite number of units has no effect on the timing behavior of any job since every job can have the resource at any time; there is no need to include the resource in our model. Therefore, we lose no generality by assuming that every resource R_i has a finite number of units.

Some resources can be used by more than one job at the same time. We model such a resource as a resource type that has many units, each used in a mutually exclusive manner. For

example, a file that can be read by at most ν users at the same time is modeled as a resource that has ν exclusive units. By modeling shared resources in this manner, we do not need to treat them differently.

8.1.1 Enforcement of Mutual Exclusion and Critical Sections

For the most part of this chapter, we assume that a lock-based concurrency control mechanism is used to enforce mutually exclusive accesses of jobs to resources. (We make this assumption for the sake of clarity. As you will see later, commonly used resource access control protocols can be implemented without locks.) When a job wants to use η_i units of resource R_i, it executes a *lock* to request them. We denote this lock request by $L(R_i, \eta_i)$. The job continues its execution when it is granted the requested resource. When the job no longer needs the resource, it releases the resource by executing an *unlock*, denoted by $U(R_i, \eta_i)$. When a resource R_i has only 1 unit, we use the simpler notations $L(R_i)$ and $U(R_i)$ for lock and unlock, respectively. When there are only a few resources and each has only 1 unit, we simply call them by capital letters, such as X, Y, and Z, or by names, such as *Black*, *Shaded*, and so on.

Following the usual convention, we call a segment of a job that begins at a lock and ends at a matching unlock a *critical section*. Furthermore, resources are released in the last-in-first-out order. Hence overlapping critical sections are properly nested.

As an example, Figure 8–1 shows three jobs, J_1, J_2, and J_3, and the time instants when locks and unlocks are executed if each job executes alone starting from time 0. Resources R_1, R_2, and R_3 have only 1 unit each, while resources R_4 and R_5 have many units. Job J_3 has three overlapping critical sections that are properly nested. A critical section that is not included in other critical sections is an *outermost critical section*; the critical section delimited by $L(R_1)$ and $U(R_1)$ in J_3 is an example. Other examples are the critical sections delimited by $L(R_2)$ and $U(R_2)$ in J_2, the second pair of $L(R_3)$ and $U(R_3)$ in J_2 and $L(R_3)$ and $U(R_3)$ in J_1.

When the execution times of the critical sections and the way they are nested are relevant in our discussion, we denote each critical section by a square bracket $[R, \eta; e]$. The entries in the bracket give the name R and the number of units η of the resource used by the job when in the critical section and the (maximum) execution time e of the critical section. In the case where R has only 1 unit, we omit the value of η and use the simpler notation $[R; e]$ instead. In the example in Figure 8–1, the critical section in J_3 that begins at $L(R_5, 4)$ is $[R_5, 4; 3]$ because in this critical section, the job uses 4 units of R_5 and the execution time of this critical section is 3. Concatenations and nestings of the brackets allow us to describe different com-

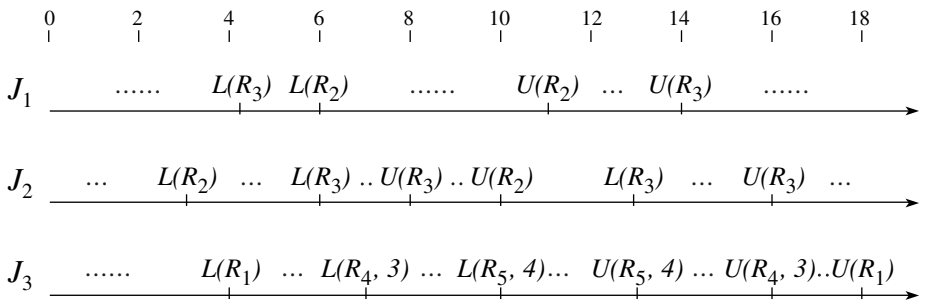


FIGURE 8–1 Examples of critical sections

binations of critical sections. Specifically, we denote nested critical sections by nested square brackets. For example, the nested critical section in J_3 is $[R_1; 14\ [R_4, 3;\ 9\ [R_5, 4; 3]]]$. This notation indicates that the critical section beginning from $L(R_1)$ includes the one beginning from $L(R_4, 3)$, which in turn includes the one beginning from $L(R_5, 4)$. Similarly, critical sections in J_2 are $[R_2; 7\ [R_3; 2]][R_3; 3]$, indicating that there are two nonlapping critical sections in this job.

8.1.2 Resource Conflicts and Blocking

Two jobs *conflict* with one another, or have a *resource conflict*, if some of the resources they require are of the same type. The jobs *contend* for a resource when one job requests a resource that the other job already has. We use these terms interchangeably as the distinction between them is often not important. The scheduler always denies a request if there are not enough free units of the resource to satisfy the request. Sometimes, as we will see later, a scheduler may deny a request even when the requested resource units are free in order to prevent some undesirable execution behavior.

When the scheduler does not grant η_i units of resource R_i to the job requesting them, the lock request $L(R_i, \eta_i)$ of the job fails (or is denied). When its lock request fails, the job is *blocked* and loses the processor. A blocked job is removed from the ready job queue. It stays blocked until the scheduler grants it η_i units of R_i for which the job is waiting. At that time, the job becomes *unblocked*, is moved backed to the ready job queue, and executes when it is scheduled.

The example in Figure 8–2 illustrates the effect of resource contentions. In this example, there are three jobs, J_1, J_2, and J_3, whose feasible intervals are (6, 14], (2, 17] and (0, 18], respectively. The release time and deadline of each job are marked by the vertical bar on each of the time lines. The jobs are scheduled on the processor on the earliest-deadline-first basis. Hence, J_1 has the highest priority and J_3 the lowest. All three jobs require the resource R, which has only 1 unit. In particular, the critical sections in these jobs are $[R; 2]$, $[R; 4]$, and $[R; 4]$, respectively. Below is a description of this schedule segment. The black boxes in Figure 8–2 show when the jobs are in their critical sections.

1. At time 0, only J_3 is ready. It executes.
2. At time 1, J_3 is granted the resource R when it executes $L(R)$.

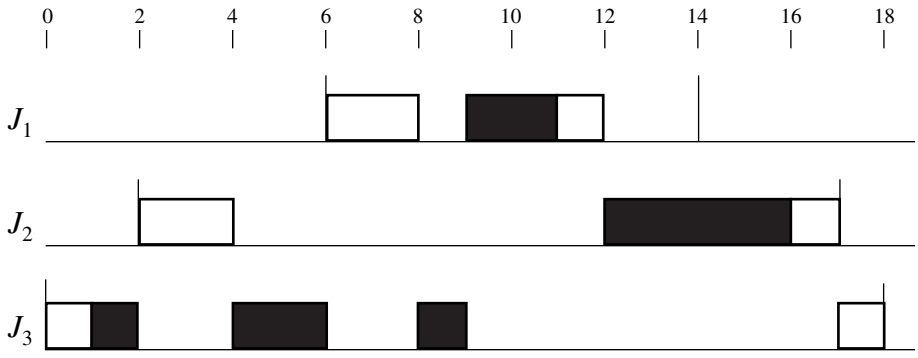


FIGURE 8–2 Example of job interaction due to resource contention.

3. J_2 is released at time 2, preempts J_3, and begins to execute.
4. At time 4, J_2 tries to lock R. Because R is in use by J_3, this lock request fails. J_2 becomes blocked, and J_3 regains the processor and begins to execute.
5. At time 6, J_1 becomes ready, preempts J_3 and begins to execute.
6. J_1 executes until time 8 when it executes a $L(R)$ to request R. J_3 still has the resource. Consequently, J_1 becomes blocked. Only J_3 is ready for execution, and it again has the processor and executes.
7. The critical section of J_3 completes at time 9. The resource R becomes free when J_3 executes $U(R)$. Both J_1 and J_2 are waiting for it. The priority of the former is higher. Therefore, the resource and the processor are allocated to J_1, allowing it to resume execution.
8. J_1 releases the resource R at time 11. J_2 is unblocked. Since J_1 has the highest priority, it continues to execute.
9. J_1 completes at time 12. Since J_2 is no longer blocked and has a higher priority than J_3, it has the processor, holds the resource, and begins to execute. When it completes at time 17, J_3 resumes and executes until completion at 18.

This example illustrates how resource contention can delay the completion of higher-priority jobs. It is easy to see that if J_1 and J_2 do not require the resource, they can complete by times 11 and 14, respectively.

8.2 EFFECTS OF RESOURCE CONTENTION AND RESOURCE ACCESS CONTROL

A *resource access-control protocol*, or simply an *access-control protocol*, is a set of rules that govern (1) when and under what conditions each request for resource is granted and (2) how jobs requiring resources are scheduled. Before moving on to describe several well-known resource access-control protocols and their performance, let us examine in more detail the undesirable effects of resource contention. By looking more closely at what can happen when jobs contend for resources, we should see more clearly the specific design objectives of such a protocol.

8.2.1 Priority Inversion, Timing Anomalies, and Deadlock

In Section 6.8, we saw that priority inversion can occur when the execution of some jobs or portions of jobs is nonpreemptable. Resource contentions among jobs can also cause priority inversion. Because resources are allocated to jobs on a nonpreemptive basis, a higher-priority job can be blocked by a lower-priority job if the jobs conflict, even when the execution of both jobs is preemptable. In the example in Figure 8–2, the lowest priority job J_3 first blocks J_2 and then blocks J_1 while it holds the resource R. As a result, priority inversion occurs in intervals $(4, 6]$ and $(8, 9]$.

When priority inversion occurs, timing anomalies invariably follow. Figure 8–3 gives an example. The three jobs are the same as those shown in Figure 8–2, except that the critical section in J_3 is $[R; 2.5]$. In other words, the execution time of the critical section in J_3 is shortened by 1.5. If we were not warned earlier in Section 4.8 about timing anomalies, our intuition might tell us that as a consequence of this reduction in J_3's execution time, all jobs

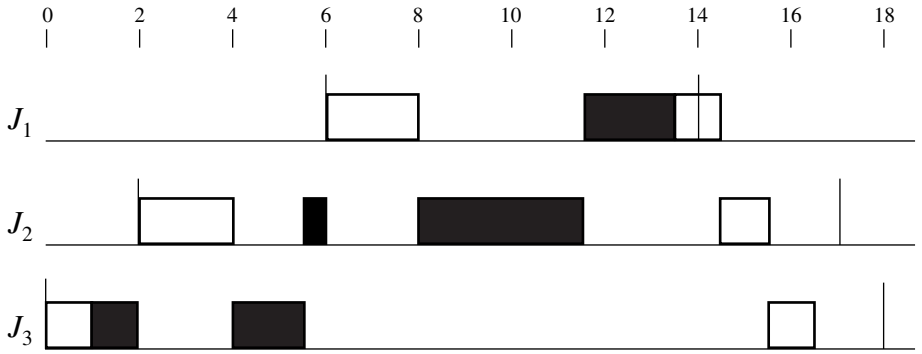


FIGURE 8–3 Example illustrating timing anomaly.

should complete sooner. Indeed, this reduction does allow jobs J_2 and J_3 to complete sooner. Unfortunately, rather than meeting its deadline at 14, J_1 misses its deadline because it does not complete until 14.5.

More seriously, without good resource access control, the duration of a priority inversion can be unbounded. The example in Figure 8–4 illustrates this fact. Here, jobs J_1 and J_3 have the highest priority and lowest priority, respectively. At time 0, J_3 becomes ready and executes. It acquires the resource R shortly afterwards and continues to execute. After R is allocated to J_3, J_1 becomes ready. It preempts J_3 and executes until it requests resource R at time 3. Because the resource is in use, J_1 becomes blocked, and a priority inversion begins. While J_3 is holding the resource and executes, a job J_2 with a priority higher than J_3 but lower than J_1 is released. Moreover, J_2 does not require the resource R. This job preempts J_3 and executes to completion. Thus, J_2 lengthens the duration of this priority inversion. In this situation, the priority inversion is said to be uncontrolled [ShRL90]. There can be an arbitrary number of jobs with priorities lower than J_1 and higher than J_3 released in the meantime. They can further lengthen the duration of the priority inversion. Indeed, when priority inversion is uncontrolled, a job can be blocked for an infinitely long time.

Nonpreemptivity of resource allocation can also lead to deadlocks. The classic example is one where there are two jobs that both require resources X and Y. The jobs are in deadlock

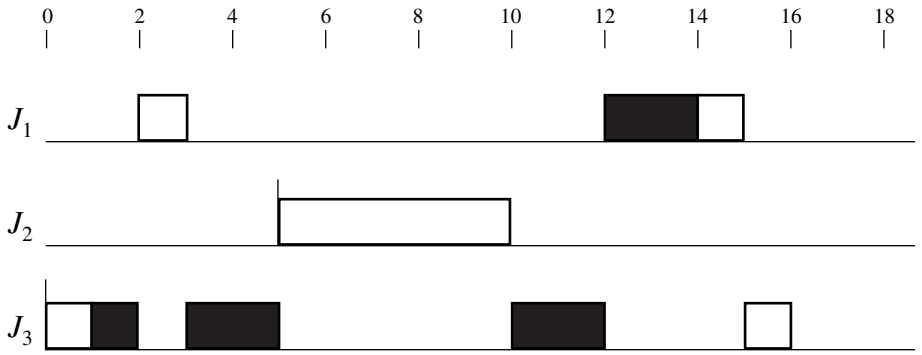


FIGURE 8–4 Uncontrolled priority inversion.

when one of them holds X and requests for Y, while the other holds Y and requests for X. The conditions that allow this circular wait of jobs for each other (i.e., a deadlock) to occur are well-known [Crow, SiGa].

From these examples, we see that no resource access-control protocol can eliminate the priority inversion and anomalous behavior caused by resource contention. A more realistic goal of such a protocol is that it keeps the delays thus incurred as short as possible. For this reason, a criterion we use to measure the performance of a resource access-control protocol is the blocking time of each job. A good resource access-control protocol should control priority inversion and prevent deadlock and, thus, keep the blocking time of every job bounded from the above.

8.2.2 Additional Terms, Notations, and Assumptions

In most parts of this chapter, we ignore all the other factors that can cause a job to be blocked, so, *no job ever suspends itself and every job is preemptable on the processor*. We will discuss the effect of self-suspension and nonpreemptivity at the end of the chapter.

We sometimes use J_h and J_l to denote a higher-priority job and a lower-priority job, respectively. The priorities of these jobs are denoted by π_h and π_l, respectively. In general, the priority of a job J_i is π_i. As in earlier chapters, we represent priorities by integers; the smaller the integer, the higher the priority.

A higher-priority job J_h is said to be *directly blocked* by a lower-priority job J_l when J_l holds some resource which J_h requests and is not allocated. In the example in Figure 8–2, J_3 directly blocks J_2 at time 5.

We describe the dynamic-blocking relationship among jobs using a wait-for graph. In the *wait-for graph* of a system, every job that requires some resource is represented by a vertex labeled by the name of the job. There is also a vertex for every resource in the system, labeled by the name and the number of units of the resource. At any time, the wait-for graph contains an (ownership) edge with label x from a resource vertex to a job vertex if x units of the resource are allocated to the job at the time. There is a (wait-for) edge with label y from a job vertex to a resource vertex if the job requested y units of the resource earlier and the request was denied. In other words, the job is waiting for the resource. (Clearly, y plus the sum of all the labels of edges from the resource vertex is larger than the number of units of the resource.) Therefore, a path in a wait-for graph from a higher-priority job to a lower-priority job represents the fact that the former is directly blocked by the latter. A cyclic path in a wait-for graph indicates a deadlock.

The simple graph in Figure 8–5 is an example. It represents the state of the system in Figure 8–2 at time 5: The resource R is allocated to J_3 and J_2 is waiting for the resource. The path from J_2 to J_3 indicates that J_2 is directly blocked by J_3. Later, J_1 will also be directly

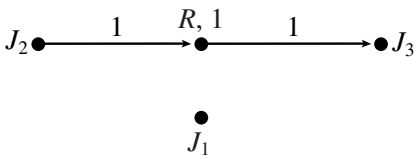


FIGURE 8–5 A wait-for-graph of the system in Figure 8–2 at time 5.

blocked by J_3 when it requests and is denied the resource, and the wait-for graph of the system will have an additional edge from J_1 to R. Since there is only one resource, deadlock can never occur.

In a system of periodic tasks, we say that a periodic task T_i has a critical section $[R, x; y]$, or that the task requires x units of resource R for y units of time, if every job in the task requires at most x units of R for at most y units of time. We sometimes denote the periodic task by the tuple $(\phi_i, p_i, e_i, D_i, [R, x; y])$. In general, when jobs in the task require more than one resource and hence have more than one critical section, we put all the critical sections in the tuple. Hence if J_2 in Figure 8–1 is a job in a periodic task of period 100, execution time 20, zero phase, and relative deadline 100, and if no job in the task has any longer critical section, we call the task $(100, 20; [R_2; 7[R_3; 2]][R_3; 3])$. When the parameters of the critical sections are not relevant, we also denote the task T_i by a tuple $(\phi_i, p_i, e_i, D_i; c_i)$; the last element c_i is the maximum execution time of the longest critical section of the jobs in the periodic task. (The execution times of all the critical sections are included in the execution time e_i of the task.) In this simpler way, the task $(100, 20; [R_2; 7[R_3; 2]][R_3; 3])$ is also called $(100, 20; 7)$.

We will specify resource requirements of a system by a bipartite graph in which there is a vertex for every job (or periodic task) and every resource. Each vertex is named by the name of the job (or task) or resource it represents. The integer next to each resource vertex R_i gives the number v_i of units of the resource. The fact that a job J (or task T) *requires* a resource R_i is represented by an edge from the job (or task) vertex J (or T) to the resource vertex R_i. We may label each edge by one or more 2-tuples or numbers, each for a critical section of the job (or task) which uses the resource. The first element of each 2-tuple gives the number of units used in the critical section. We usually omit this element when there is only 1 unit of the resource. The second element, which is always given, specifies the duration of the critical section. Figure 8–6 gives two examples. The simple graph in Figure 8–6(a) gives

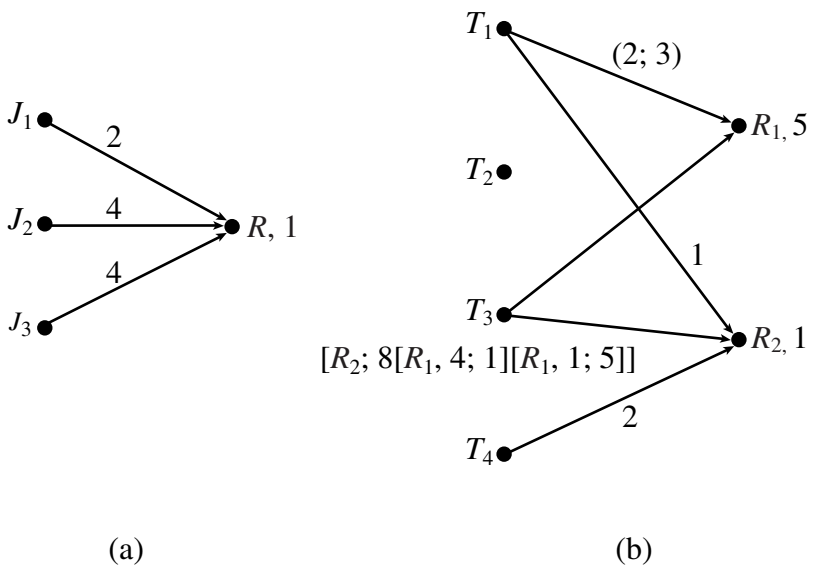


FIGURE 8–6 Graphs that specify resource requirements.

the resource requirements of the jobs in the system in Figure 8–2. The labels of the edges are the durations of the critical sections of the respective jobs. The graph in Figure 8-6(b) uses a combination of ways to specify resource requirements. The system has four periodic tasks. The edges from T_1 tells us that (each job in) this task requires 2 units of R_1 for at most 3 units of time and it requires the resource R_2 for at most 1 unit of time. Similarly, T_4 requires R_2 for 2 units of time. Rather than specifying the resource requirement of T_3 by complicated labels of edges connecting vertex T_3 to resource vertices, we simply provide the information on critical sections of T_3 next to the vertex T_3. There is no edge from T_2, meaning that this task does not require any resource.

Finally, to avoid verboseness whenever there is no ambiguity, by a critical section, we mean an outermost critical section. By a critical section of a periodic task, we mean a critical section of each job in the periodic task.

8.3 NONPREEMPTIVE CRITICAL SECTIONS

The simplest way to control access of resources is to schedule all critical sections on the processor nonpreemptively [Mok]. (In other words, when a job requests a resource, it is always allocated the resource. When a job holds any resource, it executes at a priority higher than the priorities of all jobs.) This protocol is called the *Nonpreemptive Critical Section* (*NPCS*) *protocol*. Because no job is ever preempted when it holds any resource, deadlock can never occur.

Take the jobs in Figure 8–4 for example. Figure 8–7(a) shows the schedule of these jobs when their critical sections are scheduled nonpreemptively on the processor. According to this schedule, J_1 is forced to wait for J_3 when J_3 holds the resource. However, as soon as J_3 releases the resource, J_1 becomes unblocked and executes to completion. Because J_1 is not delayed by J_2, it completes at time 10, rather than 15 according to the schedule in Figure 8–4.

In general, uncontrolled priority inversion illustrated by Figure 8–4 can never occur. The reason is that a job J_h can be blocked only if it is released when some lower-priority job is in a critical section. If it is blocked, once the blocking critical section completes, all resources are free. No lower-priority job can get the processor and acquire any resource until J_h completes. Hence, J_h can be blocked only once, and its blocking time due to resource conflict is at most equal to the maximum execution time of the critical sections of all lower-priority jobs.

Specifically, the blocking time $b_i(rc)$ due to resource conflict of a periodic task T_i in a fixed-priority system of n periodic tasks is equal to

$$b_i(rc) = \max_{i+1 \le k \le n} (c_k) \tag{8.1}$$

when the tasks are indexed in order of nonincreasing priority. In a system where periodic tasks are scheduled on the EDF basis, a job in task T_i with relative deadline D_i can be blocked only by jobs in tasks with relative deadlines longer than D_i. This was stated in Theorem 6.18. Therefore, the blocking time $b_i(rc)$ of T_i is again given by Eq. (8.1) if we index the periodic tasks according to their relative deadlines so that $i < j$ if $D_i < D_j$.

As an example, suppose that the tasks in the system in Figure 8–6(b) are scheduled on a fixed-priority basis. The blocking time $b_1(rc)$ of the highest priority task T_1 is 8, the execution time of the (outermost) critical section of T_3. Similarly, $b_2(rc)$ is 8, while $b_3(rc)$ is 2, the execution time of the critical section of T_4. No task blocks T_4 since it has the lowest

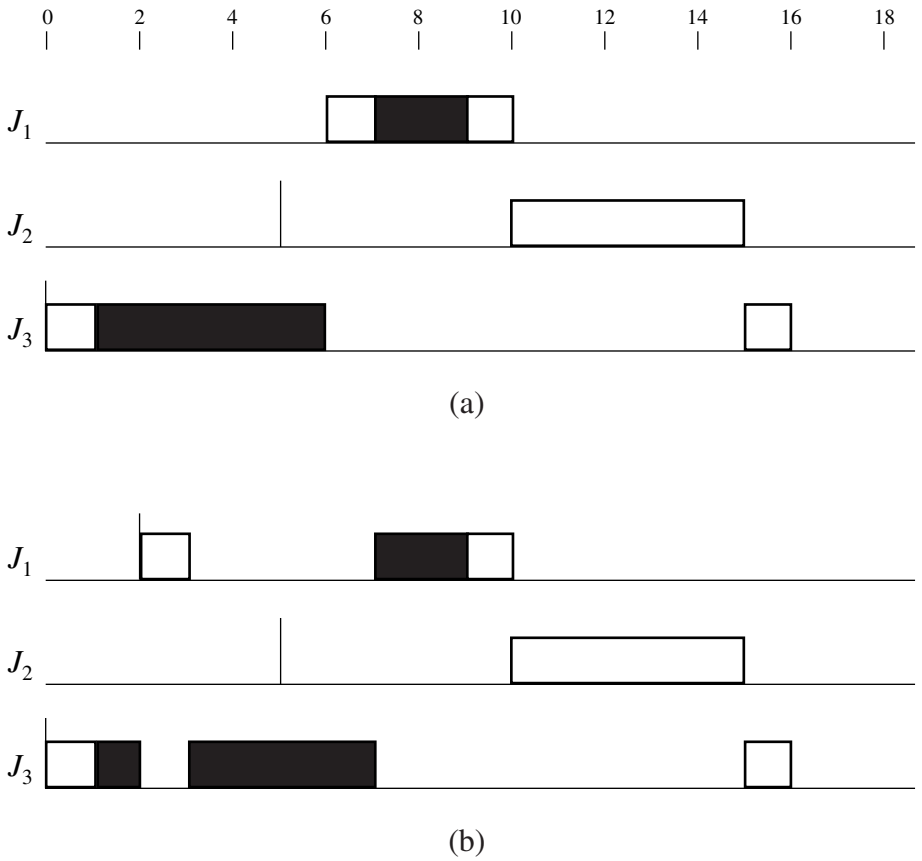


FIGURE 8–7 Example to illustrate two simple protocols. (a) Controlling priority inversion by disallowing preemption of critical section. (b) Controlling priority inversion by using priority inheritance.

priority. Suppose that the relative deadlines of the tasks are $D_1 < D_2 < D_3 < D_4$ and the tasks are scheduled on an EDF basis. Then the blocking times $b_i(rc)$ for $i = 1, 2, 3$, and 4 are also 8, 8, 2, and 0, respectively.

The most important advantage of the NPCS protocol is its simplicity, especially when the numbers of resource units are arbitrary. The protocol does not need any prior knowledge about resource requirements of jobs. It is simple to implement and can be used in both fixed-priority and dynamic-priority systems. It is clearly a good protocol when all the critical sections are short and when most of the jobs conflict with each other.

An obvious shortcoming of this protocol is that every job can be blocked by every lower-priority job that requires some resource even when there is no resource conflict between them. When the resource requirements of all jobs are known, an improvement is to let a job holding any resource execute at the highest priority of all jobs requiring the resource. This is indeed how the *ceiling-priority protocol* supported by the Real-Time Systems Annex of Ada95 [Cohe96] works. We will discuss the ceiling-priority protocol and its worst-case performance in detail in Section 8.6, after examining two other well-known protocols that aim at improving upon the NPCS protocol.

8.4 BASIC PRIORITY-INHERITANCE PROTOCOL

The priority-inheritance protocol proposed by Sha, *et al.* [ShRL90] is also a simple protocol. It works with any preemptive, priority-driven scheduling algorithm. Like the NPCS protocol, it does not require prior knowledge on resource requirements of jobs. The priority-inheritance protocol does not prevent deadlock. When there is no deadlock (i.e., when some other method is used to prevent deadlock), the protocol ensures that no job is ever blocked for an indefinitely long time because uncontrolled priority inversion cannot occur.

In this and the next three sections, we confine our attention to the special case where there is only 1 unit of each resource. (This is the reason for calling the version described here the basic version.) By doing so, we relieve ourselves temporarily of the details that arise due to multiple resource units, so we can focus on the essence of the protocols and the behavior of the system under their control. We will return in Section 8.9 to remove this restrictive assumption.

8.4.1 Definition of Basic Priority-Inheritance Protocol

In the definition of this protocol, as well as other protocols described later, we call the priority that is assigned to a job according to the scheduling algorithm its *assigned priority*. As you will see shortly, at any time t, each ready job J_l is scheduled and executes at its *current priority* $\pi_l(t)$, which may differ from its assigned priority and may vary with time. In particular, the current priority $\pi_l(t)$ of a job J_l may be raised to the higher priority $\pi_h(t)$ of another job J_h. When this happens, we say that the lower-priority job J_l *inherits* the priority of the higher priority job J_h and that J_l executes at its *inherited priority* $\pi_h(t)$. Hereafter, when there is no need to be specific or there is no possible confusion, we will simply say priority when we mean either current or assigned priority.

In its simplest form, the *priority-inheritance protocol* is defined by the following rules. These rules govern the ways current priorities of jobs are set and jobs are scheduled when some of them contend for resources. Again, this version assumes that every resource has only 1 unit.

Rules of the Basic Priority-Inheritance Protocol

1. *Scheduling Rule*: Ready jobs are scheduled on the processor preemptively in a priority-driven manner according to their current priorities. At its release time t, the current priority $\pi(t)$ of every job J is equal to its assigned priority. The job remains at this priority except under the condition stated in rule 3.
2. *Allocation Rule*: When a job J requests a resource R at time t,
   (a) if R is free, R is allocated to J until J releases the resource, and
   (b) if R is not free, the request is denied and J is blocked.
3. *Priority-Inheritance Rule*: When the requesting job J becomes blocked, the job J_l which blocks J inherits the current priority $\pi(t)$ of J. The job J_l executes at its inherited priority $\pi(t)$ until it releases R; at that time, the priority of J_l returns to its priority $\pi_l(t')$ at the time t' when it acquires the resource R.

According to this protocol, a job J is denied a resource only when the resource requested by it is held by another job. (You will see shortly that this is not true for some other protocols.) At time t when it requests the resource, J has the highest priority among all ready jobs. The current priority $\pi_l(t)$ of the job J_l directly blocking J is never higher than the priority $\pi(t)$ of J. Rule 3 relies on this fact.

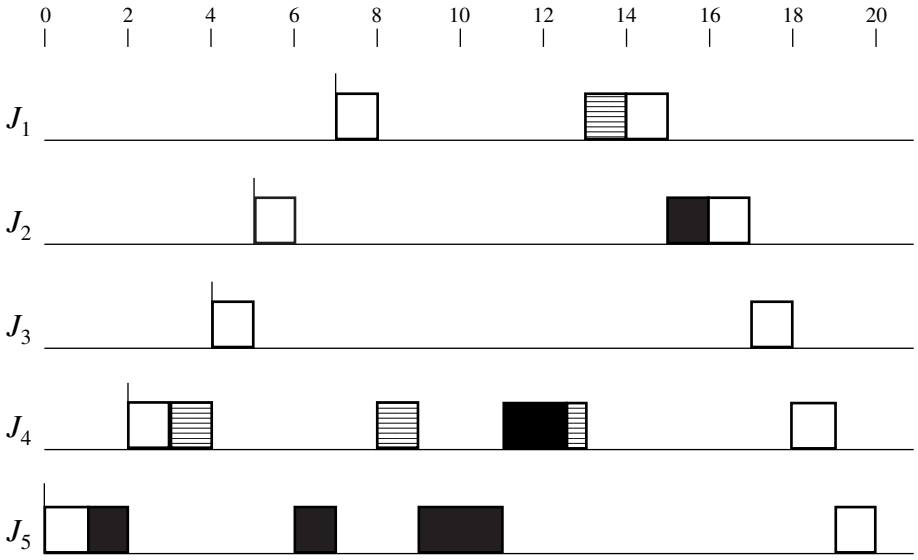
The simple example in Figure 8–7(b) illustrates how priority inheritance affects the way jobs are scheduled and executed. The three jobs in this figure are the same as the ones in Figure 8–7(a). When J_1 requests resource R and becomes blocked by J_3 at time 3, job J_3 inherits the priority π_1 of J_1. When job J_2 becomes ready at 5, it cannot preempt J_3 because its priority π_2 is lower than the inherited priority π_1 of J_3. As a consequence, J_3 completes its critical section as soon as possible. In this way, the protocol ensures that the duration of priority inversion is never longer than the duration of an outermost critical section each time a job is blocked.

Figure 8–8 gives a more complex example. In this example, there are five jobs and two resources *Black* and *Shaded*. The parameters of the jobs and their critical sections are listed in part (a). As usual, jobs are indexed in decreasing order of their priorities: The priority π_i of J_i is i, and the smaller the integer, the higher the priority. In the schedule in part (b) of this figure, black boxes show the critical sections when the jobs are holding *Black*. Shaded boxes show the critical sections when the jobs are holding *Shaded*.

1. At time 0, job J_5 becomes ready and executes at its assigned priority 5. At time 1, it is granted the resource *Black*.
2. At time 2, J_4 is released. It preempts J_5 and starts to execute.
3. At time 3, J_4 requests *Shaded*. *Shaded*, being free, is granted to the job. The job continues to execute.
4. At time 4, J_3 is released and preempts J_4. At time 5, J_2 is released and preempts J_3.
5. At time 6, J_2 executes $L(Black)$ to request *Black*; $L(Black)$ fails because *Black* is in use by J_5. J_2 is now directly blocked by J_5. According to rule 3, J_5 inherits the priority 2 of J_2. Because J_5's priority is now the highest among all ready jobs, J_5 starts to execute.
6. J_1 is released at time 7. Having the highest priority 1, it preempts J_5 and starts to execute.
7. At time 8, J_1 executes $L(Shaded)$, which fails, and becomes blocked. Since J_4 has *Shaded* at the time, it directly blocks J_1 and, consequently, inherits J_1's priority 1. J_4 now has the highest priority among the ready jobs J_3, J_4, and J_5. Therefore, it starts to execute.
8. At time 9, J_4 requests the resource *Black* and becomes directly blocked by J_5. At this time the current priority of J_4 is 1, the priority it has inherited from J_1 since time 8. Therefore, J_5 inherits priority 1 and begins to execute.
9. At time 11, J_5 releases the resource *Black*. Its priority returns to 5, which was its priority when it acquired *Black*. The job with the highest priority among all unblocked jobs is J_4. Consequently, J_4 enters its inner critical section and proceeds to complete this and the outer critical section.
10. At time 13, J_4 releases *Shaded*. The job no longer holds any resource; its priority returns to 4, its assigned priority. J_1 becomes unblocked, acquires *Shaded*, and begins to execute.

Job	r_i	e_i	π_i	Critical Sections
J_1	7	3	1	[*Shaded*; 1]
J_2	5	3	2	[*Black*; 1]
J_3	4	2	3	
J_4	2	6	4	[*Shaded*; 4 [*Black*; 1.5]]
J_5	0	6	5	[*Black*; 4]

(a)



(b)

FIGURE 8–8 Example illustrating transitive inheritance of priority inheritance. (a) Parameters of jobs. (b) Schedule under priority inheritance.

11. At time 15, J_1 completes. J_2 is granted the resource *Black* and is now the job with the highest priority. Consequently, it begins to execute.

12. At time 17, J_2 completes. Afterwards, jobs J_3, J_4, and J_5 execute in turn to completion.

8.4.2 Properties of the Priority-Inheritance Protocol

This example illustrates that when resource accesses are controlled by the priority-inheritance protocol, there are two types of blocking: direct blocking and *priority-inheritance blocking* (or simply *inheritance blocking*). J_2 is directly blocked by J_5 in (6, 11] and by J_4 in (11, 12.5],

and J_1 is directly blocked by J_4 in (8, 13]. In addition, J_3 is blocked by J_5 in (6, 7] because the latter inherits a higher priority in this interval. Later at time 8, when J_4 inherits priority 1 from J_1, J_3 becomes blocked by J_4 as a consequence. Similarly, the job J_2 in Figure 8–7(b) suffers inheritance blocking by J_3 in (5, 7]. This type of blocking suffered by jobs that are not involved in resource contention is the cost for controlling the durations of priority inversion suffered by jobs that are involved in resource contention.

The example in Figure 8–8 also illustrates the fact that jobs can *transitively block* each other. At time 9, J_5 blocks J_4, and J_4 blocks J_1. So, *priority inheritance is transitive*. In the time interval (9, 11), J_5 inherits J_4's priority, which J_4 inherited from J_1. As a consequence, J_5 indirectly inherits the J_1's priority.

The priority-inheritance protocol does not prevent deadlock. To see why, let us suppose that J_5 in this example were to request the resource *Shaded* sometime after *Shaded* has been granted to J_4 (e.g., at time 6.5). These two jobs would be deadlocked.

In addition, *the priority-inheritance protocol does not reduce the blocking times suffered by jobs as small as possible*. It is true that in the absence of a deadlock, a job can be blocked directly by any lower-priority job for at most once for the duration of one outermost critical section. However, in the worst case, a job that requires v resources and conflicts with k lower-priority jobs can be blocked for $\min(v, k)$ times, each for the duration of an outermost critical section. Figure 8–9 shows the worst-case scenario when v is equal to k. Here the job J_1 has the highest priority; it is blocked k times.

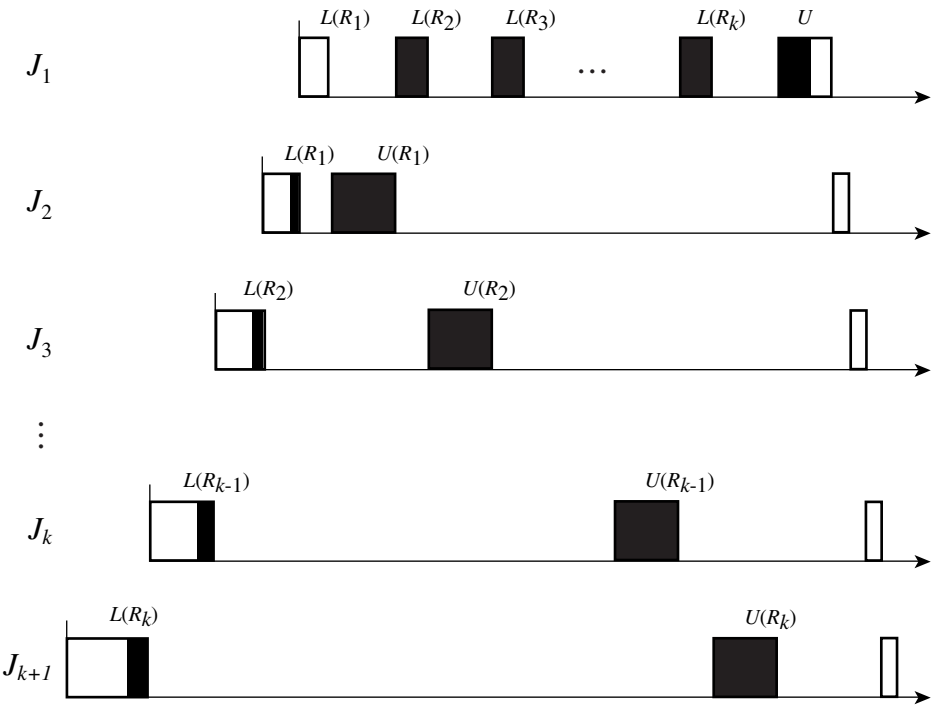


FIGURE 8–9 A worst-case blocking scenario for priority-inheritance protocol.

8.5 BASIC PRIORITY-CEILING PROTOCOL

The *priority-ceiling protocol* [ShRL88, ShRL90] extends the priority-inheritance protocol to prevent deadlocks and to further reduce the blocking time. This protocol makes two key assumptions:

1. The assigned priorities of all jobs are fixed.
2. The resources required by all jobs are known a priori before the execution of any job begins.

To define the protocol, we need two additional terms. The protocol makes use of a parameter, called priority ceiling, of every resource. The *priority ceiling* of any resource R_i is the highest priority of all the jobs that require R_i and is denoted by $\Pi(R_i)$. For example, the priority ceiling $\Pi(Black)$ of the resource *Black* in the example in Figure 8–8 is 2 because J_2 is the highest priority job among the jobs requiring it. Similarly, $\Pi(Shaded)$ is 1. Because of assumption 2, the priority ceilings of all resources are known a priori. We note that if the resource access control protocol includes the priority-inheritance rule, then a job can inherit a priority as high as x during its execution if it requires a resource with priority ceiling x.

At any time t, the *current priority ceiling* (or simply the *ceiling*) $\hat{\Pi}(t)$ of the system is equal to the highest priority ceiling of the resources that are in use at the time, if some resources are in use. If all the resources are free at the time, the current ceiling $\hat{\Pi}(t)$ is equal to Ω, a nonexisting priority level that is lower than the lowest priority of all jobs. As an example, we again look at the system in Figure 8–8. In the interval $[0, 1)$ when both resources in the system are free, the current ceiling of the system is equal to Ω, lower than 5, the priority of the lowest priority job J_5. In $(1, 3]$, *Black* is held by J_5; hence, the current ceiling of the system is 2. In $(3, 13]$ when *Shaded* is also in use, the current ceiling of the system is 1, and so it is in $(13, 14]$.

8.5.1 Definition of the Basic Priority-Ceiling Protocol

We now define the priority-ceiling protocol for the case when there is only 1 unit of every resource.

Rules of Basic Priority-Ceiling Protocol

1. *Scheduling Rule*:
   - **(a)** At its release time t, the current priority $\pi(t)$ of every job J is equal to its assigned priority. The job remains at this priority except under the condition stated in rule 3.
   - **(b)** Every ready job J is scheduled preemptively and in a priority-driven manner at its current priority $\pi(t)$.
2. *Allocation Rule*: Whenever a job J requests a resource R at time t, one of the following two conditions occurs:
   - **(a)** R is held by another job. J's request fails and J becomes blocked.
   - **(b)** R is free.
     - **(i)** If J's priority $\pi(t)$ is higher than the current priority ceiling $\hat{\Pi}(t)$, R is allocated to J.

(ii) If J's priority $\pi(t)$ is not higher than the ceiling $\hat{\Pi}(t)$ of the system, R is allocated to J only if J is the job holding the resource(s) whose priority ceiling is equal to $\hat{\Pi}(t)$; otherwise, J's request is denied, and J becomes blocked.

3. *Priority-Inheritance Rule*: When J becomes blocked, the job J_l which blocks J inherits the current priority $\pi(t)$ of J. J_l executes at its inherited priority until the time when it releases every resource whose priority ceiling is equal to or higher than $\pi(t)$; at that time, the priority of J_l returns to its priority $\pi_l(t')$ at the time t' when it was granted the resource(s).

We note that (ii) in rule 2 assumes that only one job holds all the resources with priority ceiling equal to $\hat{\Pi}(t)$. Similarly, rule 3 assumes that only one job is responsible for J's request being denied, because it holds either the requested resource or a resource with priority ceiling $\hat{\Pi}(t)$. We will return shortly to show that these assumptions are true.

Figure 8–10 shows the schedule of the system of jobs whose parameters are listed in Figure 8–8(a) when their accesses to resources are controlled by the priority-ceiling proto-

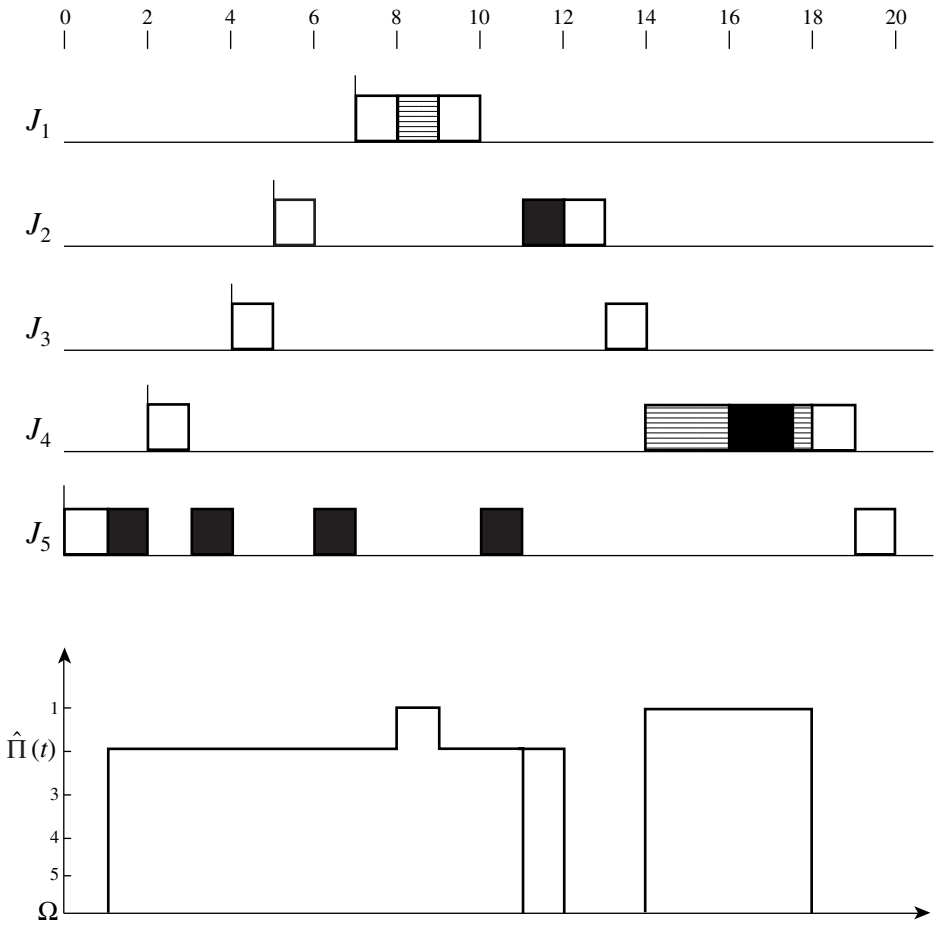


FIGURE 8–10 A schedule illustrating priority-ceiling protocol.

col. As stated earlier, the priority ceilings of the resources *Black* and *Shaded* are 2 and 1, respectively.

1. In the interval (0, 3], this schedule is the same as the schedule shown in Figure 8–8, which is produced under the basic priority-inheritance protocol. In particular, the ceiling of the system at time 1 is Ω. When J_5 requests *Black*, it is allocated the resource according to (i) in part (b) of rule 2. After *Black* is allocated, the ceiling of the system is raised to 2, the priority ceiling of *Black*.
2. At time 3, J_4 requests *Shaded*. *Shaded* is free; however, because the ceiling $\hat{\Pi}(3)$ (= 2) of the system is higher than the priority of J_4, J_4's request is denied according to (ii) in part (b) of rule 2. J_4 is blocked, and J_5 inherits J_4's priority and executes at priority 4.
3. At time 4, J_3 preempts J_5, and at time 5, J_2 preempts J_3. At time 6, J_2 requests *Black* and becomes directly blocked by J_5. Consequently, J_5 inherits the priority 2; it executes until J_1 becomes ready and preempts it. During all this time, the ceiling of the system remains at 2.
4. When J_1 requests *Shaded* at time 8, its priority is higher than the ceiling of the system. Hence, its request is granted according to (i) in part (b) of rule 2, allowing it to enter its critical section and complete by the time 10. At time 10, J_3 and J_5 are ready. The latter has a higher priority (i.e., 2); it resumes.
5. At 11, when J_5 releases *Black*, its priority returns to 5, and the ceiling of the system drops to Ω. J_2 becomes unblocked, is allocated *Black* [according to (i) in part (b) of rule 2], and starts to execute.
6. At time 14, after J_2 and J_3 complete, J_4 has the processor and is granted the resource *Shaded* because its priority is higher than Ω, the ceiling of the system at the time. It starts to execute. The ceiling of the system is raised to 1, the priority ceiling of *Shaded*.
7. At time 16, J_4 requests *Black*, which is free. The priority of J_4 is lower than $\hat{\Pi}(16)$, but J_4 is the job holding the resource (i.e., *Shaded*) whose priority ceiling is equal to $\hat{\Pi}(16)$. Hence, according to (ii) of part (b) of rule 2, J_4 is granted *Black*. It continues to execute. The rest of the schedule is self-explanatory.

Comparing the schedules in Figures 8–8 and 8–10, we see that when priority-ceiling protocol is used, J_4 is blocked at time 3 according to (ii) of part (b) of rule 2. A consequence is that the higher priority jobs J_1, J_2, and J_3 all complete earlier at the expense of the lower priority job J_4. This is the desired effect of the protocol.

8.5.2 Differences between the Priority-Inheritance and Priority-Ceiling Protocols

A fundamental difference between the priority-inheritance and priority-ceiling protocols is that the former is greedy while the latter is not. You recall that the allocation rule (i.e., rule 2) of the priority-inheritance protocol lets the requesting job have a resource whenever the resource is free. In contrast, according to the allocation rule of the priority-ceiling protocol, a job may be denied its requested resource even when the resource is free at the time. (This is what happens to J_4 at time 3 in the above example.) We will return shortly to discuss the consequence of this action.

The priority-inheritance rules of these two protocols are essentially the same. In principle, both rules say that whenever a lower priority job J_l blocks the job J whose request is just denied, the priority of J_l is raised to J's priority $\pi(t)$. The difference arises because of

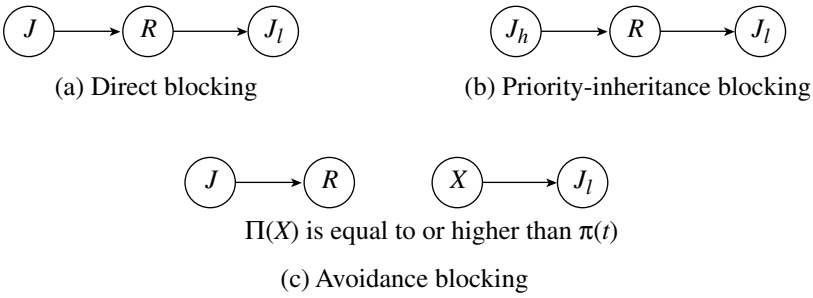


FIGURE 8–11 Ways for a job to block another job. (a) Direct blocking. (b) Priority-inheritance blocking. (c) Avoidance blocking.

the nongreedy nature of the priority-ceiling protocol. It is possible for J to be blocked by a lower-priority job which does not hold the requested resource according to the priority-ceiling protocol, while this is impossible according to the priority-inheritance protocol.

The wait-for graphs in Figure 8–11 illustrate the three ways in which a job J can be blocked by a lower-priority job when resource accesses are controlled by the priority-ceiling protocol. Of course, J can be directly blocked by a lower-priority job J_l, as shown by the wait-for graph [Figure 8–11(a)]. As a consequence of the priority-inheritance rule, a job J can also be blocked by a lower-priority job J_l which has inherited the priority of a higher-priority job J_h. The wait-for graph in Figure 8–11(b) shows this situation. (For simplicity, we show only the inheritance due to direct blocking.) As stated in Section 8.4, this is priority-inheritance blocking.

The allocation rule may cause a job J to suffer priority-ceiling blocking, which is represented by the graph in Figure 8–11(c). The requesting job J is blocked by a lower-priority job J_l when J requests a resource R that is free at the time. The reason is that J_l holds another resource X whose priority ceiling is equal to or higher than J's priority $\pi(t)$. Rule 3 says that a lower-priority job directly blocking or priority-ceiling blocking the requesting job J inherits J's priority $\pi(t)$.

Priority-ceiling blocking is sometimes referred to as *avoidance blocking*. The reason for this term is that the blocking caused by the priority-ceiling rule is the cost for avoidance of deadlocks among jobs. Hereafter, we will use the terms avoidance blocking and priority-ceiling blocking interchangeably.

8.5.3 Deadlock Avoidance by Priority-Ceiling Protocol

You recall from your study on principles of operating systems that one way to avoid deadlock is to use the ordered-resource technique [Have]. The set of priority ceilings of resources impose a linear order on all the resources. It may not surprise you that deadlock can never occur under the priority-ceiling protocol.

In order to gain a deeper insight into how the protocol works to prevent deadlock, we pause to look at a more complicated example. In the example in Figure 8–12, there are three jobs: J_1, J_2, and J_3 with priorities 1, 2, and 3, respectively. Their release times are 3.5, 1, and 0 and their critical sections are [*Dotted*; 1.5], [*Black*; 2 [*Shaded*; 0.7]], and [*Shaded*; 4.2 [*Black*; 2.3]], respectively. In this schedule, the intervals during which the jobs are in their critical sections are shown as the dotted box (the critical section associated with resource *Dotted*),

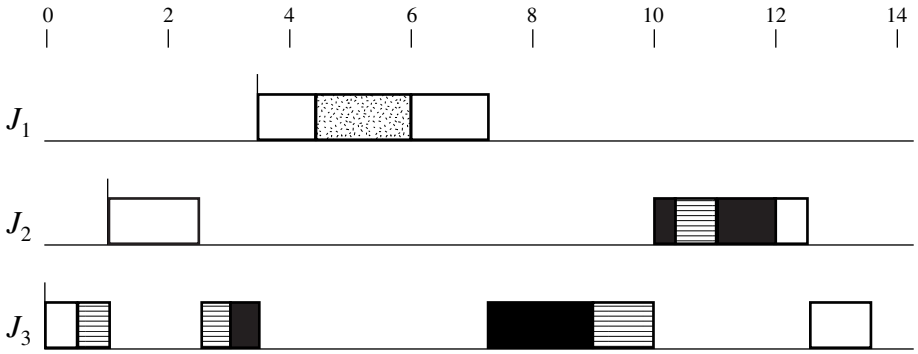


FIGURE 8–12 Example illustrating how priority-ceilling protocol prevents deadlock.

shaded boxes (critical sections associated with resource *Shaded*), and black boxes (critical sections associated with resource *Black*).

1. When J_3 requests *Shaded* at time 0.5, no resource is allocated at the time. J_3's request is granted. When job J_2 becomes ready at time 1, it preempts J_3.
2. At time 2.5, J_2 requests *Black*. Because *Shaded* is already allocated to J_3 and has priority ceiling 2, the current ceiling of the system is 2. J_2's priority is 2. According to (ii) of part (b) of rule 2, J_2 is denied *Black*, even though the resource is free. Since J_2 is blocked, J_3 inherits the priority 2 (rule 3), resumes, and starts to execute.
3. When J_3 requests *Black* at time 3, it is holding the resource whose priority ceiling is the current ceiling of the system. According to (ii) of part (b) of rule 2, J_3 is granted the resource *Black*, and it continues to execute.
4. J_3 is preempted again at time 3.5 when J_1 becomes ready. When J_1 requests *Dotted* at time 4.5, the resource is free and the priority of J_1 is higher than the ceiling of the system. (i) of part (b) of rule 2 applies, and *Dotted* is allocated to J_1, allowing the job to enter into its critical section and proceed to complete at 7.3. The description of the segment after this time is left to you.

We now use this example to explain intuitively why priority-ceiling protocol prevents deadlock. To see the rationale behind (ii) of part (b) of rule 2, which leads to the denial of J_2's request for *Black*, we note that at the time *Black* is free but *Shaded* is already allocated to J_3. The fact that the priority ceiling of *Shaded* is 2 indicates that some job with priority 2 requires this resource and this job may very well be J_2, as is indeed the case in this example. If J_2 were allowed to have *Black*, it might later request *Shaded* and would be blocked by J_3. J_3 would execute and might later request *Black*. Denying J_2's access to *Black* is one way to prevent this deadlock. On the other hand, suppose that the priority ceiling of *Shaded* were lower than the priority of J_2. This fact would indicate that J_2 does not require *Shaded*. Moreover, no job with priority equal to or higher than J_2 requires this resource. Consequently, it would not be possible for the job holding *Shaded* to later inherit a higher priority than J_2, preempt J_2, and

request *Black*. This is the rationale of (i) of part (b) of rule 2. Indeed, this is the reason that J_1 was granted *Dotted*.

Let us now state in general terms what was said above. At any time t, the priority $\pi(t)$ of a job J being higher than the current ceiling $\hat{\Pi}(t)$ of the system means that (1) job J will not require any of the resources in use at t and (2) jobs with priorities equal to or higher than J will not require any of these resource. In other words, the priority ceiling $\hat{\Pi}(t)$ of the system tells us the subset of all jobs to which we can safely grant free resources at time t; this subset contains all the jobs that have higher priorities than $\hat{\Pi}(t)$. Because of (1), J will not request any resource that is in use at the time. Because of (2), no job holding any resource at the time can inherit a higher priority than J, later preempt J, and request resources allocated to J after t. For these reasons, (i) of part (b) of rule 2 will not lead to any deadlock.

(ii) in part (b) of rule 2 states an exception to the rule that J's request for any resource is denied if its priority is not higher than the ceiling of the system. The exception applies when J is the job holding the resource(s) whose priority ceiling(s) is equal to $\hat{\Pi}(t)$; under this condition, J is granted the resource it requests at t. (This exception is necessary in order to ensure that a job can make progress as it acquires resources. Otherwise, the job would block itself!) J's priority $\pi(t)$ must be equal to $\hat{\Pi}(t)$ when it is granted its requested resource under this condition. Moreover, because of (i) and (ii) in part (b) of rule 2, no other job is holding resources with priority ceiling equal to $\hat{\Pi}(t)$. Consequently, (ii) in part (b) of rule 2 cannot lead to any deadlock.

The following theorem summarizes this discussion. A formal proof of the theorem can be found in [ShRL90].

> **THEOREM 8.1.** When resource accesses of a system of preemptive, priority-driven jobs on one processor are controlled by the priority-ceiling protocol, deadlock can never occur.

8.5.4 Duration of Blocking

We saw earlier that under the priority-ceiling protocol, a job may be directly blocked, avoidance blocked, and inheritance blocked by lower-priority jobs. A question is whether as a cost of its ability to prevent deadlock, this protocol can cause a job to be blocked for a longer duration than the priority-inheritance protocol. In the worst case, the answer is no. You recall that a job may be blocked for a multiple number of times under the basic priority-inheritance protocol when it conflicts with more than one job over more than one resource. In contrast, under the priority-ceiling protocol, every job is blocked at most once for the duration of a critical section, no matter how many jobs conflict with it. This is stated formally below [ShRL90].

> **THEOREM 8.2.** When resource accesses of preemptive, priority-driven jobs on one processor are controlled by the priority-ceiling protocol, a job can be blocked for at most the duration of one critical section.

Informal Proof of Theorem 8.2. Rather than proving the theorem formally, we use an intuitive argument to convince you that the theorem is true. There are two parts to this argument: (1) When a job becomes blocked, it is blocked by only one job, and (2) a job which blocks another job cannot be blocked in turn by some other job. State (2) in another way, *there can be no transitive blocking under the priority-ceiling protocol*.

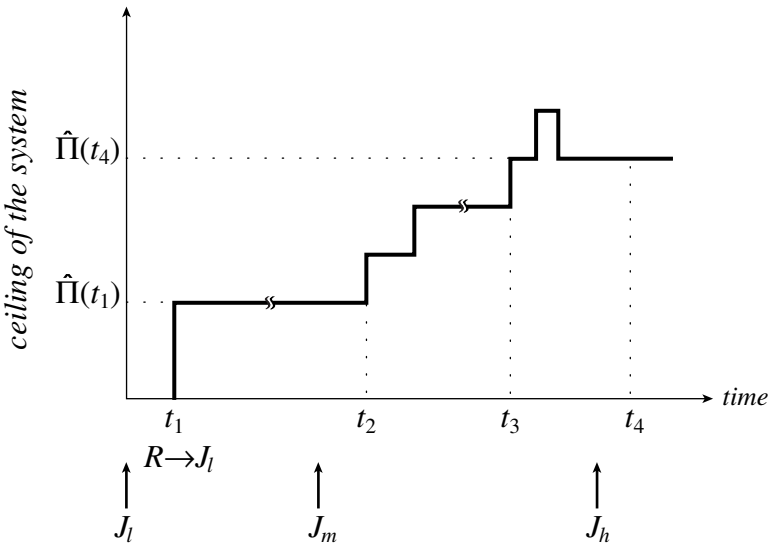


FIGURE 8–13 Scenario illustrating a property of basic priority-ceiling protocol.

We use the scenario in Figure 8–13 to to convince you that (1) is true. There are three jobs, J_l, J_m and J_h. Their release times are indicated by the arrows below the graph, which plots the ceiling of the system as a function of time. The priority π_l of J_l is lower than the priority π_m of J_m, which is in turn lower than the priority π_h of J_h. Suppose that at time t_1, J_l requests and is granted a resource R. As a consequence, the ceiling of the system rises to $\hat{\Pi}(t_1)$, which is the priority ceiling of R. Later, J_m preempts J_l and acquires some resources (say at and after t_2) while it executes. Clearly this is possible only if its priority π_m is higher than $\hat{\Pi}(t_1)$. Suppose that at time t_4, J_h becomes blocked. We note that it is not possible for J_h to be directly blocked by J_l. Otherwise, the ceiling of the system would be at least as high as π_h when J_m requests resources, and J_m would not be able to acquire any resource. If all the resources allocated since t_2 are held by J_m at time t_4, we have shown that J_m is the only job blocking J_h. On the other hand, suppose that at some time t_3 before t_4, a resource R' is allocated to some other job J_k. Then, J_m cannot be blocking J_h for the same reason that J_l cannot be blocking J_h.

This inductive argument allows us to conclude that J_h is either directly blocked or priority-ceiling blocked by only one job, and this job holds the resource that has the highest priority ceiling among all the resources in use when J_h becomes blocked. For simplicity, we have assumed in this scenario that all three jobs have their assigned priorities. It is easy to see that above argument remains valid even when the jobs have inherited priorities, as long as the priority of J_h is the highest and the priority of J_l is the lowest.

To show that (2) is true, let us suppose that the three jobs J_l, J_m and J_h are blocked transitively. Because the jobs are scheduled according to their priorities, it must be that J_l is preempted after having acquired some resource(s) and later at t, J_m is granted some other resource(s). This can happen only if J_m and all the jobs with higher priorities do not require any of the resources held by J_l at t. Until J_m completes, J_l cannot execute and acquire some other resources. Consequently, J_l cannot inherit a priority equal to or higher than $\pi_m(t)$ until J_m

completes. If transitive blocking were to occur, J_m would inherit $\pi_h(t)$, and J_l would inherit a priority higher than $\pi_m(t)$ indirectly. This leads to a contradiction. Hence, the supposition that the three jobs are transitively blocked must be false.

Computation of Blocking Time. Theorem 8.2 makes it easy for us to compute an upper bound to the amount of time a job may be blocked due to resource conflicts. We call this upper bound the *blocking time* (*due to resource conflicts*) of the job.

To illustrate how to do this computation, let us consider the system of jobs whose resource requirements are given by Figure 8–14. As always, the jobs are indexed in order of decreasing priorities. We see that J_1 can be directed blocked by J_4 for 1 unit of time. The blocking time $b_1(rc)$ of J_1 is clearly one. Although J_2 and J_3 do not require the resource *Black*, they can be priority-inheritance blocked by J_4 since J_4 can inherit priority π_1. Hence, the blocking times $b_2(rc)$ and $b_3(rc)$ are also one.

Figure 8–15(a) shows a slightly more complicated example. Even for this small system, it is error prone if we compute the blocking times of all jobs by inspection, as we did earlier. The tables in Figure 8–15(b) give us a systematic way. There is a row for each job that can be blocked. (In these tables, there is a row for every job except J_6.) The tables list only the nonzero entries; all the other entries are zero. Since jobs are not blocked by higher-priority jobs, the entries at and below "*" in each column are zero.

The leftmost part is the direct-blocking table. It lists for each job the duration for which it can be directly blocked by each of the lower-priority jobs. The entries in this table come directly from the resource requirement graph of the system. Indeed, for the purpose of calculating the blocking times of the jobs, this table gives a complete specification of the resource requirements of the jobs.

The middle part of Figure 8–15(b) is the priority-inheritance blocking table. It lists the maximum duration for which each job can be priority-inheritance blocked by each of the lower-priority jobs. For example, J_6 can inherit priority π_1 of J_1 for 2 units of time when it directly blocks J_1. Hence, it can block all the other jobs for 2 units for time. In the table, we show 2 units of inheritance blocking time of J_2 and J_3 by J_6. However, because J_6 can also inherit π_3 for 4 units of time, it can block J_4 and J_5 for 4 units of time. This is the reason that the entries in the fourth and fifth rows of column 6 are 4. In general, a systematic way to get the entries in each column of this table from the entries in the corresponding column of the direct-blocking table is as follows. *The entry at column k and row i of the inheritance blocking table is the maximum of all the entries in column k and rows* $1, 2, \ldots, i-1$ *of the direct-blocking table*.

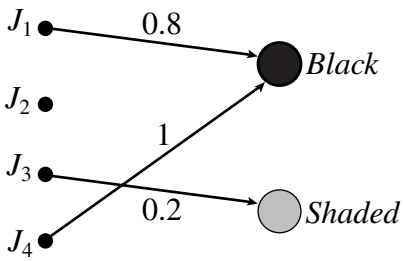
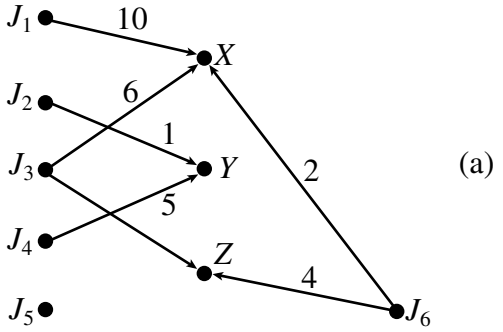


FIGURE 8–14 Example on duration of blocking.



(a)

	Directly blocked by					Priority-inher blocked by					Priority-ceiling blocked by				
	J_2	J_3	J_4	J_5	J_6	J_2	J_3	J_4	J_5	J_6	J_2	J_3	J_4	J_5	J_6
J_1		6			2										
J_2	*		5			*	6			2	*	6			2
J_3		*			4		*	5		2		*	5		2
J_4			*					*		4			*		4
J_5				*					*	4				*	

(b)

FIGURE 8–15 Example illustrating the computation of blocking times.

The rightmost table in Figure 8–15(b) is the avoidance (priority-ceiling) blocking table. It lists the maximum duration for which each job can be avoidance blocked by each lower-priority job. Again, let us focus on column 6. When J_6 holds resource X (whose priority ceiling is the highest in the system), it avoidance blocks all the jobs which require any resource. Similarly, when it holds Z, it avoidance blocks J_4. Therefore, except for the entry in row 5, all entries in column 6 of the avoidance blocking table are the same as the corresponding entries in column 6 of the inheritance blocking table. J_5 does not require any resource and is never directly or avoidance blocked. In general, *when the priorities of all the jobs are distinct, the entries in the avoidance blocking table are equal to corresponding entries in the priority-inheritance blocking table, except for jobs which do not require any resources*. Jobs which do not require any resource are never avoidance blocked, just as they are never directly blocked.

The blocking time $b_i(rc)$ of each job J_i is equal to the maximum value of all the entries in the ith row of the three tables. From Figure 8–15(b), we have $b_i(rc)$ is equal to 6, 6, 5, 4, 4, and 0 for $i = 1, 2, \ldots, 6$, respectively

For this example, every entry in the avoidance blocking table is either equal to or smaller than the corresponding entries in the direct blocking or inheritance blocking tables. Since we

are taking the maximum value of the entries of each row, there is no need for us to compute the avoidance blocking table. Indeed, we do not need this table whenever the priorities of all the jobs are distinct. When the priorities of jobs are not distinct, a job may be avoidance blocked by a job of equal priority. The avoidance blocking table gives this information. For example, suppose that in addition to the jobs in Figure 8–15(a), there is a job J_1' whose priority is also π_1, and this job requires a resource V for 9 units of time. Then the blocking time of J_1' is 10, the amount of time J_1 holds the resource X and priority-ceiling blocks J_1'. Similarly, the blocking time of J_1 is 9, the duration for which it is priority-ceiling blocked by J_1'. In this case, we need the avoidance blocking table to give us these blocking times.

In Problem 8.14, you are asked to provide a pseudocode description of an algorithm that computes the blocking time $b_i(rc)$ of all the jobs from the resource requirement graph of the system. For the sake of efficiency, you may want to first identify for each job J_i the subset of all the jobs that may block the job. This subset is called the *blocking set* of J_i. (In our simple example, J_5 is not included in the blocking set of any other job, since it cannot block any job. The blocking set of J_i includes all the lower-priority jobs other than J_5.)

8.5.5 Fixed-Priority Scheduling and Priority-Ceiling Protocol

The priority-ceiling protocol is an excellent algorithm for controlling the accesses to resources of periodic tasks when the tasks are scheduled on a fixed-priority basis. It is reasonable to assume that the resources required by every job of every task and the maximum execution times of their critical sections are known a priori, just like the other task parameters. All the jobs in each periodic task have the same priority. Hence, the priority ceiling of each resource is the highest priority of all the tasks that require the resource. This makes it possible for us to analyze and determine the potential of resource contentions among tasks statically. The effect of resource contentions on the schedulability of the tasks can be taken care of by including the blocking time $b_i(rc)$ in the schedulability test of the system.

For example, suppose that the jobs in Figure 8–14 belong to four periodic tasks. The tasks are $T_1 = (\varepsilon, 2, 0.8; [Black; 0.8])$, $T_2 = (\varepsilon, 2.2, 0.4)$, $T_3 = (\varepsilon, 5, 0.2; [Shaded; 0.2])$, and $T_4 = (10, 1.0; [Black; 1.0])$, where ε is a small positive number. For all i, J_i in Figure 8–14 is a job in T_i. Figure 8–16 shows the initial segment of the schedule of the tasks according to the rate-monotonic algorithm and priority-ceiling protocol. We see that T_2 misses it is deadline at time $2.2 + \varepsilon$. A schedulability test can predict this miss. The time-demand function of T_2 is equal to 2.2 (i.e., $0.8 + 0.4 + 1.0$) in $(0, 2.0 + \varepsilon]$ when the blocking time $b_2(rc) = 1.0$ of T_2 is included and becomes 3.0 at $2.0+\varepsilon$ (i.e., the beginning of the second period of T_1). Obviously, the time supply by T_2's first deadline at $2.2+\varepsilon$ cannot meet this demand. Similarly, if the jobs J_i, for $i = 1, 2, \ldots, 6$, in Figure 8–15 are jobs in periodic tasks T_i, respectively, we can take the effect of resource conflicts into account in our determination of whether the tasks are schedulable by including the blocking time $b_i(rc)$ computed above in the schedulability test.

To summarize this section, we recall that two factors contribute to the time-demand function of each task in addition to the execution times of its jobs and execution times of equal and higher-priority jobs. They are blocking time and context-switching time. When the system is scheduled on a fixed-priority basis and uses the priority-ceiling protocol, we can compute the blocking time $b_i(rc)$ of each task T_i due to its resource conflicts with other tasks in the way described above. After we have thus obtained $b_i(rc)$, we include this blocking factor with the other types of blocking times (e.g., due to nonpreemptivity of lower-priority

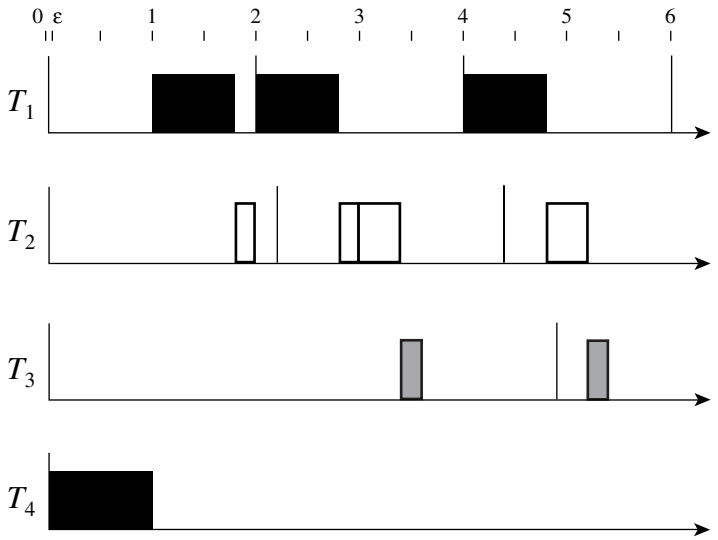


FIGURE 8–16 Example on fixed-priority scheduling and priority-ceiling protocol. ($T_1 = (\varepsilon, 2, 0.8; [Black; 0.8])$, $T_2 = (\varepsilon, 2.2, 0.4)$, $T_3 = (\varepsilon, 5, 0.2; [Shaded; 0.2])$, $T_4 = (10, 1; [Black; 1.0])$).

jobs and self-suspension) which the task may suffer to obtain the total blocking time b_i of the task.

In Section 6.8, we showed that each job in a system of fixed-priority tasks suffers at most two context switches when the tasks are independent. When the tasks contend for resources under the control of the basic priority-ceiling protocol, each job can be blocked at most once. In particular, a job that requires some resources may be blocked and lose the processor when it requests some resource and, as a consequence, suffers two additional context switches. In contrast, a job which does not require any resource suffers only two context switches: one when it starts and one when it ends. (You recall that the context-switch time when a job is preempted is attributed to the preempting job.) Hence, to account for the context switch overhead in schedulability test, we add

1. *two CS to the execution time of each task that does not require any resource and*
2. *four CS to the execution time of each task that requires one or more resource,*

where CS is the maximum time to complete a context switch.[1]

8.6 STACK-BASED, PRIORITY-CEILING (CEILING-PRIORITY) PROTOCOL

In this section, we give two different definitions of a protocol that is simpler than the priority-ceiling protocol but has the same worst-case performance as the priority-ceiling protocol. The different definitions arise from two different motivations: to provide stack-sharing capability

[1]It is important to remember the assumption that no job ever suspends itself. We will discuss the effect of self-suspension in Section 8.11.

and to simplify the priority-ceiling protocol. They led to the two different names of the same protocol.

8.6.1 Motivation and Definition of Stack-Sharing Priority-Ceiling Protocol

A resource in the system is the run-time stack. Thus far, we have assumed that each job has its own run-time stack. Sometimes, especially in systems where the number of jobs is large, it may be necessary for the jobs to share a common run-time stack, in order to reduce overall memory demand. (According to Baker [Bake91], if there are 10 jobs at each of 10 priority levels, the storage saving is 90 percent.) Space in the (shared) stack is allocated to jobs contiguously in the last-in-first-out manner. When a job J executes, its stack space is on the top of the stack. The space is freed when the job completes. When J is preempted, the preempting job has the stack space above J's. J can resume execution only after all the jobs holding stack space above its space complete, free their stack spaces, and leave J's stack space on the top of the stack again.

Clearly, if all the jobs share a common stack, schedules such as the one in Figure 8–10 should not be allowed. You can see that if the jobs were to execute according this schedule, J_5 would resume after J_4 is blocked. Since J_4 is not complete, it still holds the space on the top of the stack at this time. The stack space of J_5 would be noncontiguous after this time, which is not allowed, or J_5 would not be allowed to resume, which would result in a deadlock between J_5 and J_4 (i.e., J_5 holds *Black* and avoidance blocks J_4 and J_4 holds the "top of the stack" and blocks J_5).

From this example, we see that to ensure deadlock-free sharing of the run-time stack among jobs, we must ensure that no job is ever blocked because it is denied some resource once its execution begins. This observation leads to the following modified version of the priority-ceiling protocol, called the *stack-based, priority-ceiling protocol*. It is essentially the same as the stack-based protocol designed by Baker [Bake91], which we will describe in the next section. Like Baker's protocol, *this protocol allows jobs to share the run-time stack if they never self-suspend*.

In the statement of the rules of the stack-based, priority-ceiling protocol, we again use the term (current) ceiling $\hat{\Pi}(t)$ of the system, which is the highest-priority ceiling of all the resources that are in use at time t. Ω is a nonexisting priority level that is lower than the lowest priority of all jobs. The current ceiling is Ω when all resources are free.

Rules Defining Basic Stack-Based, Priority-Ceiling Protocol

0. *Update of the Current Ceiling*: Whenever all the resources are free, the ceiling of the system is Ω. The ceiling $\hat{\Pi}(t)$ is updated each time a resource is allocated or freed.

1. *Scheduling Rule*: After a job is released, it is blocked from starting execution until its assigned priority is higher than the current ceiling $\hat{\Pi}(t)$ of the system. At all times, jobs that are not blocked are scheduled on the processor in a priority-driven, preemptive manner according to their assigned priorities.

2. *Allocation Rule*: Whenever a job requests a resource, it is allocated the resource.

We note that according to the scheduling rule, when a job begins to execute, all the resources it will ever need during its execution are free. (Otherwise, if one of the resources it will need is not free, the ceiling of the system is equal to or higher than its priority.) This is why

the allocation rule is as simple as stated above. More importantly, *no job is ever blocked once its execution begins*. Likewise, when a job J is preempted, all the resources the preempting job will require are free, ensuring that the preempting job can always complete so J can resume. Consequently, *deadlock can never occur*.

The schedule in Figure 8–17 shows how the system of jobs in Figure 8–10 would be scheduled if the stack-based, priority-ceiling protocol were used instead of the basic priority-ceiling protocol. To better illustrate the stack-based protocol, we let J_2 be released at 4.8 and the execution time of the critical section of J_2 be 1.2. At time 2 when J_4 is released, it is blocked from starting because its priority is not higher than the ceiling of the system, which is equal to 2 at the time. This allows J_5 to continue execution. For the same reason, J_3 does not start execution when it is released. When J_2 is released at time 4.8, it cannot start execution because the ceiling of the system is 2. At time 5, the resource held by J_5 becomes free and the ceiling of the system is at Ω. Consequently, J_2 starts to execute since it has the highest priority among all the jobs ready at the time. As expected, when it requests the resource *Black* at time 6, the resource is free. It acquires the resource and continues to execute. At time 7 when J_1 is released, its priority is higher than the ceiling of the system, which is 2 at the time. (Again, this fact indicates that the resource *Shaded*, which it will require later, is free.) J_1, therefore, preempts J_2 and holds the space on the top of the stack until it completes at time 10. J_2 then resumes and completes at 11. Afterwards, J_3, J_4, and J_5 complete in the order of their priorities.

From this example, we see that the scheduling rule of the stack-based priority-ceiling protocol achieves the same objective as the more complicated priority-inheritance rule of the basic priority-ceiling protocol. (As a consequence of this rule, J_5 is not preempted by J_4 and J_3 while it holds *Black*.) When we compare the schedule in Figure 8–17 with the schedule in Figure 8–10, which is produced by the basic priority-ceiling protocol, we see that the higher-

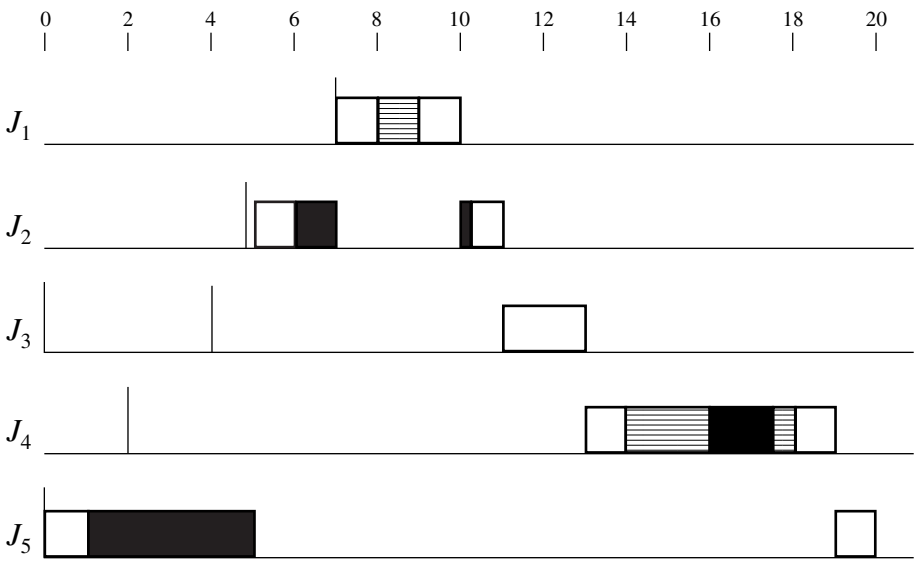


FIGURE 8–17 Schedule illustrating the stack-based, priority-ceiling protocol.

priority jobs J_1, J_2 and J_3 either complete earlier than or at the same time as when they are scheduled according to the basic priority-ceiling protocol.

8.6.2 Definition of Ceiling-Priority Protocol

As we will see shortly, the worst-case performance of the stack-based and the basic priority-ceiling protocols are the same. The former is considerably simpler and has a lower context switching overhead. This is a good reason for using the stack-based version if jobs never self-suspend even when the jobs have their own stacks. Indeed, the stack-based version is the protocol supported by the Real-Time Systems Annex of Ada95 [Cohe96]. In Section 8.3, we mentioned that it is called the *ceiling-priority protocol*. The following rules defines it more formally.

Rules Defining the Ceiling-Priority Protocol

1. *Scheduling Rule*:
   - **(a)** Every job executes at its assigned priority when it does not hold any resource. Jobs of the same priority are scheduled on the FIFO basis.
   - **(b)** The priority of each job holding any resource is equal to the highest of the priority ceilings of all resources held by the job.
2. *Allocation Rule*: Whenever a job requests a resource, it is allocated the resource.

We note that when jobs never self-suspend, the stack-based priority-ceiling and ceiling-priority protocols are the same. By saying these protocols are the same we mean that they will produce the same schedule for all jobs. (If you are not convinced, you may want to apply this set of rules on the example given by Figure 8–8(a). The schedule you will produce should be same as one in Figure 8–17.) *The two sets of rules give two implementations of the same protocol.*

Sometimes, jobs of equal priority are scheduled on the round-robin basis. We must modify the definition of the ceiling-priority protocol to make it work for these jobs. This modification is left as an exercise to you in Problem 8.5.

8.6.3 Blocking Time and Context-Switching Overhead

Because of the following theorem [Bake91], we can use the same method to find the blocking time $b_i(rc)$ of every job J_i for both versions of the priority-ceiling protocol.

> **THEOREM 8.3.** The longest blocking time suffered by every job is the same for the stack-based and basic priority-ceiling protocols.

To see why this theorem is true, we observe first that a higher-priority job J_h can be blocked only by the currently executing job J_l under the stack-based priority-ceiling protocol. The reason is that if a job J_l can start to execute at t, its priority is higher than the ceiling of the system at t. This means that none of the resources in use at t are required by J_l or any higher-priority job. Furthermore, similar arguments allow us to conclude that under the stack-based protocol, no job is ever blocked due to resource conflict more than once.

Now let us suppose that under the stack-based priority-ceiling protocol, a job J_h is blocked by the currently executing job J_l. This can happen only if J_h is released after J_l has acquired a resource X whose priority ceiling is equal to or higher than the priority π_h of J_h and at the time when J_h is released, J_l still holds X. The occurrence of this sequence of events is not dependent on the protocol used; the sequence can also occur under the basic priority-ceiling protocol. Moreover, under this circumstance, J_h is also blocked by J_l under the basic priority-ceiling protocol. In the worst case, J_h is blocked for as long as the job J_l holds such a resource. Theorem 8.3 follows from the fact that we can repeat this argument for every lower-priority job which can block J_h.

While the (worst-case) blocking time of individual jobs and periodic tasks are the same for both versions of the priority-ceiling protocol, the context-switch overhead is smaller under the stack-based version. Because no job is ever blocked once its execution starts, no job ever suffers more than two context switches. In particular, to take into account the context-switching overhead in the schedulability analysis of a system of periodic tasks scheduled according to a fixed-priority algorithm and stack-based priority-ceiling protocol, we add $2CS$ to the execution time of every task.

8.7 USE OF PRIORITY-CEILING PROTOCOL IN DYNAMIC-PRIORITY SYSTEMS

While both versions of the priority-ceiling protocol are relatively simple to implement and perform well when periodic tasks are scheduled on a fixed-priority basis, it is another matter in a dynamic-priority system. In a dynamic-priority system, the priorities of the periodic tasks change with time while the resources required by each task remain constant. As a consequence, the priority ceilings of the resources may change with time.

As an example, let us look at the EDF schedule of two tasks $T_1 = (2, 0.9)$ and $T_2 = (5, 2.3)$ in Figure 6–4. In its first two periods (i.e., from time 0 to 4), T_1 has priority 1 while T_2 has priority 2, but from time 4 to 5, T_2 has priority 1 and T_1 has priority 2. Suppose that the task T_1 requires a resource X while T_2 does not. The priority ceiling of X is 1 from time 0 to 4 and becomes 2 from time 4 to 5, and so on.

For some dynamic systems, we can still use the priority-ceiling protocol to control resource accesses provided we update the priority ceiling of each resource and the ceiling of the system each time task priorities change. This is the approach taken by the dynamic-priority-ceiling protocol proposed by Chen and Lin [ChLi]. As it will become evident shortly, except for this update, the priority-ceiling protocol can be applied without modification in job-level fixed-priority systems. In such a system, the priorities of jobs, once assigned, remain fixed with respect to each other. In particular, the order in which jobs in the ready job queue are sorted among themselves does not alter each time a newly released job is inserted in the queue. This assumption is true for systems scheduled on the EDF and LIFO basis, for example.

8.7.1 Implementation of Priority-Ceiling Protocol in Dynamic-Priority Systems

One way to implement the basic priority-ceiling protocol in a job-level fixed-priority system is to update the priority ceilings of all resources whenever a new job is released. Specifically, when a new job is released, its priority relative to all the jobs in the ready queue is assigned according to the given dynamic-priority algorithm. Then, the priority ceilings of all the resources are updated based on the new priorities of the tasks, and the ceiling of the system is

updated based on the new priority ceilings of the resources. The new priority ceilings are used until they are updated again upon the next job release. Chen and Lin [ChLi] showed that the protocol remains effective (i.e., it prevents deadlock and transitive blocking and no job is ever blocked for longer than the length of one critical section) in a job-level fixed-priority system.

The example in Figure 8–18 illustrates the use of this protocol in an EDF system. The system shown here has three tasks: $T_1 = (0.5, 2.0, 0.2; [Black; 0.2])$, $T_2 = (3.0, 1.5; [Shaded; 0.7])$, and $T_1 = (5.0, 1.2; [Black; 1.0\ [Shaded; 0.4])$. The priority ceilings of the two resources *Black* and *Shaded* are updated at times 0, 0.5, 2.5, 3, 4.5, 5, 6, and so on. We use consecutive positive integers to denote the priorities of all the ready jobs, the highest priority being 1.[2] In the following description, we focus primarily on how priority ceilings of resources and ceiling of the systems are updated and leave to you many details on how the priority-ceiling protocol works to produce the schedule segment shown in this figure. To emphasize that the priority ceiling of a resource R_i may change with time, we denote it by $\Pi_t(R_i)$.

1. At time 0, there are only two ready jobs, $J_{2,1}$ and $J_{3,1}$. $J_{2,1}$ (and hence T_2) has priority 1 while T_3 has priority 2, the priority of $J_{3,1}$. The priority ceilings of *Black* and *Shaded* are 2 and 1, respectively. Since $J_{2,1}$ has a higher priority, it begins to execute. Because no resource is in use, the ceiling of the system is Ω. At time 0.3, $J_{2,1}$ acquires *Shaded*, and the ceiling of the system rises from Ω to 1, the priority ceiling of *Shaded*.
2. At time 0.5, $J_{1,1}$ is released, and it has a higher priority than $J_{2,1}$ and $J_{3,1}$. Now the priorities of T_1, T_2, and T_3 become 1, 2, and 3, respectively. The priority ceiling Π_t (*Black*) of *Black* is 1, the priority of $J_{1,1}$ and T_1. The priority ceiling Π_t (*Shaded*) of *Shaded* becomes 2 because the priority of $J_{2,1}$ and T_2 is now 2. The ceiling of the system based on these updated values is 2. For this reason, $J_{1,1}$ is granted the resource *Black*. The ceiling of the system is 1 until $J_{1,1}$ releases *Black* and completes at time 0.7. Afterwards, $J_{2,1}$ continues to execute, and the ceiling of the system is again 2. When $J_{2,1}$ completes at time 1.7, $J_{3,1}$ commences to execute and later acquires the resources as shown.
3. At time 2.5, $J_{1,2}$ is released. It has priority 1, while $J_{3,1}$ has priority 2. This update of task priorities leads to no change in priority ceilings of the resources. Since the ceiling of the system is at 1, $J_{1,2}$ becomes blocked at 2.5. At time 2.9, $J_{3,1}$ releases *Black*, and $J_{1,2}$ commences execution.
4. At time 3.0, only T_1 and T_2 have jobs ready for execution. Their priorities are 1 and 2, respectively. The priority ceilings of the resources remain unchanged until time 4.5.
5. At time 4.5, the new job $J_{1,3}$ of T_1 has a later deadline than $J_{2,2}$. (Again, T_3 has no ready job.) Hence, the priority of T_1 is 2 while the priority of T_2 becomes 1. This change in task priorities causes the priority ceilings of *Black* and *Shaded* to change to 2 and 1, respectively.
6. At time 5 when $J_{3,2}$ is released, it is the only job ready for execution at the time and hence has the highest priority. The priority ceilings of both resources are 1. These values remain until time 6.

[2]Each time a job is completed, it is removed from the ready job queue. Therefore, when priorities and priority-ceiling values are recomputed, some task may not have any job in the ready job queue. We can either ignore such a task or let it have the lowest (nonexisting) priority Ω until the next update instant. In this example, we choose the former for the sake of simplicity.

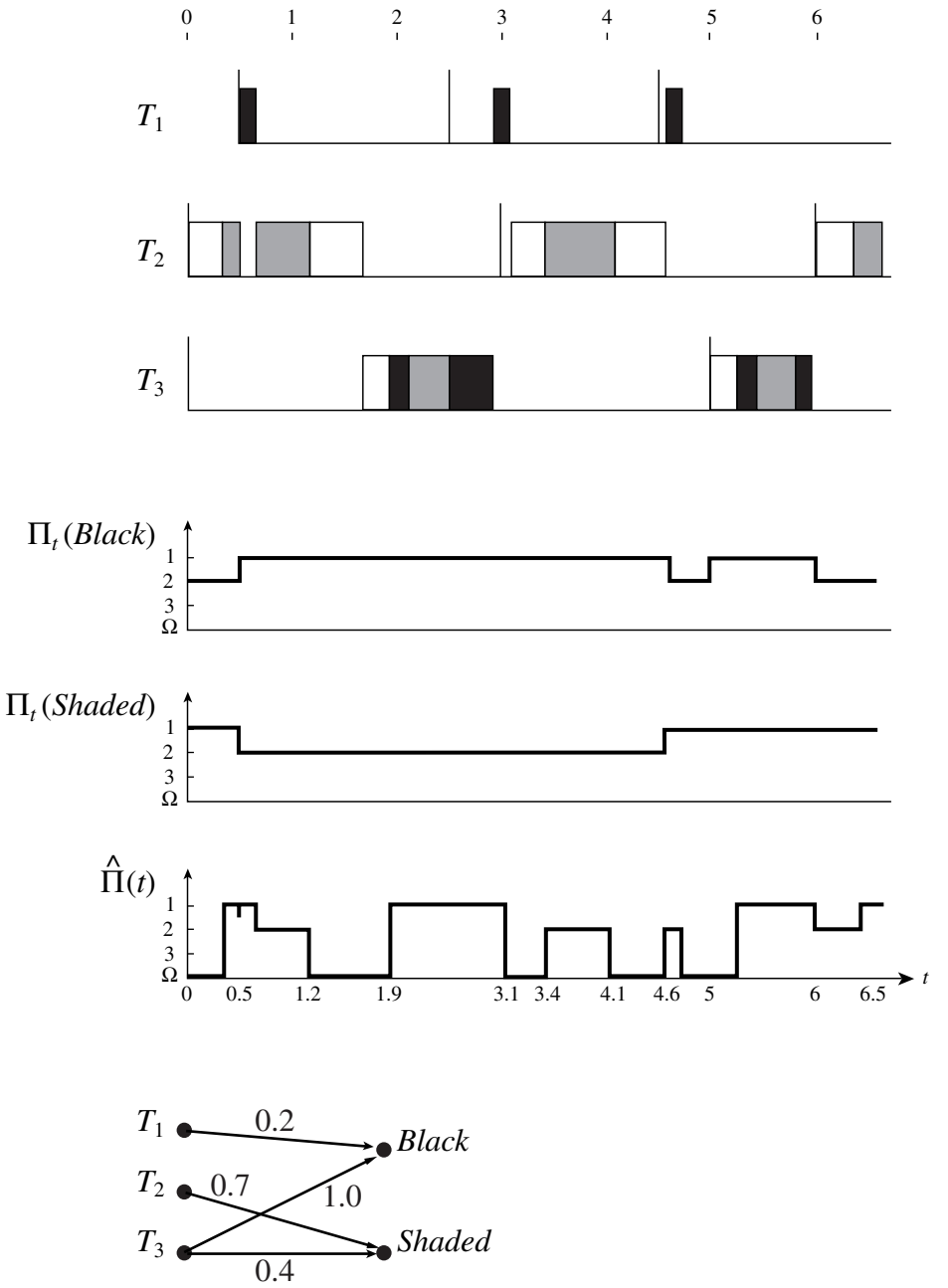


FIGURE 8–18 Example illustrating the use of the basic priority-ceiling protocol in an EDF system.

7. At time 6, both $J_{2,3}$ and $J_{3,2}$ are ready, and the former has an earlier deadline. We now have the same condition as at time 0.

In a system with ρ resources, each of which is required by $\overline{n}$ periodic tasks on average, the time required to update the priority ceilings of all resources is $O(\rho)$ each time a new job is released. This is a significant amount of overhead. For systems where the release-time jitters are negligible, we can save this run-time overhead by precomputing and storing the priority ceilings of all the resources for every time interval between consecutive releases of all N jobs in the entire hyperperiod. Each time a new job is released, we use the precomputed priority ceilings of the interval beginning from the release time of the job. The storage overhead for this purpose is $O(N\rho)$.

It is easy to see that the stack-based priority-ceiling protocol can also be used without modification for job-level fixed-priority assignments. Each time a new job is released, the question is whether its priority is sufficiently high for it to join the ready job queue.

8.7.2 Duration of Blocking

Although in principle every job is blocked at most once for the duration of one outermost critical section, the worst-case blocking time $b_i(rc)$ due to resource conflict of each task is in general larger in a dynamic-priority system than in a fixed-priority system. In particular, when it is possible for the jobs in every task to have a higher priority and preempt jobs in every other task, the worst-case blocking time $b_i(rc)$ of a task T_i is the execution time of the longest critical sections of all tasks other than T_i. For example, suppose that the jobs in Figure 8–15 are in six periodic tasks which are scheduled on the LIFO basis. The worst-case blocking times of all tasks except T_1 (i.e., the task containing J_1) are equal to 10, and the worst-case blocking time of T_1 is 6.

On the other hand, in a deadline-driven system, jobs with relative deadline D_i are never preempted by jobs with relative deadlines equal to or larger than D_i. Hence if the job J_i in Figure 8–16 belongs to task T_i, for $i = 1, 2, \ldots, 6$, and the tasks are indexed in order of increasing relative deadlines, then the worst-case blocking times $b_i(rc)$ are 6, 6, 5, 4, 4, and 0 when the tasks are scheduled in the EDF basis.

*8.7.3 Applicability of the Basic Priority-Ceiling Protocol in Job-Level Dynamic-Priority Systems

As discussed in Chapter 6, jobs do not have fixed priorities when scheduled according to some job-level dynamic-priority algorithm. As an example, we consider a system that contains three tasks $T_1 = (2, 1.0)$, $T_2 = (2.5, 1.0)$, and $T_3 = (0.8, 10, 0.5)$ and is scheduled according to the nonstrict LST algorithm. Initially, $J_{1,1}$ has priority 1 while $J_{2,1}$ has priority 2. However, at time 0.8 when $J_{3,1}$ is released and the slacks of all the jobs are updated, the slack of $J_{1,1}$ is still 1.0, but the slack of $J_{2,1}$ is only 0.7. Hence, $J_{2,1}$ has the highest priority, while the priority of $J_{1,1}$ drops to 2.

Another example is where the tasks are scheduled in a round-robin manner. This scheduling discipline can be implemented by giving the highest priority to jobs in turn, each for a fixed slice of time.

The allocation rule of the basic priority-ceiling protocol remains effective as a means to avoid deadlock and transitive blocking. To see that deadlock between two jobs J_i and J_k

can never occur, let us suppose that both jobs require X and Y. As soon as either X or Y is granted to one of the jobs, the ceiling of the system becomes π_i or π_k whichever is higher. As a consequence, it is no longer possible for the other job to acquire any resource. For this reason, it is not possible for J_i and J_k to circularly wait for one another, even though their priorities may change.

To see why it is not possible for any three jobs to block each other transitively, let us suppose that at time t, J_i is granted some resource X. Moreover, sometime afterwards, J_k is granted another resource Y. This is possible only when J_k does not require X. Therefore, it is not possible for J_k to be directly blocked by J_i, while it blocks some other job J_j.

A remaining question is whether priority inheritance is still effective in preventing uncontrolled priority inversion and keeping the delay suffered by every job due to priority inversion bounded from above. Clearly the statement of the priority-inheritance rule given in Section 8.4 needs to be modified in order to make sense. While the priority of the job J requesting and being denied a resource is higher than the priority of the job J_l blocking J at the time, the assigned priority of J_l may become higher than the priority of J at some later time. One way to take this fact into account is to let the blocking job J_l execute at the highest of the priority ceilings of all resources it holds.

8.8 PREEMPTION-CEILING PROTOCOL

We can avoid paying the time or storage overhead of the dynamic-priority-ceiling protocol described above for a class of dynamic-priority systems, which includes deadline-driven systems. We call systems in this class fixed preemption-level systems and will define this term shortly. For a fixed preemption-level system, Baker [Bake91] has a simpler approach to control resource accesses. The approach is based on the clever observation that the potentials of resource contentions in such a dynamic-priority system do not change with time, just as in fixed-priority systems, and hence can be analyzed statically. The observation is supported by the following facts:

1. The fact that a job J_h has a higher priority than another job J_l and they both require some resource does not imply that J_l can directly block J_h. This blocking can occur only when it is possible for J_h to preempt J_l.
2. For some dynamic priority assignments, it is possible to determine a priori the possibility that jobs in each periodic task will preempt the jobs in other periodic tasks.

Because of fact 1, when determining whether a free resource can be granted to a job, it is not necessary to be concerned with the resource requirements of all higher-priority jobs; only those that can preempt the job. Fact 2 means that for some dynamic priority systems, the possibility that each periodic task will preempt every other periodic task does not change with time, just as in fixed-priority systems. We have already seen that in a deadline-driven system, no job in a periodic task with a smaller relative deadline is ever preempted by jobs in periodic tasks with identical or larger relative deadlines, despite the fact that some jobs in the latter may have higher priorities.

8.8.1 Preemption Levels of Jobs and Periodic Tasks

The possibility that a job J_i will preempt another job is captured by the parameter *preemption level* ψ_i of J_i. The preemption levels of jobs are functions of their priorities and release times. According to a *valid preemption-level assignment*, for every pair of jobs J_i and J_k, the preemption level ψ_i of J_i being equal to or higher than the preemption level ψ_k of J_k implies that it is never possible for J_k to preempt J_i. Stated in another way,

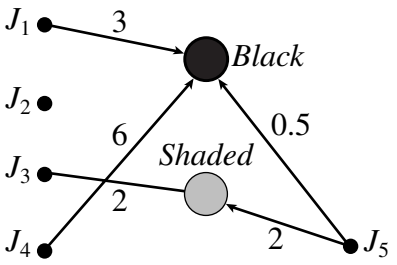
Validity Condition: If π_i is higher than π_k and $r_i > r_k$, then ψ_i is higher than ψ_k.

Given the priorities and release times of all jobs, this condition gives us a partial assignment of preemption levels, that is, the preemption levels of a subset of all jobs. The preemption levels of jobs that are not given by the above rule are valid as long as the linear order over all jobs defined by the preemption-level assignment does not violate the validity condition.

As an example, we return to the system of jobs in Figure 8–8. Since jobs with higher priorities are released later, it is possible for every job to preempt all the jobs with lower priorities than itself. In this case, the preemption levels of the jobs dictated by the the validity condition give us a complete assignment. For these jobs, we can simply let the preemption levels of the jobs be equal to their respective priorities.

Figure 8–19 gives another example. As usual, the five jobs are indexed in decreasing priorities. Their release times are such that $r_4 < r_5 < r_3 < r_1 < r_2$. We note that J_1, the job with the highest priority, has a later release time than J_3, J_4, and J_5. Hence, J_1 should have a higher preemption level than these three jobs. However, it is never possible for J_1 to preempt J_2 because J_1 has an earlier release time, and it is never possible for J_2 to preempt J_1, because J_2 has a lower priority. We therefore give these two jobs the same preemption level. Similarly, J_3 should have a higher preemption level than J_4 and J_5, and we can give J_4 and J_5 the same preemption level. In summary, we can assign ψ_i for $i = 1, 2, 3, 4$, and 5 the values 1, 1, 2, 3, and 3, respectively; it is easy see that this is a valid preemption level assignment. (Again, a smaller integer represents a higher preemption level.) Alternatively, we can assign preemption levels according to the release times of the jobs: the earlier the release time, the lower the preemption level. This assignment also satisfies the validity condition. The resultant preemption levels are 2, 1, 3, 5, and 4, respectively.

Let us now return to periodic tasks. When periodic tasks are scheduled on the EDF basis, a valid preemption-level assignment is according to the relative deadlines of jobs: the smaller the relative deadline, the higher the preemption level. (An assumption here is that



Jobs	r_i	e_i	π_i	ψ_i
J_1	5	5	1	1
J_2	7	3	2	1
J_3	4	4	3	2
J_4	0	9	4	3
J_5	3	4	5	3

FIGURE 8–19 A schedule according to the preeption-ceiling protocol.

either release-time jitters are negligible or the relative deadlines of all jobs remain fixed despite release-time jitters.) For this preemption-level assignment, all the jobs in every periodic task in a deadline-driven system have the same preemption level. This is an example of fixed preemption-level systems. A system of periodic tasks is a *fixed preemption-level system* if there is a valid assignment of preemption levels to all jobs such that all the jobs in every task have the same preemption level. Clearly, all fixed-priority systems are also fixed preemption-level systems. Indeed, an obvious preemption-level assignment in a fixed-priority system is to make the preemption level of each job equal to its priority.

When there is no chance of confusion, we call the preemption level of all the jobs in a fixed preemption-level task T_i the *preemption level of the task* and denote it by ψ_i. We index periodic tasks in a fixed preemption-level system according to their preemption levels: the higher the level, the smaller the index. For example, suppose that the system of periodic tasks in Figure 8–16 are scheduled on the EDF basis. The preemption levels of the tasks T_1, T_2, T_3, and T_4 are 1, 2, 3 and 4, respectively, because the relative deadlines of the tasks are 2, 2.2, 5, and 10, respectively.

In Chapter 6, we pointed out that when priorities are assigned on the FIFO basis or the LIFO basis, periodic tasks have dynamic priorities. In a FIFO system, no job is ever preempted. This is a degenerate fixed preemption-level system where all periodic tasks have the same preemption level. In contrast, periodic tasks scheduled on the LIFO basis have varying preemption levels. To illustrate, let us consider the system of three tasks T_1, T_2, and T_3 in Figure 8–18. Suppose that the priorities of jobs in them are assigned on the LIFO basis. $J_{1,1}$ is released later than $J_{2,1}$ and $J_{3,1}$, has a higher priority than they and hence can preempt these jobs. Also, $J_{2,2}$ can preempt $J_{1,2}$ and $J_{3,1}$, and $J_{3,2}$ can preempt $J_{1,3}$ and $J_{2,2}$. Suppose that we let the preemption level of each task at any time in a period equal the preemption level of the job released at the beginning of the period. For this system, we find that the preemption levels of the three tasks are 1, 2, and 3, respectively, in (0, 3); 2, 1, and 3 in (3, 4.5); 1, 2, and 3 in (4.5, 5); 2, 3, and 1 in (5, 6) and so on. In other words, this system has dynamic preemption levels.

8.8.2 Definitions of Protocols and Duration of Blocking

A *preemption-ceiling protocol* makes decisions on whether to grant a free resource to any job based on the preemption level of the job in a manner similar to the priority-ceiling protocol. This protocol also assumes that the resource requirements of all the jobs are known a priori. After assigning preemption levels to all the jobs, we determine the preemption ceiling of each resource. Specifically, when there is only 1 unit of each resource, which we assume is the case here, the *preemption ceiling* $\Psi(R)$ of a resource R is the highest preemption level of all the jobs that require the resource. For the example in Figure 8–19, the preemption ceiling of *Black* is 1, while the preemption ceiling of *Shaded* is 2.

The (*preemption*) *ceiling of the system* $\hat{\Psi}(t)$ at any time t is the highest preemption ceiling of all the resources that are in use at t. When the context is clear and there is no chance of confusion, we will simply refer to $\hat{\Psi}(t)$ as the ceiling of the system. We again use Ω to denote a preemption level that is lower than the lowest preemption level among all jobs since there is no possibility of confusion. When all the resources are free, we say that the ceiling of the system is Ω.

Like the priority-ceiling protocol, the preemption-ceiling protocol also has a basic version and a stack-based version. The former assumes that each job has its own stack and

the latter allows the jobs to share a common stack. Basic versions of priority-ceiling and preemption-ceiling protocols differ mainly in their allocation rules. For this reason, only the allocation rule of the basic preemption-ceiling protocol is given below. You can see that the principle of this rule for both protocols is the same, the only difference being the parameters (i.e., priority or preemption levels and ceilings) used by the rule.

Rules of Basic Preemption-Ceiling Protocol

1 and 3. The *scheduling rule* (i.e., rule 1) and *priority inheritance rule* (i.e., rule 3) are the same as the corresponding rules of the priority-ceiling protocol.

2. *Allocation Rule*: Whenever a job J requests resource R at time t, one of the following two conditions occurs:

(a) R is held by another job. J's request fails, and J becomes blocked.

(b) R is free.

(i) If J's preemption level $\psi(t)$ is higher than the current preemption ceiling $\hat{\Psi}(t)$ of the system, R is allocated to J.

(ii) If J's preemption level $\psi(t)$ is not higher than the ceiling $\hat{\Psi}(t)$ of the system, R is allocated to J only if J is the job holding the resource(s) whose preemption ceiling is equal to $\hat{\Psi}(t)$; otherwise, J's request is denied, and J becomes blocked.

The stack-based preemption-ceiling protocol is called the Stack-Based Protocol (SBP) by Baker [Bake91]. It is defined by the following rules. Rules 0, 1, and 2 are essentially the same as the corresponding rules of the stack-based priority-ceiling protocol; again, the difference is that priority levels/ceilings are replaced by preemption levels/ceilings. In addition, the stack-based preemption-ceiling protocol has an inheritance rule. We will show the need for this rule shortly.

Rules of Basic Stack-Based, Preemption-Ceiling Protocol

0. *Update of the Current Ceiling*: Whenever all the resources are free, the preemption ceiling of the system is Ω. The preemption ceiling $\hat{\Psi}(t)$ is updated each time a resource is allocated or freed.

1. *Scheduling Rule*: After a job is released, it is blocked from starting execution until its preemption level is higher than the current ceiling $\hat{\Psi}(t)$ of the system and the preemption level of the executing job. At any time t, jobs that are not blocked are scheduled on the processor in a priority-driven, preemptive manner according to their assigned priorities.

2. *Allocation Rule*: Whenever a job J requests for a resource R, it is allocated the resource.

3. *Priority-Inheritance Rule*: When some job is blocked from starting, the blocking job inherits the highest priority of all the blocked jobs.

Obviously, when the preemption levels of jobs are identical to their priorities, these versions of the preemption-level protocol are the same as the corresponding versions of the priority-ceiling protocol. For this reason, if the jobs in Figure 8–8 are scheduled according to the preemption-ceiling protocol, the resultant schedules are the same as those shown in Figures 8-10 and 8-17.

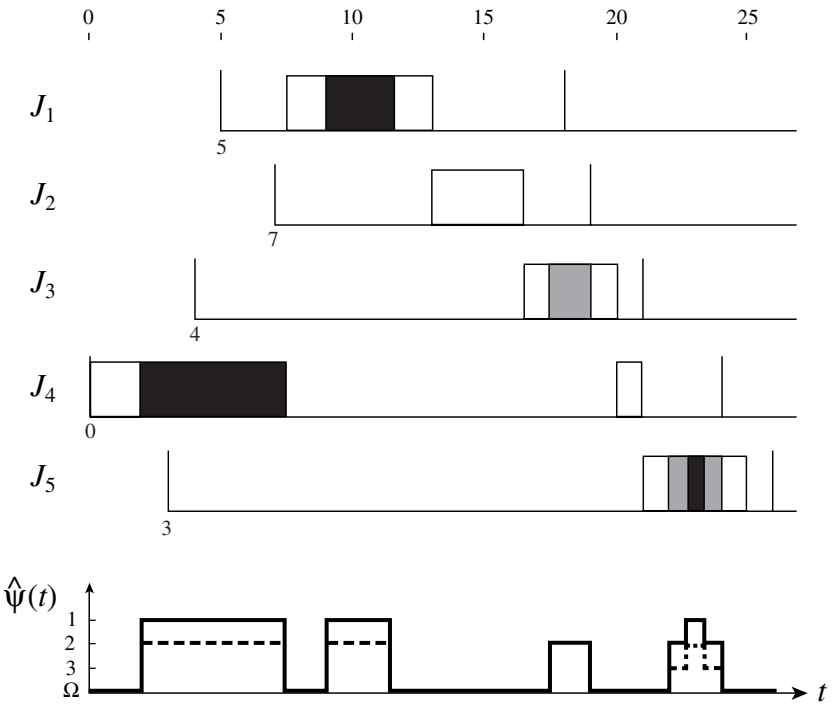


FIGURE 8–20 Example of priority ceilings of multiple-unit resources.

To see the necessity of the priority-inheritance rule of the stack-based preemption-ceiling protocol when preemption levels are assigned on the basis of jobs's relative deadlines, let us examine Figure 8–20. This figure gives an EDF schedule of jobs in Figure 8–19. (The deadline of each job is indicated by the second vertical bar on its time line. Specifically, the deadlines of J_1 and J_2 are at 18 and 18.5, respectively.) The preemption levels of the jobs are 2, 1, 3, 5 and 4, respectively, which are chosen based on the relative deadlines of the jobs.[3] The preemption ceiling of *Black* is 2, while the preemption ceiling of *Shaded* is 3. Hence, the preemption ceiling of the system is given by the dotted line in the graph. Because of the priority-inheritance rule, J_4 inherits the priority of J_1 when it blocks J_1 from starting execution at time 5. Consequently, J_4 can continue to execute and completes its critical section. Without this rule, J_1 would be blocked because of rule 1, but J_2, whose preemption level is 1, could start execution at time 7. Consequently, it would complete before J_4 completes its

[3]We note that the stack-based preemption-ceiling protocol without the priority-inheritance rule would produce the schedule in Figure 8–20 if the preemption levels of jobs are 1, 1, 2, 3, and 3, respectively. (The ceiling of the system for this set of preemption levels is given by the solid line in the graph below the schedule.) Nevertheless, we may not want to use this set of preemption levels and choose the set based on jobs' relative deadlines, despite the need for priority inheritance. You recall that this set of preemption levels was obtained by analyzing both the priorities and release times of the jobs. If each of these jobs is the first job of a periodic task, the preemption levels of the second jobs in the tasks containing J_1 and J_2, respectively, must be 2 and 1 in order to meet the validity condition. Now, undesirably, the preemption levels of these tasks vary with time.

critical section. In other words, without the inheritance rule, J_1 would be blocked first by J_4 and then by J_2. This uncontrolled priority inversion is prevented by the inheritance rule.

It is not surprising that both versions of the preemption-ceiling protocol prevent deadlock and transitive blocking and ensure that every job is blocked for at most the duration of one critical section. Like priority ceilings, preemption ceilings of resources impose an order on all the resources. The preemption ceiling $\hat{\Psi}(t)$ of the system tells us the subset of all jobs which we can safely grant available resources at time t. This subset contains all the jobs whose preemption levels are higher than the ceiling $\hat{\Psi}(t)$ of the system. Such a job J can be granted any resource R that is available at t because it does not require any resource that is in use at the time. Moreover, none of the jobs that are holding any resources at t can later preempt J.

8.9 CONTROLLING ACCESSES TO MULTIPLE-UNIT RESOURCES

Both versions of the priority-ceiling protocol and preemption-ceiling protocol described in the previous sections assume that there is only one unit of each resource. We now describe an extension to these protocols so that they can deal with the general case where there may be more than one unit of each resource (type).

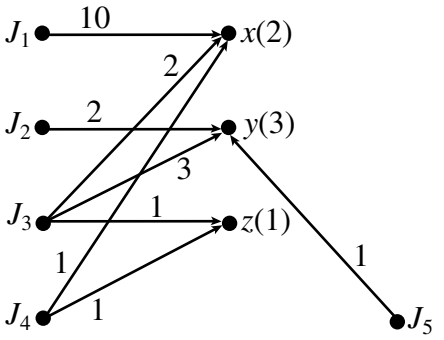
8.9.1 Priority (Preemption) Ceilings of Multiple-Unit Resources

The first step in extending the priority-ceiling protocol is to modify the definition of the priority ceilings of resources. We let $\Pi(R_i, k)$, for $k \leq v_i$, denote the priority ceiling of a resource R_i when k out of the v_i (≥ 1) units of R_i are free. If one or more jobs in the system require more than k units of R_i, $\Pi(R_i, k)$ is the highest priority of all these jobs. If no job requires more than k units of R_i, $\Pi(R_i, k)$ is equal to Ω, the nonexisting lowest priority. In this notation, the priority ceiling $\Pi(R_j)$ of a resource R_j that has only 1 unit is $\Pi(R_j, 0)$.

Let $k_i(t)$ denote the number of units of R_i that are free at time t. Because this number changes with time, the priority ceiling of R_i changes with time. The (current) priority ceiling of the system at time t is equal to the highest priority ceiling of all the resources at the time.

Figure 8–21 gives an example. The resource requirement graph gives the numbers of units of the resources X, Y, and Z required by the five jobs that are indexed in decreasing order of their priorities. The table below the graph gives the priority ceilings of each resource for different numbers of free resource units. For example, there are 2 units of X. When 1 unit of X is used, only J_3 is directly blocked. Therefore, $\Pi(X, 1)$ is π_3. J_1 is also directly blocked when both units of X are in use. For this reason, $\Pi(X, 0)$ is π_1, the higher priority between π_1 and π_3. When both units of X are free, the ceiling of the resource is Ω. Similarly, since J_2, J_3, and J_5 require 2, 3, and 1 unit of Y, which has 3 units, $\Pi(Y, 0)$, $\Pi(Y, 1)$, and $\Pi(Y, 2)$ are equal to π_2, π_2, and π_3, respectively. Suppose that at time t, 1 unit of each of X, Y, and Z is free. The priority ceilings of the resources are π_3, π_2, and Ω, respectively, and the priority ceiling of the system is π_2.

The preemption ceilings of resources that have multiple units can be defined in a similar manner: The preemption ceiling $\Psi(R_i, k)$ of the resource R_i when k units of R_i are free is the highest preemption level of all the jobs that require more than k units of R_i. Hence, if the jobs in Figure 8–21 were indexed in decreasing order according to their preemption levels and we replaced π_i and $\Pi(*, k)$ in the table by ψ_i and $\Psi(*, k)$, respectively, we would get the preemption ceilings of the three resources. The preemption ceiling of the system at time t is equal to the highest preemption ceiling of all the resources at the time.



Units Required by	Resources $x(2)$	$y(3)$	$z(1)$
J_1	1	0	0
J_2	0	2	0
J_3	2	3	1
J_4	1	0	1
J_5	0	1	0
$\Pi(*, 0)$	π_1	π_2	π_3
$\Pi(*, 1)$	π_3	π_2	Ω
$\Pi(*, 2)$	Ω	π_3	Ω
$\Pi(*, 3)$	Ω	Ω	Ω

FIGURE 8–21 Example of priority ceilings of multiple-unit resources.

8.9.2 Modified Rules

It is straightforward to modify the ceiling-priority protocol so it can deal with multiple-unit resources. In essence, the scheduling and allocation rules remains unchanged except for the new definition of priority ceiling of resources. However, since more than one job can hold (some units of) a resource, scheduling rule 1b needs to be rephrased for clarity. It should read as follows:

Scheduling Rule of Multiple-Unit Ceiling-Priority Protocol

Upon acquiring a resource R and leaving $k \geq 0$ free units of R, a job executes at the higher of its priority and the priority ceiling $\Pi(R, k)$ of R.

Similarly, the allocation rule of the priority-ceiling (or preemption-ceiling) protocol for multiple units of resources is a straightforward modification of the allocation rule of the basic priority-ceiling (preemption-ceiling) protocol.

Allocation Rule of Multiple-Unit Priority-(Preemption-) Ceiling Protocol

Whenever a job J requests k units of resource R at time t, one of the following two conditions occurs:

(a) Less than k units of R are free. J's request fails and J becomes directly blocked.

(b) k or more units of R are free.

(i) If J's priority $\pi(t)$ [preemption level $\psi(t)$] is higher than the current priority ceiling $\hat{\Pi}(t)$ [preemption ceiling $\hat{\Psi}(t)$] of the system at the time, k units of R are allocated to J until it releases them.

(ii) If J's priority $\pi(t)$ [preemption level $\psi(t)$] is not higher than the system ceiling $\hat{\Pi}(t)$ [$\hat{\Psi}(t)$], k units of R are allocated to J only if J holds the re-

source(s) whose priority ceiling (preemption ceiling) is equal to $\hat{\Pi}(t)$ [$\hat{\Psi}(t)$]; otherwise, J's request is denied, and J becomes blocked.

You can see that this rule is essentially the same as the allocation rule of the basic version. The only change is in the wording to accommodate multiple-unit requests.

8.9.3 Priority-Inheritance Rule

In the case where there is only 1 unit of each resource, we have shown that when a job J is blocked, only one lower-priority job is responsible for this blocking and this lower-priority job inherits J's priority. This may no longer be true in a system containing multiple resource units. More than one lower-priority job may be responsible. The question is which one of these jobs should inherit J's priority.

To illustrate, let us examine a system where there are 3 units of resource R, and there are four jobs, each requiring 1 unit of R. Suppose that at the time when the highest priority job J_1 requests a unit of R, all 3 units are held by the other three jobs. Now, all three lower-priority jobs block J_1. In this case, it is reasonable to let J_2 (i.e., the job with the highest priority among the three lower-priority jobs) inherit J_1's priority until it releases its units of R.

Indeed, an important special case is when a job can request and hold at most 1 unit of every resource at a time. (An example is counting semaphores.) In this case, the following priority-inheritance rule proposed by Chen [Chen] works well, that is, each job is blocked at most once for the duration of one critical section.

Priority-Inheritance Rule

When the requesting job J becomes blocked at t, the job with the highest priority among all the jobs holding the resource R that has the highest priority ceiling among all the resources inherits J's priority until it releases its unit of R.

In general, a job may request and hold arbitrary numbers of resources. The example in Figure 8–22 illustrates that a straightforward generalization of the priority-ceiling protocol and the above priority-inheritance rule ensures that each job is blocked at most once. The system in this example has five jobs indexed in decreasing order of their priorities. (In the description below, the priorities are 1, 2, 3, 4 and 5.) There are two resources *Black* and *Shaded*. The numbers of units are 5 and 1, respectively. The resource requirements of the jobs and priority ceilings of the resources are listed in Figure 8–22(a).

1. At time 0, J_5 starts to execute. When it requests 1 unit of *Black* at time 0.5, the ceiling of the system is Ω; therefore, it is granted 1 unit of *Black* and continues to execute. The ceiling of the system stays at Ω because there still are sufficient units of *Black* to meet the demand of every job.
2. At time 1, J_4 becomes ready. It preempts J_5 and, later, requests and is granted 1 unit of *Black*. Now, J_2 would be directly blocked if it requests *Black*, and the ceiling of *Black*, and consequently of the system, becomes 2, the priority of J_2.
3. At time 2, J_3 preempts J_4, and at time 2.5, J_2 preempts J_3. J_2 becomes blocked when it requests *Shaded* at time 3 because its priority is not higher than the ceiling of the system. J_4 now inherits priority 2 and executes.

	no. of units	units required					$\Pi(*, k), k =$					
		J_1	J_2	J_3	J_4	J_5	0	1	2	3	4	5
Black	5	2	4	0	1	1	1	1	2	2	Ω	Ω
S haded	1	1	1	0	0	1	1	Ω	Ω	Ω	Ω	Ω

(a)

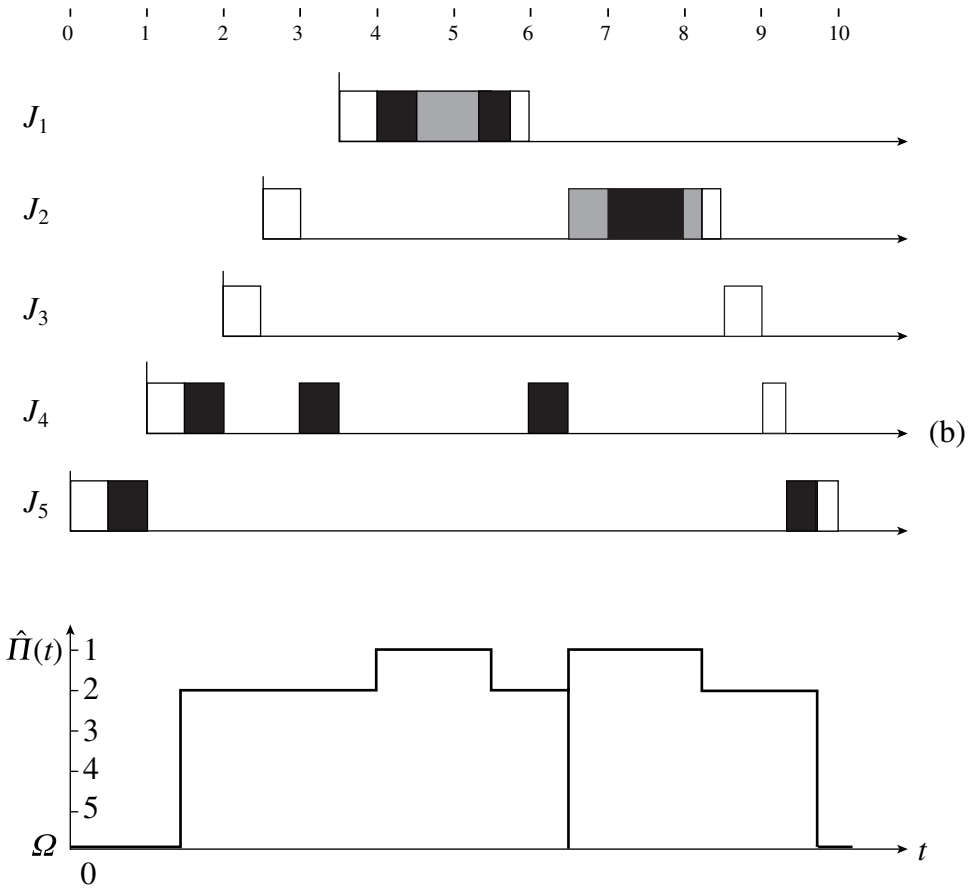


FIGURE 8–22 Example of multiple-unit resource access.

4. At time 3.5, J_1 preempts J_4. Since its priority is higher than the ceiling of the system, J_1 is allocated both resources when it requests them.
5. At time 6, J_1 completes, and J_4 continues to execute until it releases its 1 unit of *Black* at time 6.5. The ceiling of the system returning to Ω, J_2 is allocated *Shaded*. After *Shaded* is allocated, the ceiling of the system becomes 1.
6. At time 7, when J_2 requests 4 units of *Black*, the units are available. The ceiling of the system is 1, but J_2 holds the resource with this priority ceiling. Hence it is allocated 4 units of *Black*.
7. When J_2 completes, J_3 resumes. When J_3 completes, J_4 resumes, and it is followed by J_5. The system becomes idle at time 10.

From this example, we can see that Chen's priority-inheritance rule works for the general case as well. Specifically, we let R denote the resource whose priority ceiling is the highest among all resources in the system at time t. Let J' denote the job that has acquired the resource latest among all jobs holding R. If a requesting job J becomes blocked at t, J' inherits the priority of J. The fact that the priority-ceiling protocol defined by the rules stated in this section ensures that every job is blocked at most once can be proved in a manner similar to the proof of Theorem 8.2.

*8.10 CONTROLLING CONCURRENT ACCESSES TO DATA OBJECTS

Data objects are a special type of shared resources. When jobs are scheduled preemptively, their accesses to (i.e., reads and writes) data objects may be interleaved. To ensure data integrity, it is common to require that the reads and writes be serializable. A sequence of reads and writes by a set of jobs is *serializable* if the effect produced by the sequence on all the data objects shared by the jobs is the same as the effect produced by a serial sequence (i.e., the sequence of reads and writes when the jobs execute according to a nonpreemptive schedule).

8.10.1 Convex-Ceiling Protocol

The resource access-control protocols described in earlier sections do not ensure serializability. For example, both the NPCS and PC (Priority- and Preemption-Ceiling) protocols allow a higher-priority job J_h to read and write a data object X between two disjoint critical sections of a lower-priority job J_l during which J_l also reads and writes X. The value of X thus produced may not be the same as the value produced by either of the two possible serial sequences (i.e., all the reads and writes of J_l either proceed or follow that of J_h).

Motivation and Assumptions. A well-known way to ensure serializability is Two-Phase Locking (2PL). According to the 2PL protocol, a job never requests any lock once it releases some lock. Hence, the critical sections of J_1 and J_3 in Figure 8–1 satisfy this protocol, but the critical sections of J_2 do not. Under the 2PL protocol, J_2 would have to hold the locks on R_2 and R_3 until time 16. (This is because we also require the critical sections be properly nested.)

We can easily get concurrency-control protocols that not only ensure serializability but also prevent deadlock and transitive blocking by augmenting the protocols described in earlier

sections with the two-phase locking rule. As a result, we have the NPCS-2PL and the PCP-2PL protocols. The augmented protocols have an obvious shortcoming: prolonged blocking. Following the 2PL rule, a job may hold a data object even when it no longer require the object. As a consequence, it may block other jobs for a longer duration.

The convex-(priority-) ceiling protocol [Naka] described below is another extension of the priority-ceiling protocol. It is an improvement over the PCP-2PL protocol because it reduces the duration of blocking. In the description below, we assume that there is only one of each data object.

Priority-Ceiling Function. As with the PCP-2PL protocol, the convex-ceiling protocol assumes that the scheduler knows a priori the data objects require by each job and, therefore, the priority ceiling of each data object. In addition, each job notifies the scheduler immediately after it accesses each of its required objects for the last time. We call a notification sent by a job J_i after it accesses R_k for the last time the *last access notification* for R_k by J_i and the time of this notification the *last access time of* R_k *by* J_i.

For each job J_i in the system, the scheduler generates and maintains the following two functions: the *remainder priority ceiling*, $RP(J_i, t)$ and the *priority-ceiling function*, $\Pi(J_i, t)$. $RP(J_i, t)$ is the highest priority ceiling of all data objects that J_i will require after time t. When J_i is released, $RP(J_i, t)$ is equal to the highest priority ceiling of all data objects required by the job. The scheduler updates this function each time when it receives a last access notification from J_i. When the job no longer requires any object, its remainder priority ceiling is Ω.

When each job J_i starts execution, its priority-ceiling function $\Pi(J_i, t)$ is equal to Ω. When J_i is allowed to access an object R_k for the first time, $\Pi(J_i, t)$ is set to the priority ceiling $\Pi(R_k)$ of R_k if the current value of $\Pi(J_i, t)$ is lower than $\Pi(R_k)$. Upon receiving a last access notification from J_i, the scheduler first updates the function $RP(J_i, t)$. It then sets the priority-ceiling function $\Pi(J_i, t)$ of the job to $RP(J_i, t)$ if the remainder priority ceiling is lower.

Figure 8–23 gives an example. The job J_i requires three data objects: *Dotted*, *Black*, and *Shaded*. Their priority ceilings are 1, 2, and 3, respectively. Figure 8-23(a) shows the time intervals when the job executes and accesses the objects. The two functions of the job are shown in Figure 8–23(b). At time 0, $RP(J_i, 0)$ is 1. The job sends last access notifications at times 4, 6, and 8 when it no longer needs *Dotted*, *Black*, and *Shaded*, respectively. The scheduler updates $RP(J_i, t)$ at these instants; each time, it lowers the remainder priority ceiling to the highest priority ceiling of all objects still required by the job in the future.

Initially, $\Pi(J_i, t)$ is equal to Ω. At time 2 when J_i accesses *Black* for the first time, its priority ceiling function is raised to 2, the priority ceiling of *Black*. $\Pi(J_i, t)$ stays at 2 until time 3 and is raised to 1 at time 3 when J_i accesses *Dotted* for the first time. At time 4 when the last access notification for *Dotted* is received, $\Pi(J_i, t)$ is set to 2, the updated value of $RP(J_i, t)$. Similarly, $\Pi(J_i, t)$ is lowered to 3 and Ω at the last access times 6 and 8 of *Black* and *Shaded*, respectively.

By definition, $RP(J_i, t)$ is a nonincreasing function of t. The priority-ceiling function $\Pi(J_i, t)$ first raises as the job is allowed to access more and more data objects. Its value, once lowered, never rises again. In other words, the priority-ceiling function of every job is "two-phase"; it has only one peak.

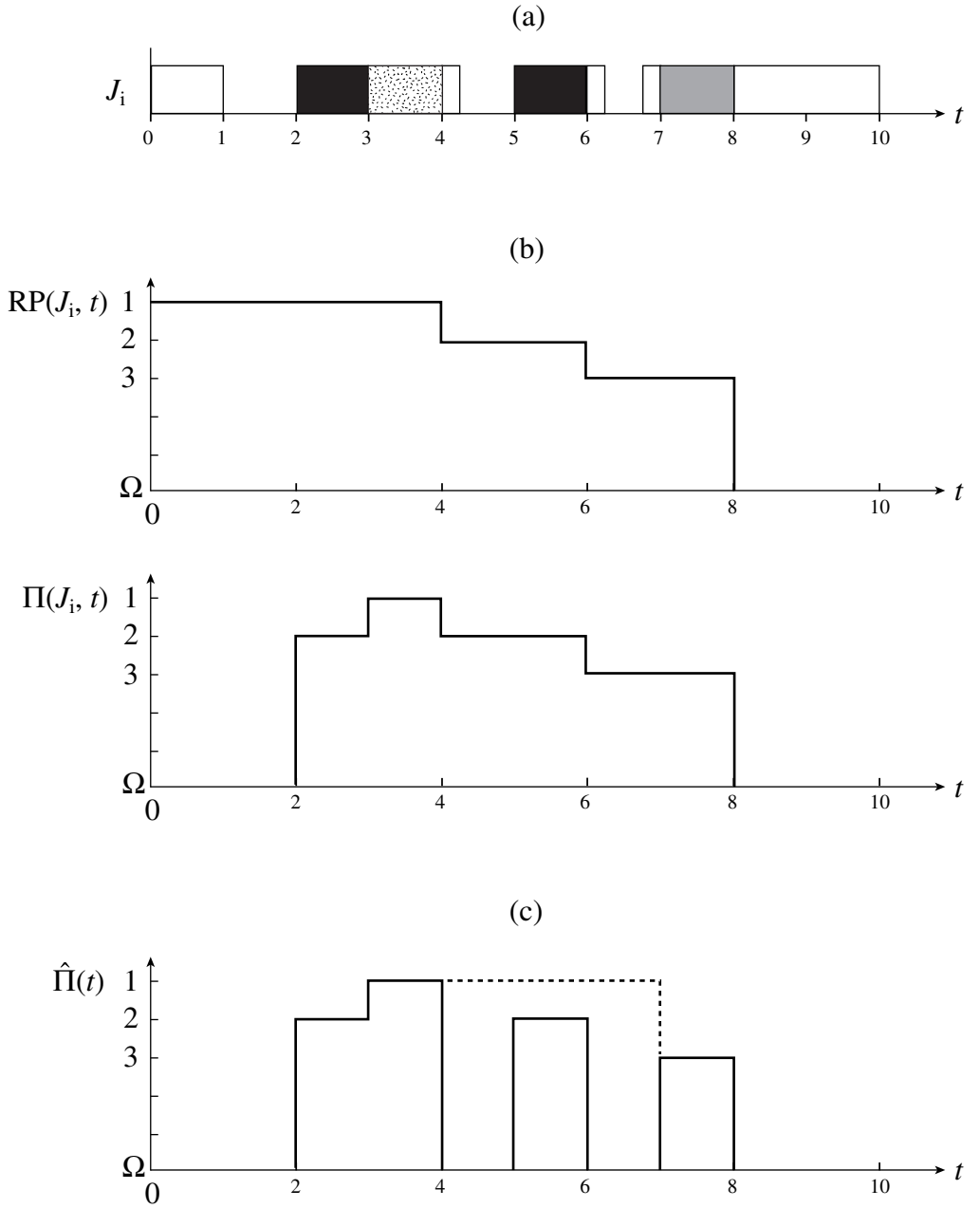


FIGURE 8–23 Example of priority-ceiling functions.

Definition and Capability. As with the priority-ceiling protocol, at any time t when the scheduler receives a request to access an object R for the first time from any job J, it computes the system ceiling $\hat{\Pi}(t)$. $\hat{\Pi}(t)$ is equal to the highest priority of the priority-ceiling functions of all the jobs in the system. The convex-ceiling protocol defined by the following rules.

Rules of Convex-Ceiling Protocol

1. *Scheduling Rule*: At any time, jobs that are not suspended are scheduled on the processor in a preemptive, priority-driven manner. Upon its release, the current priority of every job J_i is its assigned priority π_i. It executes at this priority except when the inheritance rule is applied.
2. *Allocation Rule*: When a job J_i requests to access a data object R for the first time,
   - **(a)** if J_i's priority is higher than the system ceiling $\hat{\Pi}(t)$, J is allowed to continue execution and access R;
   - **(b)** if J_i's priority is not higher than $\hat{\Pi}(t)$,
     - **i.** if $\hat{\Pi}(t)$ is equal to $\Pi(J_i, t)$, J_i is allowed to access R;
     - **ii.** otherwise, J is suspended.
3. *Priority-Inheritance Rule*: When J_i becomes suspended, the job J_l whose priority-ceiling function is equal to the system ceiling at the time inherits the current priority $\pi_i(t)$ of J_i

By definition, the priority-ceiling function of any job J (and hence the ceiling of the system) never falls below the priority ceiling of any data object already read and written by J until J no longer requires the object. In this way, the convex-ceiling protocol prevents other jobs from reading and writing the objects required by J between the reads and writes of J. The formal proof that the protocol ensures serializability can be found in [Naka].

In principle, the convex-ceiling protocol is the same as the basic priority-ceiling protocol. (The system ceiling used by the basic priority-ceiling protocol can also be defined as the highest priority of the priority-ceiling functions of all the jobs: The priority-ceiling function of a job is equal to the highest priority ceiling of all the resources being used by the job.) That the convex-ceiling protocol prevents deadlock and transitive blocking follows straightforwardly from this fact.

From its definition, we see that that the convex-ceiling protocol can be implemented entirely in the application level. An application-level object manager can serve as the scheduler. It maintains the priority-ceiling functions of jobs and controls the concurrent accesses to data objects. It can do so without relying on a locking mechanism. Since the basic priority-ceiling and PCP-2PL protocols can also be defined in the same manner, the same can be said for these protocols as well.

Comparison with Basic and PCP-2PL Protocols. To illustrate the differences between this protocol and the basic priority-ceiling protocol, we return to the example in Figure 8–23. Suppose that after J_i accesses *Black*, a job J_2 with priority 2 were to request access to a data object. According to the convex-ceiling protocol, this request would be denied and J_2 would be suspended until time 6 when the system ceiling is lowered to 3. In contrast, according to the basic priority-ceiling protocol, the system ceiling is given by the solid line in Figure

8–23(c). J_2 would be allowed to access its required data object starting from time 4. J_2 would be blocked for a shorter amount of time at the expense of potential violation of serializability.

On the other hand, according to the PCP-2PL protocol, the system ceiling would be given by the dotted line in Figure 8–23(c) because J_i does not release any data object until it no longer requires additional objects. As a consequence, J_2 would be blocked for 1 more unit of time when compared with the convex-ceiling protocol. A job with priority 1 can be blocked by J_i for only 1 unit of time under the convex-ceiling protocol, but can be blocked for 5 units of time under PCP-2PL protocol.

8.10.2 Other Real-Time Concurrency Control Schemes

One way to improve the responsiveness of soft real-time jobs that read and write multiple data objects is to abort and restart the lower-priority job whenever it conflicts (i.e., contends for some data object) with a higher-priority job. A policy that governs which job proceeds at the expense of which job is called a *conflict resolution policy*.

Well-Known Conflict Resolution Policies. To explain conflict resolution policies that have been proposed for database transactions, we note that each transaction typically keeps a copy of each object it reads and may write in its own memory space. When it completes all the reads, writes, and computations, it writes all data objects it has modified back to the global space. This last step is called commit. So, until a transaction commits, no shared data object in the database is modified, and it is safe to abort and restart a transaction and take back the data objects allocated to it.

Abbott, *et al.* [AbMo] showed that the 2PL-HP schemes perform well for soft real-time transactions compared with Optimistic Concurrency Control (OCC) schemes, which we will explain shortly. According to the 2PL-HP scheme, all transactions follow a two-phase policy in their acquisitions of (locks of) data objects. Whenever two transactions conflict (i.e., they want to access the same object), the lower-priority transaction is restarted immediately (i.e., the executing instance is aborted and new instance is made ready immediately). In essence, this scheme allocates data objects to transactions preemptively. Therefore, priority inversion cannot occur.

The results of a soft real-time transaction remains useful to some extent even when the transaction completes late. In contrast, the result of a late transaction with a firm deadline is useless and hence is discarded. The 2PL-HP scheme does not work well for firm real-time transactions compared with OCC schemes. The reason is that 2PL-HP can cause wasted restart and wasted wait [HoJC]. The former refers to a restart of a lower-priority transaction which turns out to be unnecessary because the conflicting higher-priority transaction completes late. A way to reduce wasted restarts is by simply suspending a conflicting lower-priority transaction when a conflict occurs. If the conflicting higher-priority transaction completes in time, the suspended transaction is aborted and restarted. On the other hand, if the conflicting transaction completes late, the suspended transaction can resume. Unfortunately, the stop and wait may cause the lower-priority transaction to be late, and thus incurs wasted wait.

Optimistic Concurrency Control Schemes. Optimistic concurrency control is an alternative approach to two-phase locking. Under the control of an OCC scheme, whether a transaction conflict with other executing transactions is checked immediately before the transaction commits. This step is called validation. If the transaction is found to conflict with other

transactions at the time, one of them is allowed to proceed and commit, while the conflicting transactions are restarted. Different OCC schemes differ in the choices of these transactions. A priority-based OCC scheme allows a conflicting lower-priority scheme to proceed until validation time and then is restarted if it is found to conflict with some higher-priority transaction at validation time.

Most performance evaluation studies (e.g., [HaCL, LeSo] found that OCC schemes perform better than 2PL-based schemes in terms of on-time completions of real-time transactions. However, when the performance criterion is temporal consistency, OCC schemes tend to give poorer performance, especially when the transactions are periodic. In particular, Song, *et al.* [SoLi] found that the age and dispersion of data read by transactions tend to be larger under an OCC scheme than under lock-based schemes. (The terms temporal consistency, age and dispersion were defined in Section 1.4.1.) This is because both blocking and preemption can cause temporal inconsistency (i.e., age and dispersion of data being large). Blocking due to resource contention can cause higher-priority update transactions to complete late and data to be old. However, preempted transactions may read data that are old and have large age dispersion as well. When transactions are periodic, a transaction restarted in one period is likely to restart again in other periods. The repeated restarts lead to a large variance in the age and dispersion of data.

8.11 SUMMARY

This chapter described several protocols for controlling accesses to shared resources in systems where jobs are scheduled on a processor in a preemptive, priority-driven manner. This section first summarizes them and then discusses the effect of resource contention with aperiodic jobs and rules for safe mode changes.

8.11.1 Resource Access-Control Protocols

Because every unit of resource can be used by at most one job at a time without preemption, priority inversions may occur. The primary design objective of resource access control protocols is to control priority inversion, that is, to keep the duration of every priority inversion (and hence the blocking time suffered by the higher-priority job) bounded from above. Among the protocols described above,

1. the NPCS and priority-inheritance protocols do not require prior knowledge on the resources required by jobs, while all other protocols do;
2. the priority-inheritance protocol does not prevent deadlock, while all other protocols do; and
3. all protocols except the priority-inheritance protocol ensures that every job is blocked at most once.

Nonpreemptive Critical Section (NPCS) Protocol. The NPCS protocol described in Section 8.3 offers the simplest means to ensure that every job is blocked at most once if it does not self-suspend. According to this protocol, the scheduler allows the executing job to use any resource at any time and never preempts a job when the job is using any resource.

A job can be blocked only if it becomes ready when some lower-priority job is in a critical section (i.e., using some resource). The blocking time thus incurred is at most equal to the execution time of the critical section. Simplicity is the major advantage of this protocol. Its disadvantage is that every job can be blocked by all lower-priority jobs.

Priority-Inheritance Protocol. According to the priority-inheritance protocol described in Section 8.4, the request for a resource is denied and the requesting job becomes blocked only if the resource is in use. When this occurs, the job that directly blocks the requesting job (i.e., is using the requested resource) inherits (i.e., executes at) the requesting job's priority. Because of priority inheritance, a job may also be priority-inheritance blocked when a lower-priority job inherits a higher priority. The priority-inheritance protocol keeps the blocking time bound from above by the maximum execution time of critical sections of lower-priority jobs each time a job is blocked. However, a job that conflicts with more than one job over more than one resource can be blocked a multiple number of times. Again, this protocol does not prevent deadlock.

Basic Priority-Ceiling Protocol. Section 8.5 described a version of the priority-ceiling protocol. This protocol prevents deadlock and transitive blocking among jobs. When it is used, a job that never self-suspends is blocked at most once, and the duration of blocking is bounded from above by the maximum execution time of critical sections of lower-priority jobs.

The scheduler finds a priori the priority ceiling of each resource (i.e., the highest priority of all jobs that require the resource). When a job J requests a resource, say at time t, the scheduler first finds the system ceiling $\hat{\Pi}(t)$, which is the highest of the priority ceilings of all resources in use at the time. The requesting job can use the resource only when the job's priority is higher than the system ceiling or when the job has the resource whose priority ceiling is equal to $\hat{\Pi}(t)$. When the requesting job becomes blocked, the job holding the resource with priority ceiling $\hat{\Pi}(t)$ inherits the priority of the requesting job.

The priority-ceiling protocol is nongreedy: a requesting job may be denied a resource and becomes priority-ceiling blocked even when the resource is free. Every time a job requests a resource, the scheduler must update the system ceiling. The complexity of this operation is $O(\log \rho)$ where ρ is the number of resource types. The priority-ceiling protocol is better suited for a fixed-priority system of periodic tasks than dynamic-priority systems.

Ceiling-Priority (Stack-Based Priority-Ceiling) Protocol. According to the ceiling-priority protocol, which is identical to the stack-based version of the priority-ceiling protocol, each job executes at its assigned priority when it does not hold any resource, and at the highest of priority ceilings of all resources held by the job whenever the job holds any resource. Another way to implement the protocol is to have the scheduler update the system ceiling each time the status (i.e., in use or free) of a resource changes. Upon release, a job whose priority is no higher than the system ceiling is blocked until the system ceiling becomes lower than its priority. Once a job is unblocked and scheduled for execution, it is never blocked because all its required resources are available.

The stack-based version of the priority-ceiling protocol is described in Section 8.6. It allows the jobs to share the run-time stack if the jobs never self-suspend. The stack-based version has the same worst-case blocking time as the basic version. It is much simpler to

implement. For this reason, it is preferred over the basic priority-ceiling protocol even for jobs that do not share stack space.

Preemption-Ceiling Protocols. In principle, the priority-ceiling protocol can be used in a dynamic-priority system. However, the high overhead due to the updates of priority ceilings of resources upon the releases of new jobs makes it unattractive. The preemption-ceiling protocol described in Section 8.8 is a better alternative. The protocol is motived by the observation that the potential for a job to preempt other jobs depends not only on its priority but also on its release time. Therefore, rather than than working with priorities of jobs, the preemption-ceiling protocol assigns each job a preemption level based on its priority and release time: the higher the preemption level of a job, the larger the possibility of its preempting other jobs. So, a job that has a higher priority and later release time should have a higher preemption level. For example, the preemption levels of jobs in a deadline-driven system can be assigned on the basis of their relative deadlines: the shorter the deadline, the higher the preemption level.

The preemption ceiling of a resource is the highest preemption level of all jobs that require the resource. The basic and stack-based versions of the preemption-ceiling protocol are essentially the same as the corresponding versions of the priority-ceiling protocol except that in place of priority ceilings, the schedulers works with the preemption ceiling of resources.

Periodic tasks in many dynamic-priority systems (e.g., those scheduled on the EDF or LST basis) have fixed preemption levels. The preemption-ceiling protocol is a better option because it should have lower run-time overhead than the priority-ceiling protocol.

Priority- and Preemption-Ceiling Protocols for Multiple-Unit Resources. The priority-ceiling (and preemption-ceiling) protocols were extended in Section 8.9 so they can deal with multiple-unit resources. The priority ceiling $\Pi(R, k)$ of a resource R that has $\nu \geq 1$ units when k ($\nu \geq k \geq 0$) are available is the highest priority of all jobs that require more than k units of R. The system ceiling at any time is equal to the highest of priority ceilings of all resources in the system. Except for this modification, the multiple-unit priority-ceiling protocol is the same as the basic version.

Implementation Issue. The most straightforward implementation of the priority-ceiling (or preemption-ceiling) protocol makes use of a locking mechanism: A job locks a (unit of a) resource when the scheduler allows it to use the resource and releases the lock to free the resource. The definition of the protocol makes this assumption.

This protocol can also be implemented without relying a locking mechanism. An implementation without locking requires that each job notifies the scheduler when it is no longer requires a resource, in addition to requesting the use of a resource. The scheduler keeps a priority-ceiling function $\Pi(J, t)$ for each job J. At any time t, $\Pi(J, t)$ is the highest priority ceiling of all resources which J is allowed to use at time t. This function is updated each time the scheduler allows J to access a resource and when a notification from the job is received. The system ceiling at time t is equal to the highest ceiling of priority-ceiling functions of all jobs at time t. You can find in Section 8.10.1 reworded allocation and inheritance rules of the protocol that assume the information used by the scheduler is kept in this form and jobs do not lock resources.

Protocols for Controlling Accesses to Data Objects. Section 8.10 discussed the additional concern about serializability when jobs share data. A way to ensure serializable accesses to data objects is to augment the priority-ceiling and preemption-ceiling protocols with a two-phase locking rule. The augmented protocol is called the PCP-2PL protocol. This is also the basic principle of the convex-ceiling protocol described in Section 8.10.1. This protocol improves on the PCP-2PL protocol by reducing the blocking times. Optimistic concurrency control are well-known alternatives to locking-based protocols described above.

Computing the Total Blocking Time. Finally, let us denote the number of times each job J_i self-suspends by K. Moreover, each job J_i has resource conflict over different resources with v_r lower-priority jobs and v_n lower-priority jobs have nonpreemption segments. Let $b_i(ss)$, $b_i(rc)$, and $b_i(np)$ denote the blocking times of J_i due to self-suspension, resource contention, and nonpreemptivity, respectively. Then the total blocking time b_i suffered by J_i is given by

$$b_i = b_{i,k}(ss) + (K+1)\max(b_i(rc), b_i(np))$$

$$b_i = b_{i,k}(ss) + \max(K+1, v_r)b_i(rc) + \min(K+1, v_n)b_i(np)$$

$$b_i = b_{i,k}(ss) + (K+1)b_i(rc) + (K+1)b_i(np)$$

when resource accesses are controlled according to the NPCS, priority-inheritance, and priority-ceiling (or preemption-ceiling) protocols, respectively.

8.11.2 Resource Contention with Aperiodic Jobs

Aperiodic (and sporadic) jobs may also use resources. Their resource requirements must be taken into account when we compute the blocking time due to resource contention of any job (or task). The techniques for computing blocking time described earlier can be used for this purpose as well, provided that an aperiodic job is never suspended while it is in a critical section because the server executing it runs out of budget.

To prevent prolonged blocking, the deferrable and sporadic server algorithms must be modified as follows so the server will not be suspended because it runs out of budget when the aperiodic job being executed is holding some resource. (1) When the server exhausts its budget, the scheduler suspends the server immediately if the aperiodic job being executed is not in a critical section, but if the job is in a critical section, the scheduler waits until the job exits from the critical section and then suspends the server. In other words, the server is nonpreemptable while it is in a critical section. (2) Whenever the scheduler has allowed the server to execute after the server budget is exhausted (i.e., allows the server to overrun its allocated time), the scheduler deducts the length of the overrun from the server budget when it replenishes the server budget. As a consequence of this overrun, a lower-priority task may suffer additional delay. When we compute the time demand of the lower-priority task, we need to add the execution time of the longest critical section of the aperiodic task as a blocking term into the time demand function [e.g., given by the expression in Eq. (7.1)] of the task.

An advantage of a total bandwidth, constant utilization, or weighted fair-queueing server is that such a server never runs out of budget before the completion of the aperiodic job being executed. Consequently, when some job in the aperiodic task executed by such a server requires resources, we can simply treat the aperiodic task in the same way as a periodic task in

our computation of blocking time. A slight complication is the choice of preemption level for the server in a system that uses a preemption-ceiling protocol, since the "relative deadline" of the server changes with time. The server should be given a high preemption level for the sake of the system's aperiodic response. If the server has a low preemption level (e.g., based on the longest relative deadline it may have), it in effect executes in the background of periodic tasks.

8.11.3 Mode Changes

We conclude this chapter by returning to the question that was last discussed in Section 5.7.2: how to carry out mode change. In other words, when it is safe to add new (periodic and sporadic) tasks into a priority-driven system and delete existing tasks from it. Earlier chapters mentioned admissions of tasks after they pass acceptance, but do not explicitly say when. During a mode change, that is, when the system changes from one operation mode to another, not only *new tasks* (i.e., tasks that do not run in the current mode but will run in the new mode) need to be added, but also *old tasks* (i.e., tasks that run in the current mode but will not run in the new mode) need to be deleted. As we did in Section 5.7.2, we focus here on periodic tasks. This subsection provides a brief summary of rules governing the deletion and addition of these tasks; you can find details on mode-change protocols in [SRLR].

Task Deletions and Admissions. In the simpler case, tasks' resource accesses are controlled by the NPCP and priority-inheritance protocols. The scheduler does not maintain, and therefore does not need to update, resource usage information. In the task deletion phase, the scheduler deletes every old task T and reclaims the processor time allocated to the task. Specifically, it deletes T immediately if at the start t of the mode change, the current job of T has not begun to execute. However, it waits to do so until the beginning of the next period of T or when the current job completes, whichever is later, if the job has executed at time t. Therefore, the length of time the scheduler takes to complete the task-deletion phase is at most equal to the longest of the periods of all old tasks.

In principle, new tasks can be added one by one as sufficient processor time becomes available and the new task becomes schedulable with existing tasks. This does not speed up the completion of mode change, however, and it is much simpler for the scheduler to wait until the task-deletion phase completes and then adds all the new tasks at once.

Updates of Resource Usage Information. In a system that uses priority-ceiling, ceiling-priority, or preemption-ceiling protocols, priority (preemption) ceilings of resources must be updated before new tasks can be added into the system. The scheduler raises the priority ceiling of a resource whose new priority ceiling (i.e., the ceiling in the new mode) is higher than the old after the start t of the mode change as soon as the resource is free. It waits until all the tasks that use a resource R whose new priority ceiling is lower than the old have been deleted and then lowers the priority ceiling of R to the new value.

A new task that will use a resource R whose priority ceiling must be raised can be added into the system only after the priority ceiling of R has been raised. All the updates of priority (or preemption) ceilings are surely complete shortly after the task-deletion phase completes or all the resources are released, whichever is later.

EXERCISES

8.1 A system contains five jobs. There are three resources X, Y, and Z. The resource requirements of the jobs are listed below.

J_1: $[X; 2]$
J_2: none
J_3: $[Y; 1]$
J_4: $[X; 3\,[Z; 1]]$
J_5: $[Y; 4\,[Z; 2]]$

The priority J_i is higher than the priority of J_j for $i < j$. What are the maximum blocking times of the jobs under the nonpreemptable critical-section protocol and under the priority-ceiling protocol?

8.2 A system contains the following four periodic tasks. The tasks are scheduled by the rate-monotonic algorithm and the priority-ceiling protocol.

$T_1 = (3, 0.75)$ $\quad b_1 = 0.9$
$T_2 = (3.5, 1.5)$ $\quad b_2 = 0.75$
$T_3 = (6, 0.6)$ $\quad b_3 = 1.0$
$T_4 = (10, 1)$

b_i is the blocking time of T_i. Are the tasks schedulable? Explain your answer.

8.3 Consider a fixed-priority system in which there are five tasks T_i, for $i = 1, 2, 3, 4$, and 5, with decreasing priorities. There are two resources X and Y. The critical sections of T_1, T_2, T_4, and T_5 are $[Y; 3]$, $[X; 4]$, $[Y; 5\,[X; 2]]$, and $[X; 10]$, respectively. (Note that T_3 does not require any resource.) Find the blocking times $b_i(rc)$ of the tasks.

8.4 A fixed-priority system contains four tasks T_i, for $i = 1, 2, 3, 4$, and 5, with decreasing priorities and uses the ceiling-priority protocol to control resource access. There are three resources X, Y, and Z; each has 1 unit. The critical sections of the tasks are $[X; 4]$, $[Y; 6]$, $[Z; 5]$, and $[X; 3\,[Y; 2\,[Z; 1]]]$, respectively. Suppose that T_2 may self-suspend once, and $b_2(ss)$ is 1. The other tasks never self-suspend. What are the blocking times of the tasks?

8.5 Sections 8.6.1 and 8.6.2 give two different implementations (and two different names) of the ceiling-priority protocol.

(a) Discuss the pros and cons of the implementations.

(b) The definitions of the stack-based, priority-ceiling protocol and ceiling-priority protocol do not say whether jobs are allowed to self-suspend. Do protocols still limit the duration of blocking if jobs may self-suspend? If yes, give an intuitive argument to support your answer. If no, give an illustrative example.

(c) Oftentimes, jobs of equal priority are scheduled on the round-robin basis. Modify the definition of priority ceilings of resources and the scheduling rule of the ceiling-priority protocol to make the protocol work for such jobs. (*Hint*: Consider restricting the priorities of all jobs to even integers. Define the priority ceiling of each resource to be the highest of the priorities of all jobs that require the resource minus one. In other words, the ceiling priorities of all resources are odd integers.)

8.6 A fixed-priority system contains five tasks. There are two kinds of resources X and Y. The resource X has 3 units and Y has 2 units. The resource requirements of the tasks are as follows:

T_1 : [X, 1; 3]
T_2 : [Y, 1; 4]
T_3 : [Y, 1; 4 [X, 3; 2]]
T_4 : [X, 1; 4] [Y, 2; 2]]
T_5 : [Y, 1; 3]

The priority of T_i is higher than the priority of T_k if $i < k$.

(a) Suppose that the system uses the stack-based priority-ceiling protocol. What are the maximum blocking times of the jobs?

(b) Suppose that these periodic tasks have the following parameters: $T_1 = (40, 5, 20)$, $T_2 = (30, 5, 25)$, $T_3 = (35, 5, 30)$, $T_4 = (60, 6, 40)$, and $T_5 = (55, 5, 50)$. Are these tasks schedulable? Explain your answer.

8.7 A system contains the following five periodic tasks. The tasks are scheduled rate-monotonically.

$T_1 = (6, 3, [X; 2])$
$T_2 = (20, 5, [Y; 1])$
$T_3 = (200, 5, [X; 3 [Z; 1]])$
$T_4 = (210, 6, [Z; 5 [Y; 4]])$

Compare the schedulability of the system when the priority-ceiling protocol is used versus the NPCS protocol.

8.8 A system contains six jobs, J_i, for $i = 1, \ldots, 6$. Jobs J_2 and J_3 have the same priority. Otherwise, the smaller the index, the higher the priority. There are three resources X, Y, and Z in the system; each resource has 1 unit. The stack-based priority-ceiling protocol is used to control resource access. The amounts of time the jobs use the resources are given by the following table. (A blank means zero.)

Jobs	X	Y	Z
J_1	1		
J_2		7	
J_3			1
J_4			
J_5			2
J_6	3	4	

(a) Find the blocking times of all jobs.

(b) Suppose that all the jobs except J_5 are released at the same time t. At time t, J_5 holds Z. Which jobs are ready for execution and which ones are not? Explain your answers.

8.9 A system contains three periodic tasks (T_1, T_2 and T_3) and three resources (X, Y, and Z). There are 3 units of resource X, 2 units of resource Y, and 3 units of resource Z. The resource requirements of the tasks are as follows:

T_1: [Y, 1; 4 [X, 3; 2]]
T_2: [Z, 3; 5 [Y, 1; 1]]
T_3: [Z, 1; 7 [X, 1; 3]]

(a) Draw the resource requirement graph for the task system.

(b) Suppose that the periods and execution times of the tasks are given by

$T_1 = (20, 5)$
$T_2 = (15, 6)$
$T_3 = (35, 8)$

Construct a table of preemption ceilings for the resources, assuming that the tasks are scheduled according to the earliest-deadline-first algorithm. Are the tasks schedulable? Why?

(c) What are the blocking times of the tasks if the tasks are scheduled rate-monotonically? Are the tasks schedulable according to the rate-monotonic algorithm? Why?

8.10 Given a system consisting of the following tasks whose periods, execution times, and resource requirements are given below.

$$T_1 = (2, 0.4, [X, 3; 0.3])$$
$$T_2 = (3, 0.75, [X, 1; 0.3][Y, 1; 0.4])$$
$$T_3 = (6, 1.0[Y, 1; 0.4][Z, 1; 0.5[X, 1; 0.4]]$$
$$T_4 = (8, 1.0[X, 1; 0.5][Y, 2; 0.1][Z, 1; 0.4])$$

There are 3 units of X, 2 units of Y, and 1 unit of Z. The tasks are scheduled by the EDF algorithm and the stack-based protocol.

(a) Find the preemption ceiling of each resource and the the maximum blocking time for each task.

(b) Are the tasks schedulable according to the earliest-deadline-first algorithm? Why?

8.11 Rule 3 of the basic priority-inheritance protocol assumes that when a job J becomes blocked, the job which directly blocks it has an equal or lower priority. Prove that this is assumption is valid in the absence of a deadlock. Given an example to show that this assumption is not true when there is a deadlock.

8.12 Show that under the control of the priority-inheritance protocol, a job can be blocked directly by any lower-priority job for at most once for the duration of one outermost critical section, in the absence of a deadlock.

8.13 As defined in Section 8.4, three jobs J_1, J_2, and J_3 are said to be *transitively blocked* when J_3 blocks J_2 which in turn blocks J_1. Show that priority-ceiling protocol prevents transitive blocking of three or more jobs and, hence prevents deadlock among three or more jobs.

8.14 You are given a system of n periodic tasks that are scheduled on a fixed-priority basis and under the basic priority-ceiling protocol and the resource requirement graph of the system. Write a pseudocode description of an algorithm that computes the blocking set $\mathbf{BS}_i$ and the worst-case blocking time b_i of each task.

8.15 Suppose that periodic tasks are scheduled according to a fixed-priority scheduling algorithm and there is only 1 unit of resource of each type. We can, therefore, choose to use either the Stack-Based Protocol (SBP) or Priority-Ceiling Protocol (PCP). Give examples to illustrate that (a) PCP is better than SBP and (b) SBP is better than PCP, when performance is measured in terms of blocking times. Explain why such an example (or examples) cannot exist if you cannot find one (or both) of the examples.