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

In a dynamic-priority system, the priorities of the periodic tasks change with time while the resources required by each task remain constant. So, 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.

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. The order in which jobs in the ready job queue are sorted among themselves does not alter. 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. 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.

The protocol remains effective 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.

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 . At time 0.3, $J_{2,1}$ acquires *Shaded*, and the ceiling of the system rises from 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.

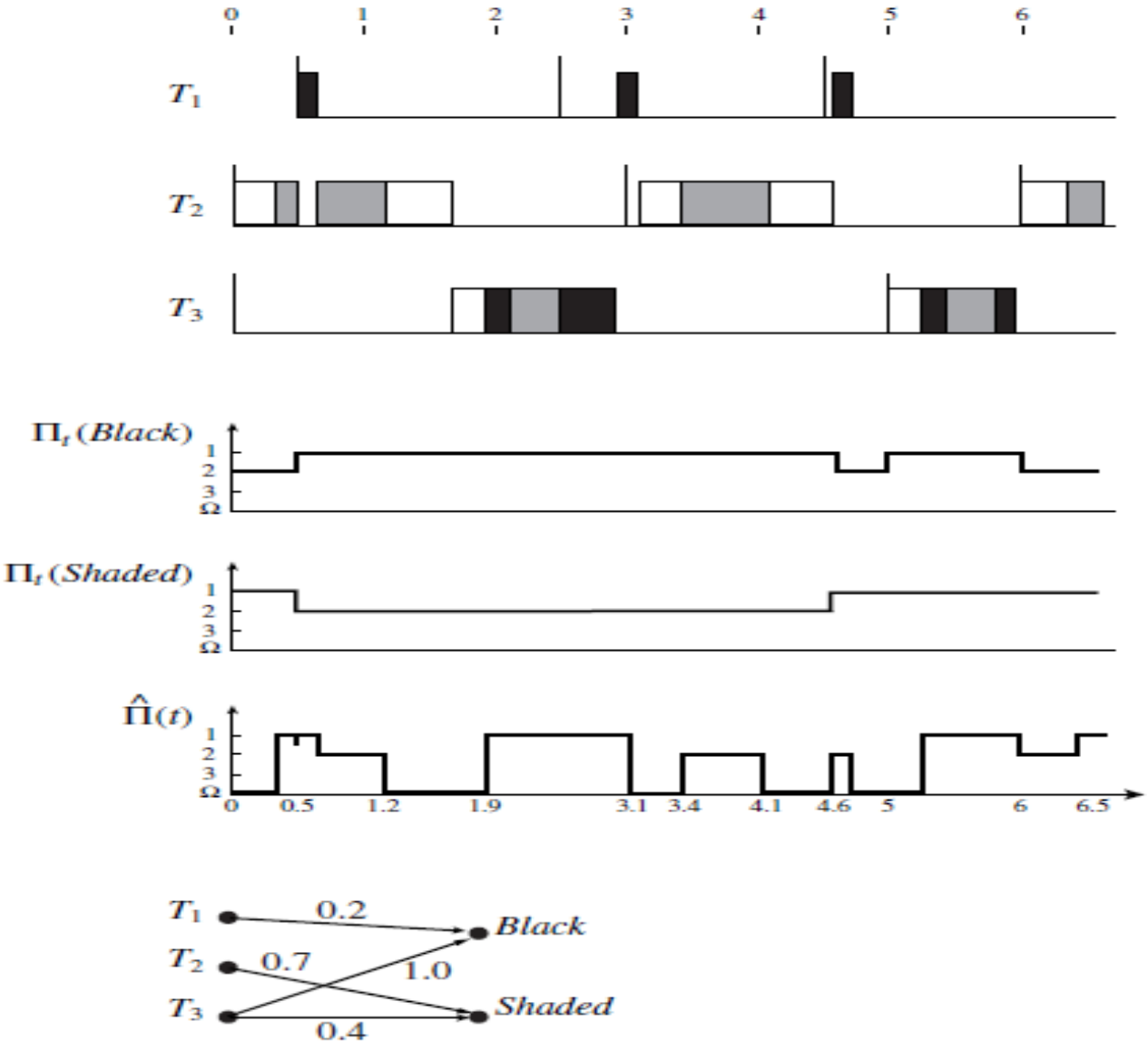


FIGURE 8–18 Example illustrating the use of the basic priority-ceiling protocol in an EDF system.

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 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. Each time a new job is released, we use the precomputed priority ceilings. The storage overhead for this purpose is $O(N_\rho)$.

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**

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.

So, 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 Jk to be directly blocked by Ji , while it blocks some other job Jj .

8.8 PREEMPTION-CEILING PROTOCOL

For a fixed preemption-level system, Baker 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.

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,

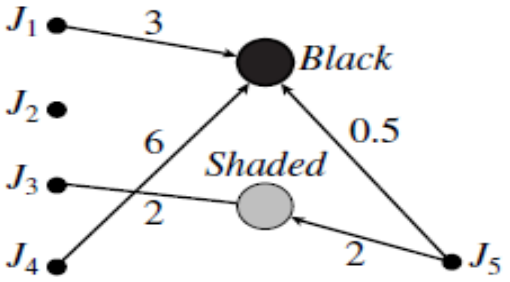
Validity Condition: If π_i is higher than π_k and $r_i > r_k$, then ψ_i is higher than ψ_k .

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.

Figure 8–19 gives an 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. 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.



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.

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 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.

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. When there is only 1 unit of each resource 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 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 .

Like the priority-ceiling protocol, the preemption-ceiling protocol also has a ***basic version*** and a ***stack-based version.*** The allocation rule of the basic preemption-ceiling protocol is given below.

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). It is defined by the following rules.

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.

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.

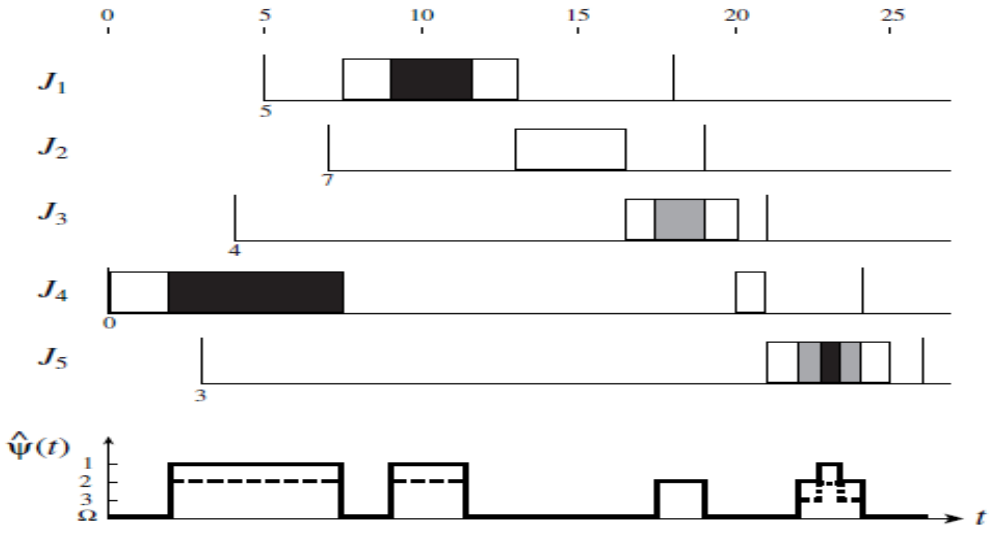


FIGURE 8–20 Example of priority ceilings of multiple-unit resources.

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 $J1$ and $J2$ 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. 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. By 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. So, it would complete before J_4 completes its critical section. This uncontrolled priority inversion is prevented by the inheritance rule.

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. None of the jobs that are holding any resources at t can later preempt J .

8.9 CONTROLLING ACCESSES TO MULTIPLE-UNIT RESOURCES

8.9.1 Priority (Preemption) Ceilings of Multiple-Unit Resources

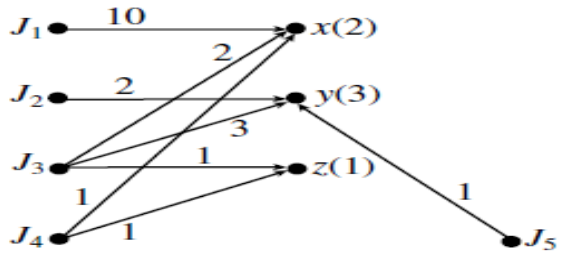
The first step in extending the priority-ceiling protocol is to modify the definition of the priority ceilings of resources. We let $\Pi(R_i, k)$, for $k \le v_i$, denote the priority ceiling of a resource R_i when k out of the v_i (≥ 1) units of R_i are free. If one or more jobs in the system require 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 , 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.



Resources		x(2)	y(3)	z(1)
Units Required by	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 (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 resource(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.

8.9.3 Priority-Inheritance Rule

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 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. In this case, the following priority-inheritance rule works well.

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 ; therefore, it is granted 1 unit of *Black* and continues to execute. The ceiling of the system stays at 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)

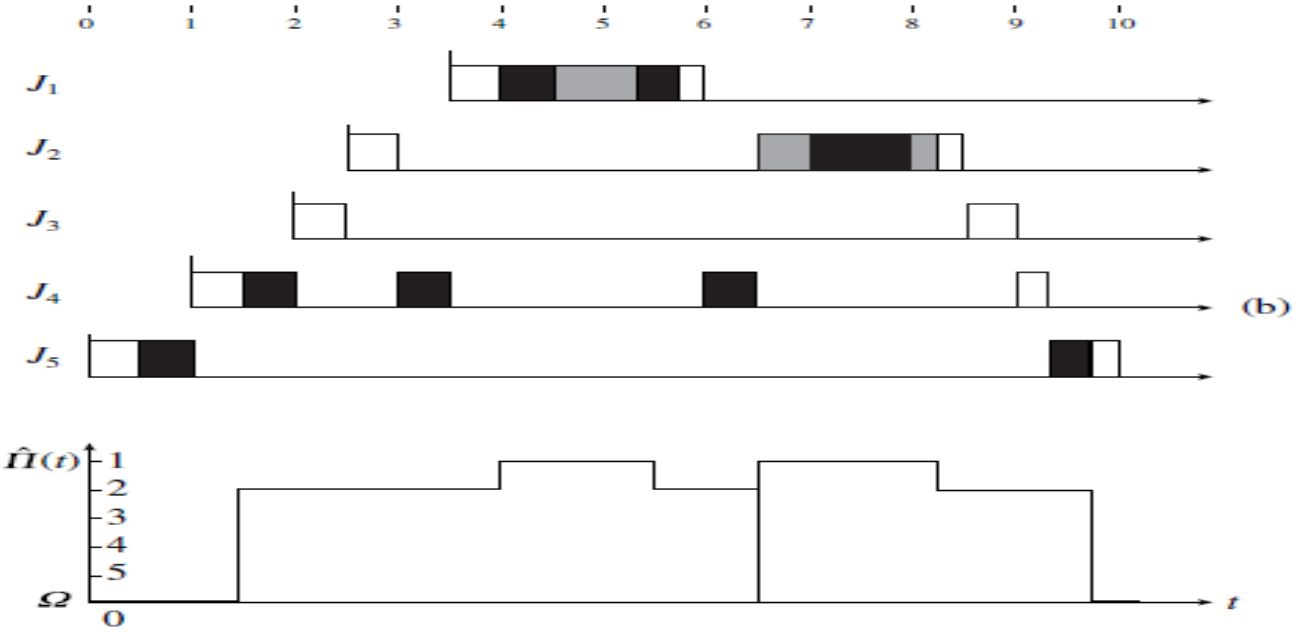


FIGURE 8–22 Example of multiple-unit resource access.

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.

*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 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.

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.

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.

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 . 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 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.

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.

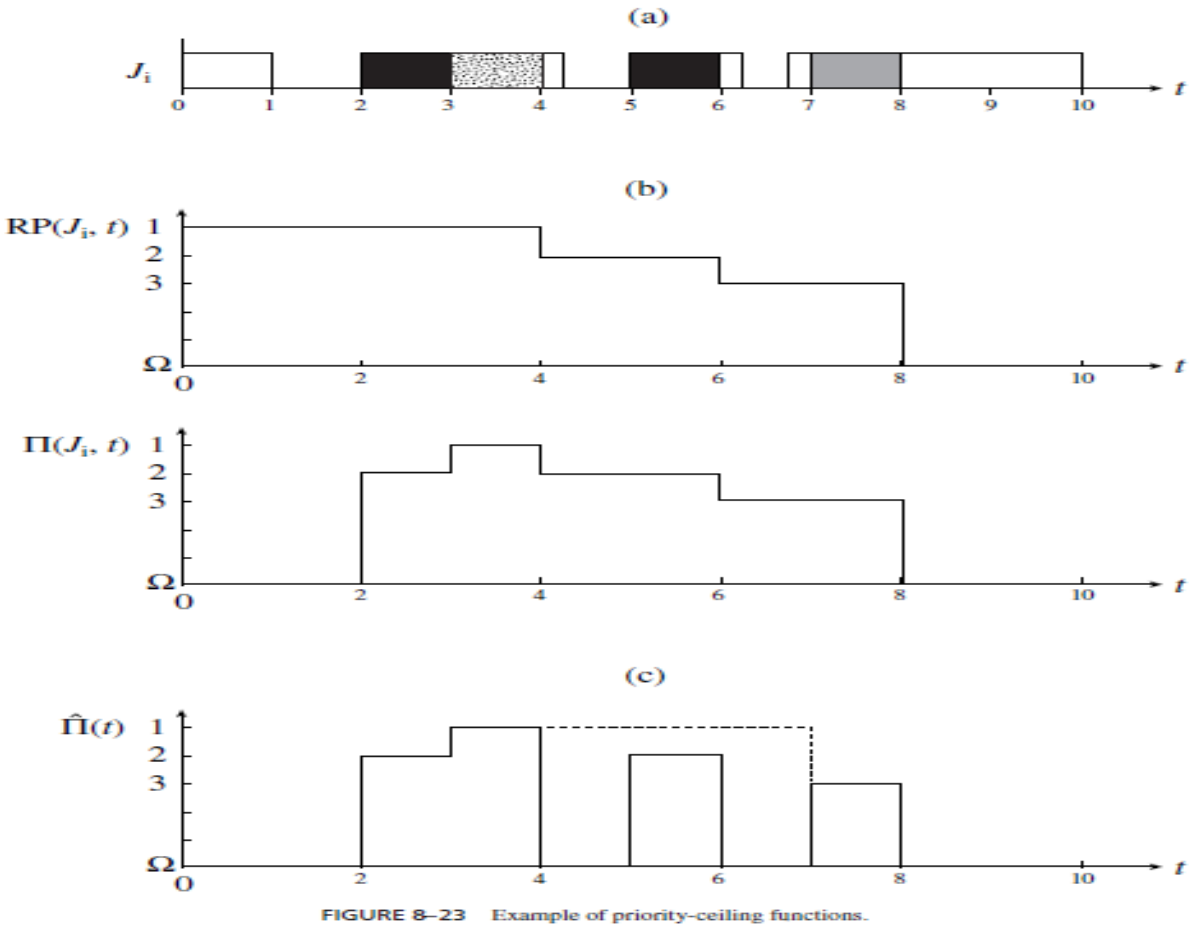


FIGURE 8–23 Example of priority-ceiling functions.

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

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.

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.

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. 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.* showed that the 2PL-HP schemes perform well for soft real-time transactions compared with Optimistic Concurrency Control (OCC) schemes. 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 the lower-priority transaction is restarted immediately. 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. 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.

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 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. Song, *et al.* found that the age and dispersion of data read by transactions tend to be larger under an OCC scheme than under lock-based schemes. This is because both blocking and preemption can cause temporal inconsistency.

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.