UNIT – 5
Resources and Resource Access Control

8.1 ASSUMPTIONS ON RESOURCES AND THEIR USAGE

We assume that the system contains only one processor 'ρ' types of serially reusable resources named $R_1, R_2, \ldots, R_\rho$. There are v_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. 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. 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.

8.1.1 Enforcement of Mutual Exclusion and Critical Sections

We assume that a lock-based concurrency control mechanism is used to enforce mutually exclusive accesses of jobs to resources. 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***. 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. We denote each critical section by a square bracket [R, η; 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 binations of critical sections. We denote nested critical sections by nested square brackets.

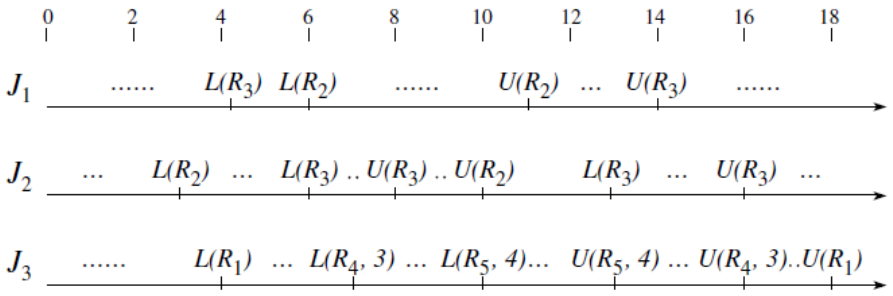


FIGURE 8–1 Examples of critical sections

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. The scheduler always denies a request if there are not enough free units of the resource to satisfy the request. A scheduler may deny a request even when the requested resource units are free.

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.

In the Figure-8.2, 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 $L(R)$.

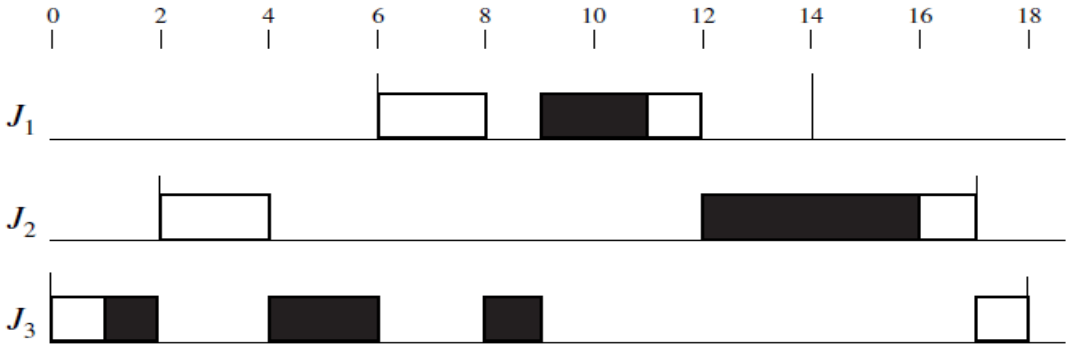


FIGURE 8–2 Example of job interaction due to resource contention.

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.

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.

8.2.1 Priority Inversion, Timing Anomalies, and Deadlock

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.

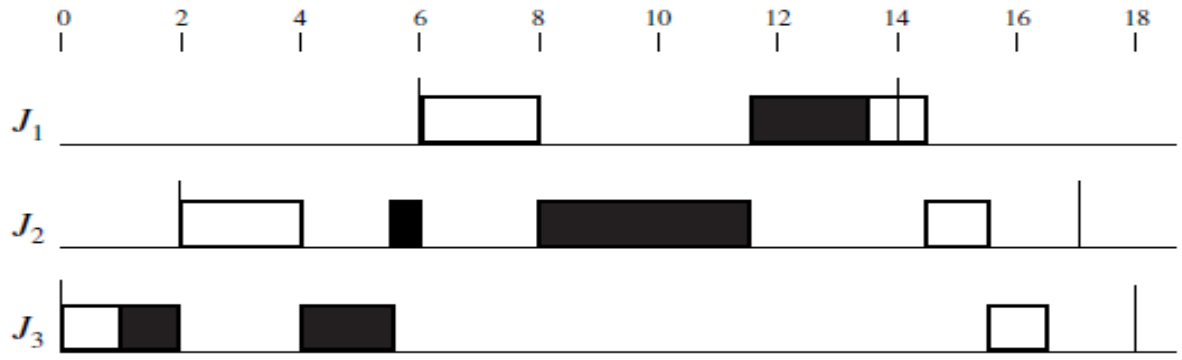


FIGURE 8–3 Example illustrating timing anomaly.

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. 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.

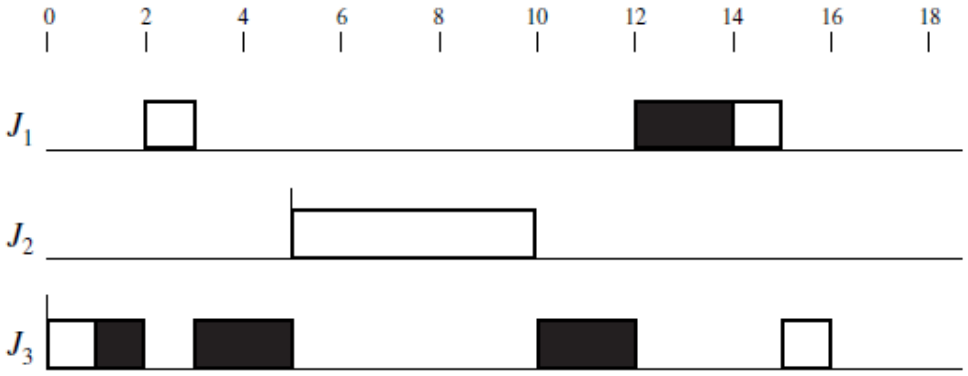


FIGURE 8–4 Uncontrolled priority inversion.

8.2.2 Additional Terms, Notations, and Assumptions

We assume that ***no job ever suspends itself and every job is preemptable on the processor***. 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 .

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. So, 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

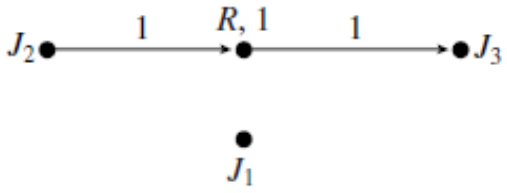


FIGURE 8–5 A wait-for-graph of the system in Figure 8–2 at time 5.

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 (ϕ_i , p_i , e_i , D_i , [R, x; y]).

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 (ϕ_i , p_i , e_i , D_i ; c_i); the last element ci is the maximum execution time of the longest critical section of the jobs in the periodic 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

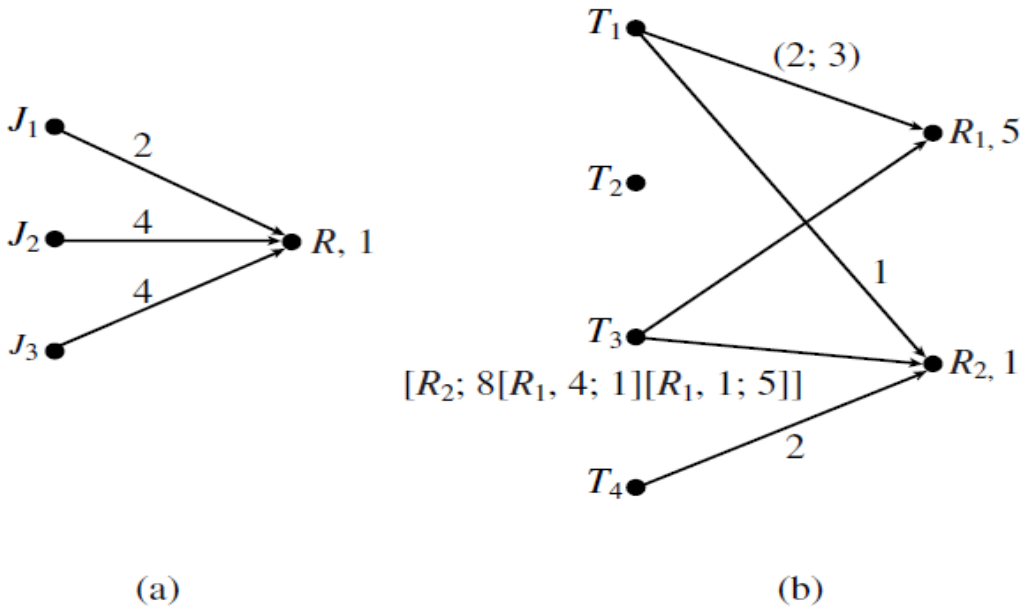


FIGURE 8–6 Graphs that specify resource requirements.

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. 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 when the tasks are indexed in order of nonincreasing priority.

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

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 priority.

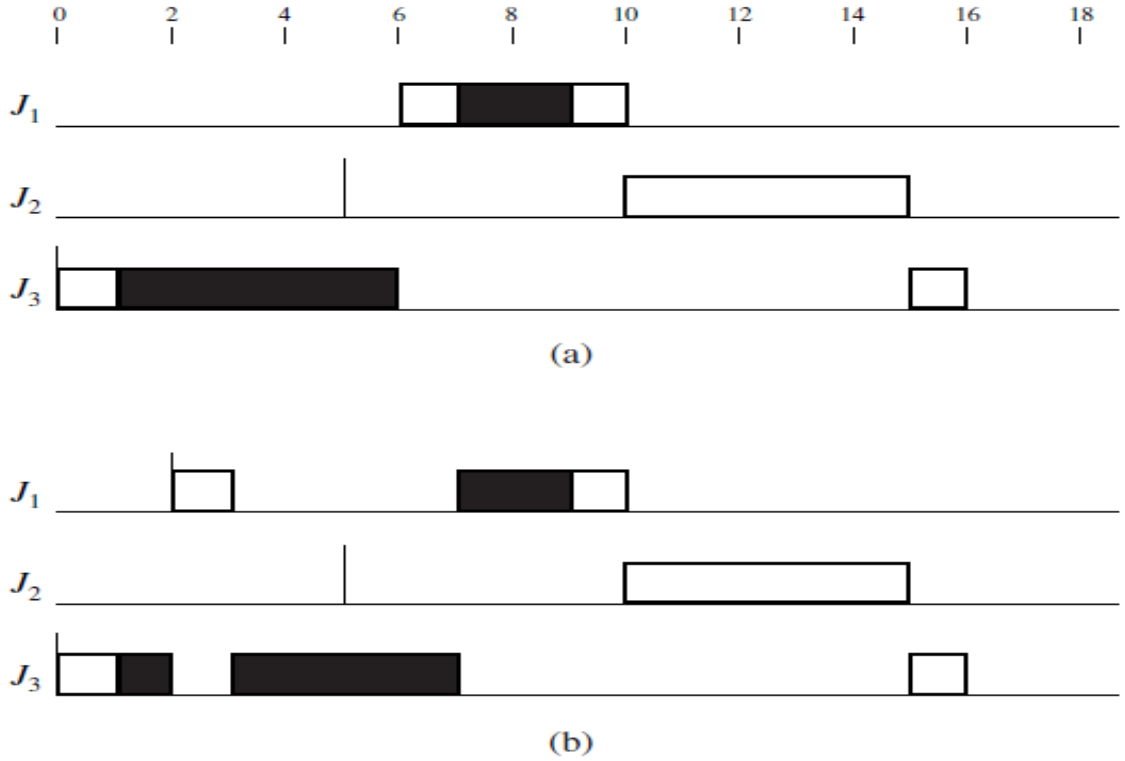


FIGURE 8–7 Example to illustrate two simple protocols. (a) Controlling priority inversion by disallowing preemption of critical section. (b) Controlling priority inversion by using priority inheritance.

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 works.

8.4 BASIC PRIORITY-INHERITANCE PROTOCOL

The priority-inheritance protocol proposed by Sha, *et al.* 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 the protocol ensures that no job is ever blocked for an indefinitely long time because uncontrolled priority inversion cannot occur.

8.4.1 Definition ofBasic 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***. 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*** $\pi_h(t)$.

The ***priority-inheritance protocol*** is defined by the following rules.

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. 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. So, 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.

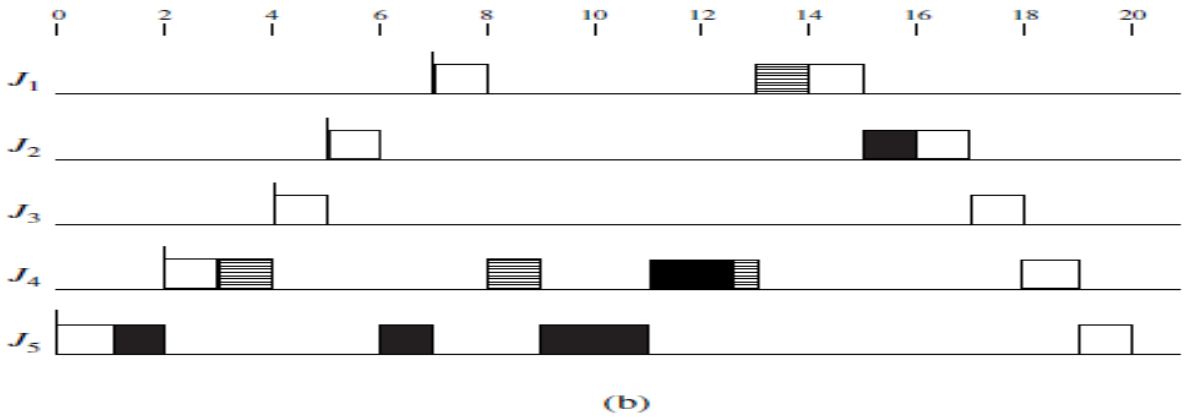
Figure 8–8 gives a more complex example. In this example, there are five jobs and two resources ***Black* and *Shaded***. The parameters of the jobs and their critical sections are listed in part (a). As usual, jobs are indexed in decreasing order of their priorities: The priority π_i of J_i is i , and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

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.

8.4.2 Properties of 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 J3 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, J5 indirectly inherits the J_1's priority. The priority-inheritance protocol does not prevent deadlock.

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. 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.

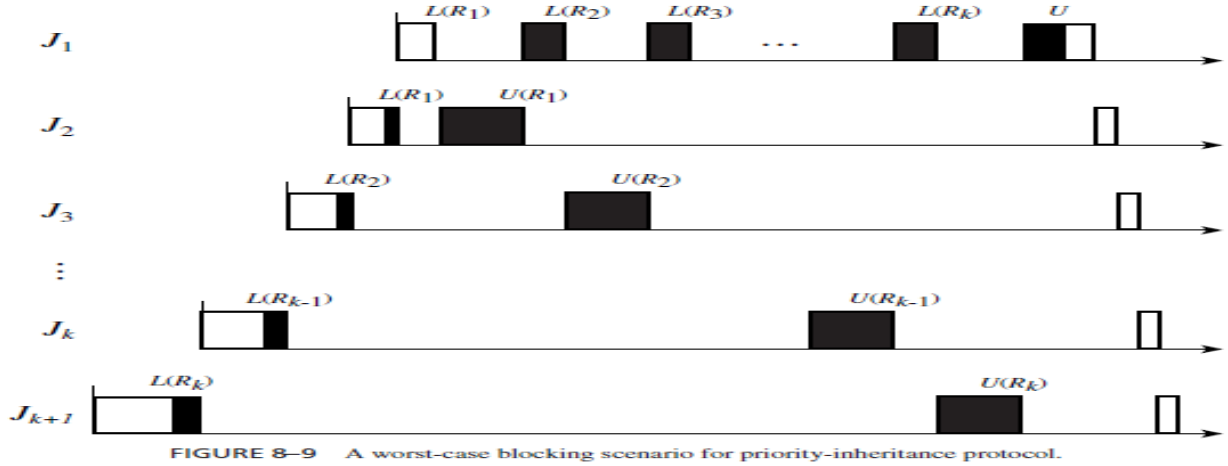


FIGURE 8–9 A worst-case blocking scenario for priority-inheritance protocol.

8.5 BASIC PRIORITY-CEILING PROTOCOL

The ***priority-ceiling protocol*** 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.

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 , 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 , 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).

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 protocol.

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.

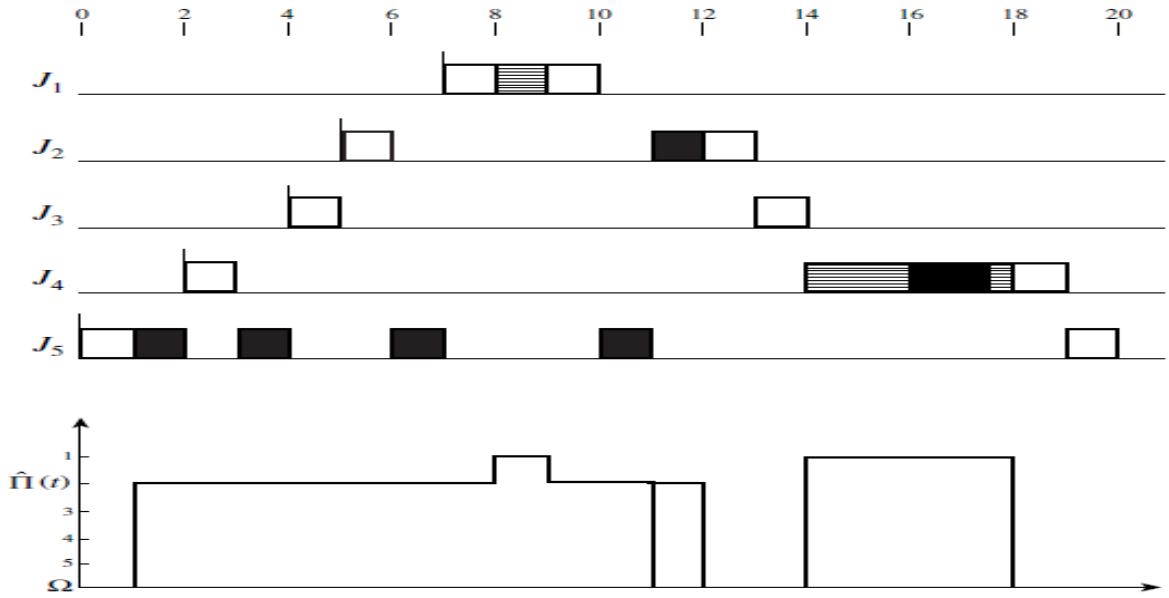


FIGURE 8–10 A schedule illustrating priority-ceiling protocol.

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 , 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 J4 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. 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

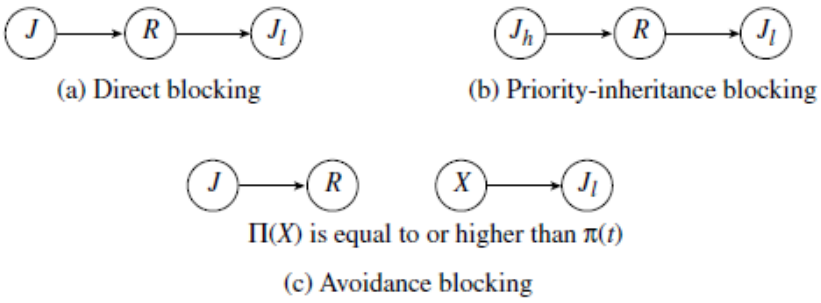


FIGURE 8–11 Ways for a job to block another job. (a) Direct blocking. (b) Priority-inheritance blocking. (c) Avoidance blocking.

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. 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*.**

8.5.3 Deadlock Avoidance by 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*),

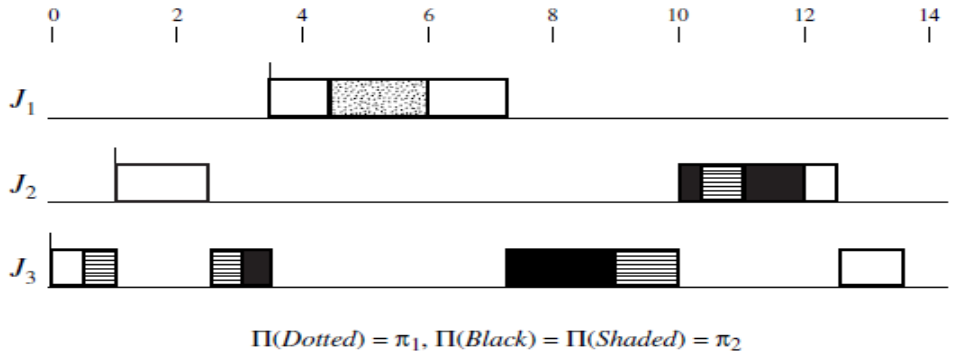


FIGURE 8–12 Example illustrating how priority-ceilling protocol prevents deadlock.

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.

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. This is the reason that J_1 was granted *Dotted*.

In general, at any time t , the priority π(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.

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 ofBlocking

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*.**

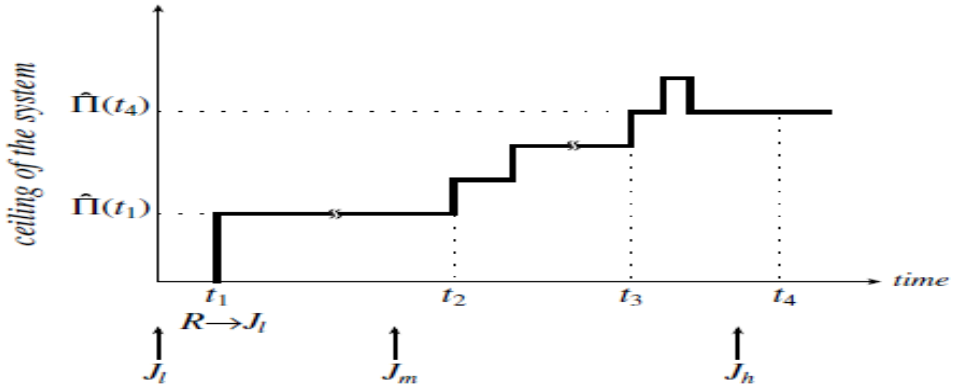


FIGURE 8–13 Scenario illustrating a property of basic priority-ceiling protocol.

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 πm is higher than $\hat{\Pi}(t_1)$. Suppose that at time t_4, J_h becomes blocked.

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. 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, . . . , *i* − 1 *of the direct-blocking table*.

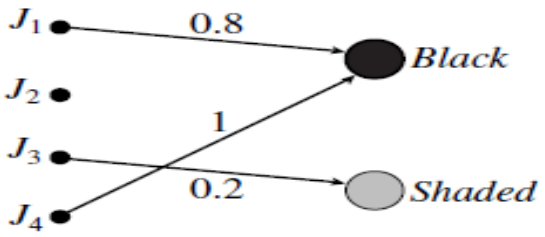
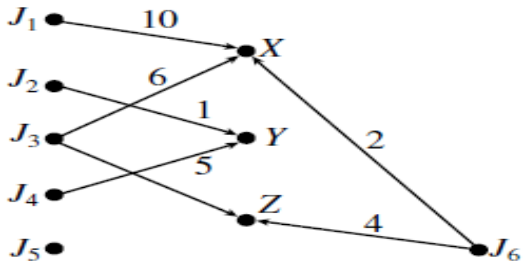


FIGURE 8–14 Example on duration of blocking.



(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* , 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, . . . , 6, respectively. 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.

In Problem 8.14, you are asked to provide a pseudocode description of an algorithm that computes the blocking time *bi (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 *Ji* the subset of all jobs that may block the job. This subset is called the ***blocking set*** of J_i .

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. 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 *= (ε, 2, 0.8; [Black; 0.8]),* T_2 *= (ε, 2.2, 0.4),* T_3 *= (ε, 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. T_2 misses it is deadline at time *2.2 + ε*. 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+ε]* when the blocking time b_2*(rc)* = 1.0 of T_2 is included and becomes *3.0* at *2.0+ε*.

So, the time supply by T_2's first deadline at 2.2+*ε* cannot meet this demand. Similarly, if the jobs J_i, for *i* = 1, 2, . . . , 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.

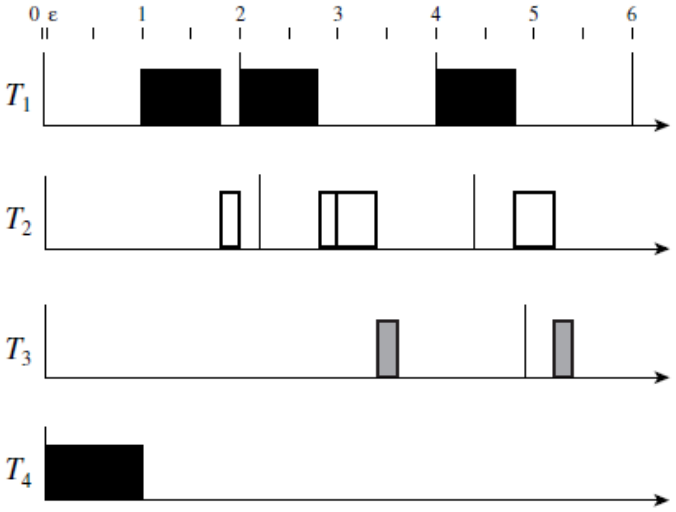


FIGURE 8–16 Example on fixed-priority scheduling and priority-ceiling protocol. ($T_1 = (\varepsilon, 2, 0.8; [Black; 0.8])$, $T_2 = (\varepsilon, 2.2, 0.4)$, $T_3 = (\varepsilon, 5, 0.2; [Shaded; 0.2])$, $T_4 = (10, 1; [Black; 1.0])$).

When the tasks contend for resources under the control of the basic priority-ceiling protocol, each job can be blocked at most once. 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.

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

.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. Like Baker's protocol, ***this protocol allows jobs to share the run-time stack if they never self-suspend*.**

In the statement of 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 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.

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. 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 $J5$ becomes free and the ceiling of the system is at Ω. So, 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. 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.

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- 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.

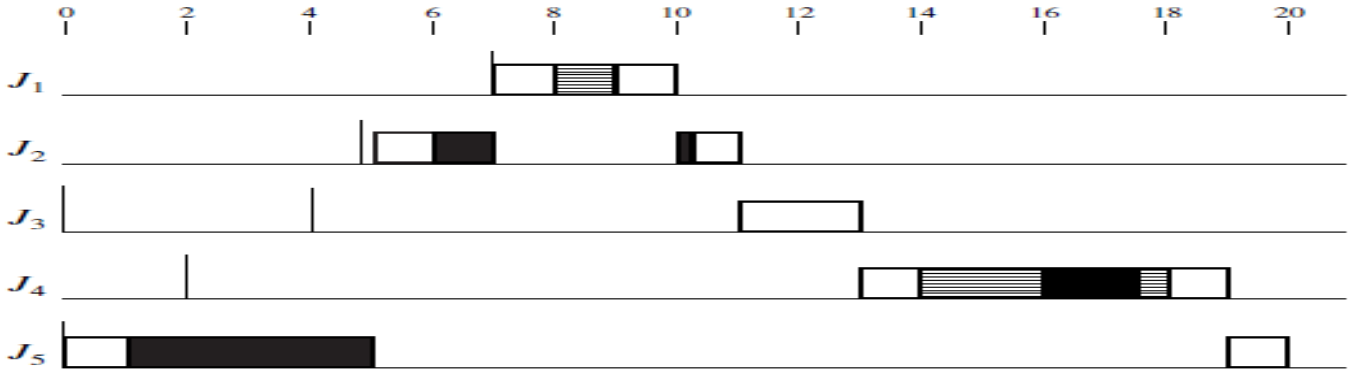


FIGURE 8–17 Schedule illustrating the stack-based, priority-ceiling protocol.

8.6.2 Definition of Ceiling-Priority Protocol

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. 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.

8.6.3 Blocking Time and Context-Switching Overhead

Because of the following theorem, 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.

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.