3 Handling Resource Sharing and Dependencies Among Real-Time Tasks

The different task scheduling algorithms that we discussed in the last chapter were all based on the premise that the different tasks in a system are all independent. However, that is rarely the case in real-life applications. Tasks often have explicit dependencies specified among themselves, however implicit dependencies are more common. Tasks might become inter dependent for several reasons. A common form of dependency arises when one task needs the results of another task to proceed with its computations. For example, the positional error computation task of a fly-by-wire aircraft may need the results of a task computing the current position of the aircraft from the sampled velocity and acceleration values. Thus, the positional error computation task can meaningfully run only after the "current position determination" task completes. Further, even when no explicit data exchanges are involved, tasks might be required to run in a certain order. For example, the system initialization task may need to run first, before other tasks can run.

Dependency among tasks severely restricts the applicability of the results on task scheduling we developed in the last chapter. The reason is that EDF and RMA schedulers impose no constraints on the order in which various tasks execute. Schedules produced by EDF or RMA might violate the constraints imposed due to task dependencies. We, therefore, need to extend the results of the last chapter in order to be able to cope up with inter-task dependencies.

Further, the CPU scheduling techniques that we studied in the last chapter cannot be satisfactorily used to schedule access of a set of real-time tasks to shared critical resources. We had assumed in the last chapter that tasks can be started and preempted at any time without making any difference to the correctness of the final computed results. However a task using a critical resource can not be preempted at any time from resource usage without risking the correctness of the computed results. Non-preemptability is a violation of one of the basic premises with which we had developed the scheduling results of the last chapter. So, the results of the last chapter are clearly inapplicable to scheduling the access of several real-time tasks to some critical resources and new methods to schedule critical resources among tasks are required.

In this chapter we first discuss how critical resources may be shared (scheduled) among a set of real-time tasks. We investigate the problems that may arise if the traditional resource sharing mechanisms are deployed in real-time applications. Subsequently, we discuss a few important protocols for effective sharing of critical resources among real-time tasks. Finally, we discuss how the scheduling methods of the last chapter can be augmented to make them applicable to tasks with dependencies.

3.1 RESOURCE SHARING AMONG REAL-TIME TASKS

In many applications, real-time tasks need to share some resources among themselves. Often these shared resources need to be used by the individual tasks in exclusive mode. This means that a task that is using a resource, can not immediately hand over the resource to another task that requests the resource at any arbitrary point in time; but it can do so only after it completes its use of the resource. If a task is preempted from using a resource before it completes using the resource, then the resource can become corrupted. Examples of such resources are files, devices, and certain data structures. These resources are also called non-preemptable resources, or critical resources. Some authors loosely refer to non-preemptable resources as *critical sections*, though the standard operating system literature uses this term to refer to sections of code in which some non-preemptable resources are used in exclusive mode.

Sharing of critical resources among tasks requires a different set of rules, compared to the rules used for sharing resources such as a CPU among tasks. We have in the last chapter discussed how resources such as CPU can be shared among tasks with cyclic scheduling, EDF, and RMA being the popular methodologies. We must understand that CPU is an example of a *serially reusable resource*. But, what exactly is a serially reusable resource and how does it differ from a critical resource? Once a task gains access to a serially reusable resource such as CPU, it uses it exclusively. That is, two different tasks can not run on one CPU at the same time. Another important feature of a serially reusable resource is that a task executing on a CPU can be preempted and restarted at a later time without any problem. A serially reusable resource is one which is used in exclusive mode, but a task using it may at any time be preempted from using it, and then allowed to use it at some later time without affecting the correctness of the computed results. A non-preemptable resource is also used in the exclusive mode. However, a task using a non-preemptable resource can not be preempted from the resource usage, otherwise the resource would become inconsistent and can lead to system failure. Therefore, when a lower priority task has already gained access to a non-preemptable resource and is using it, even a higher priority task would have to wait until the lower priority task using the resource completes. For this reason, algorithms such as EDF and RMA that are popularly used for sharing a set of serially reusable resources (e.g., CPU) can not satisfactorily be used to share non-preemptable resources among a set of real-time tasks.

We now discuss the problems that would arise when traditional techniques of resource sharing such as semaphores and locks are used to share critical resources among a set of real-time tasks.

3.2 PRIORITY INVERSION

In traditional systems, the mechanisms popularly employed to achieve mutually exclusive use of data and resources among a set of tasks include semaphores, locks, and monitors. However, these traditional operating system techniques prove inadequate for use in real-time applications. The reason being that if these techniques are used for resource sharing in a real-time application, not only simple *priority inversions* but also more serious *unbounded priority inversions* can occur. Unbounded priority inversions are severe problems that may lead to a real-time task to miss its deadline and cause system failure.

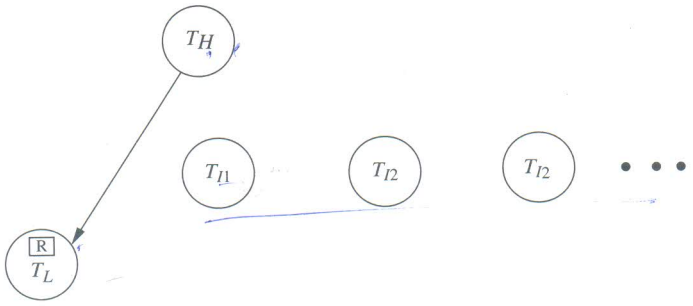
To explain the concept of a simple priority inversion, consider the following. When a lower priority task is already holding a resource, a higher priority task needing the same resource has to wait and can not make progress with its computations. The higher priority task would remain blocked until the lower priority task releases the required non-preemptable resource. In this situation, the higher priority task is said to undergo simple priority inversion on account of the lower priority task for the duration it waits while the lower priority task keeps holding the resource.

It should be obvious that simple priority inversions are often unavoidable when two or more tasks share a non-preemptable resource. Fortunately, a single simple priority inversion is not really that difficult to cope with. The duration for which a simple priority inversion can occur is bounded by the longest duration for which a lower priority task might need a critical resource in an exclusive mode. While even a simple priority inversion does delay a higher priority task by some time, the duration for which a task blocks due to a simple priority inversion can be made very small if all tasks are made to restrict themselves to very brief periods of critical section usage. Therefore, a simple priority inversion can easily be tolerated through careful programming. However, a more serious problem that arises during sharing of critical resource among tasks is unbounded priority inversion.

Unbounded priority inversion is a troublesome problem that programmers of real-time and embedded systems often encounter. Unbounded priority inversions can upset all calculations of a programmer regarding the worst case response time of a real-time task and cause it to miss its deadline.

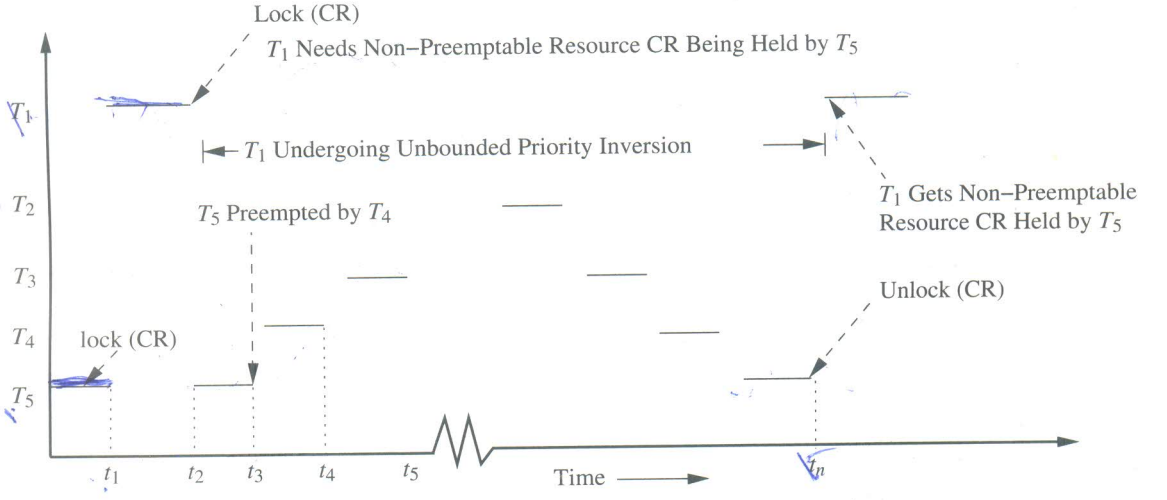
> Unbounded priority inversion occurs when a higher priority task waits for a lower priority task to release a resource it needs, and meanwhile intermediate priority tasks preempt the lower priority task from CPU usage repeatedly. As a result, the lower priority task can not complete its usage of the critical resource and the higher priority task waits indefinitely for its required resource to be released.

Let us now examine more closely what is meant by unbounded priority inversion and how it arises through a simple example. Consider a real-time application having a high priority task T_H and a low priority task T_L. Assume that T_H and T_L need to share a critical resource R. Besides T_H and T_L, assume that there are several tasks T_{I1}, T_{I2}, T_{I3}, . . . that have priorities intermediate between T_H and T_L, and do not need the resource R in their computations. These tasks have schematically been shown in Fig. 3.1. Assume that the low priority task (T_L) starts executing at some instant and locks the non-preemptable resource as shown in Fig. 3.1. Suppose soon afterwards, the high priority task (T_H) becomes ready, preempts T_L and starts executing. Also assume that it needs the same non-preemptable resource. It is clear that T_H would block for the resource as it is already being held by T_L and T_L would continue its execution. But, the low priority task T_L may be preempted (from CPU usage) by other intermediate priority tasks (T_{I1}, T_{I2}, . . .) which become ready and do not require R. In this case, the high priority task (T_H) would have to wait not only for the low priority task (T_L) to complete its use of the resource, but also all intermediate priority tasks (T_{I1}, T_{I2}, . . .) preempting the low priority task to complete their computations. This might result in the high priority task having to wait for the required resource for a considerable period of time. In the worst case, the high priority task might have to wait indefinitely for a low priority task to complete its use of the resource in the face of repeated preemptions of the low priority tasks by the intermediate priority tasks not needing the resource. In such a scenario, the high priority task is said to undergo *unbounded priority inversion.*



▲ **FIGURE 3.1**
Unbounded Priority Inversion

Unbounded priority inversion is an important problem that real-time system developers face. We, therefore, illustrate this problem using another example and this time showing the actions of the different tasks on a time line. Consider the example shown in Fig. 3.2. In this example, there are five tasks: $T_1 \ldots T_5$. T_5 is a low priority task and T_1 is a high priority task. Tasks T_2, T_3, T_4 have priorities higher than T_5 but lower than T_1 (called intermediate priority tasks). At time t_0, the low priority task T_5 is executing and is using a non-preemptable resource (CR). After a while (at time instant t_1) the higher priority task T_1 arrives, preempts T_5 and starts to execute. Task T_1 requests to use the resource CR at t_2 and blocks since the resource is being held by T_5. Since T_1 blocks, T_5 resumes its execution at t_2. However, the task T_4 which does not require the non-preemptable resource CR preempts T_5 from CPU usage and starts to execute at time t_3. T_4 is in turn preempted by T_3 and so on. As a result, T_1 suffers multiple priority inversions and may even have to wait indefinitely for T_5 to get a chance to execute and complete its usage of the critical resource CR and release it. It should be clear that when a high priority task undergoes unbounded priority inversion, it is very likely to miss its deadline.



▲ **FIGURE 3.2**
Explanation of Unbounded Priority Inversion Using a Timing Diagram

It is important to note that unbounded priority inversions arise when traditional techniques for sharing non-preemptable resources such as semaphores or monitors are deployed in real-time applications. Possibly the simplest way to avoid priority inversions is to prevent preemption (from CPU usage) of the low priority task holding a critical resource by intermediate priority tasks. This can be achieved by raising the priority level of the low priority task to be equal to that of the waiting high priority task. In other words, the low priority task is made to inherit the priority of the waiting high priority task. This basic mechanism of a low priority task inheriting the priority of a high priority task forms the central idea behind the *priority inheritance protocol (PIP)*. This simple protocol serves as the basic real-time resource sharing mechanism, based on which more sophisticated protocols have been designed.

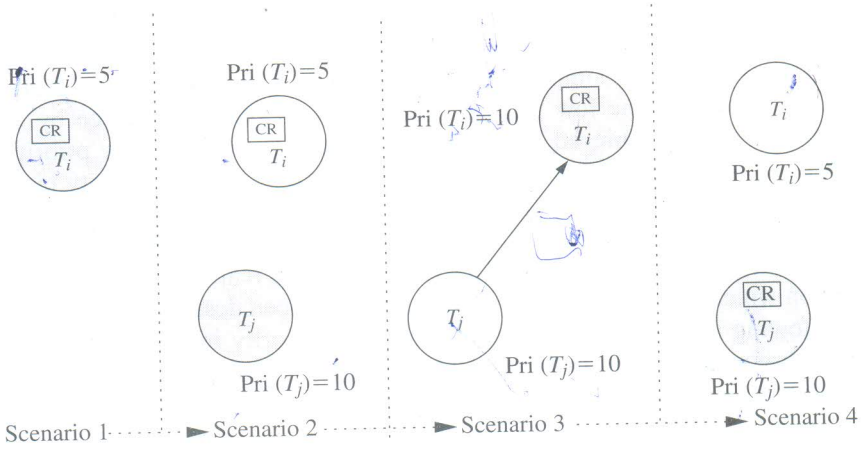
In the following sections we first discuss the basic priority inheritance protocol (PIP) in some detail. Subsequently, we discuss highest locker protocol (HLP) and priority ceiling protocol (PCP) that have been developed by extending the simple priority inheritance idea further.

3.3 PRIORITY INHERITANCE PROTOCOL (PIP)

The basic priority inheritance protocol (PIP) is a simple technique to share critical resources among tasks without incurring unbounded priority inversions. As it turns out, real-time operating system designers do not find it very difficult to support the basic priority inheritance mechanism. In fact, as we discuss in Chapter 5, most of the real-time operating systems that are commercially available at present do support this protocol.

The basic PIP was proposed by Sha and Rajkumar [24]. The essence of this protocol is that whenever a task suffers priority inversion, the priority of the lower priority task holding the resource is raised through a priority inheritance mechanism. This enables it to complete its usage of the critical resource as early as possible without having to suffer preemptions from the intermediate priority tasks. When several tasks are waiting for a resource, the task holding the resource inherits the highest priority of all tasks waiting for the resource (if this priority is greater than its own priority). Since the priority of the low priority task holding the resource is raised to equal the highest priority of all tasks waiting for the resource being held by it, intermediate priority tasks can not preempt it and unbounded priority inversion is avoided. As soon as a task that had inherited the priority of a waiting higher priority task (because of holding a resource), releases the resource it gets back its original priority value if it is holding no other critical resources. In case it is holding other critical resources, it would inherit the priority of the highest priority task waiting for the resources being held by it.

```
if the required resource is free then
       grant it.
if the required resource is being held by a higher priority task then
       wait for the resource
if the required resource is held by a lower priority task then
{
       wait for the resource
       the low priority task holding the
       resource acquires the highest priority of
       tasks waiting for the resource.
}
```



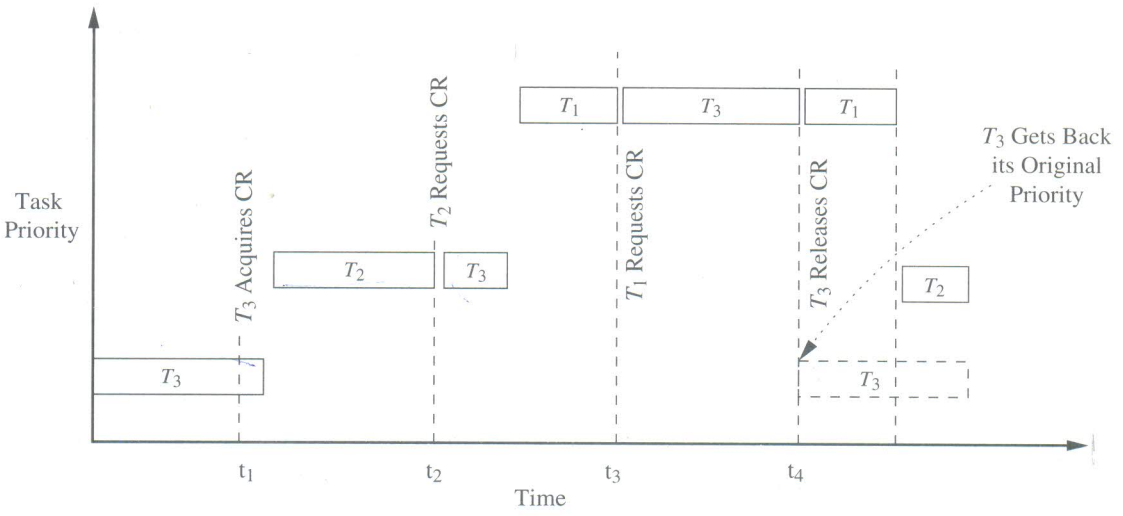
▲ FIGURE 3.3
Snapshots Showing Working of Priority Inheritance Protocol

The priority changes that a task holding a resource undergoes has been illustrated through an example as shown in the Figs. 3.3 and 3.4. In Fig. 3.3 four consecutive scenarios in the execution of a system deploying PIP are shown. In each scenario, the executing task is shown shaded, and the blocked task are shown unshaded. In this figure, how the priority of the two tasks T_i and T_j change due to priority inheritance in the course of execution of the system is shown. T_i is a low priority task with priority 5 and T_j is a higher priority task with priority 10. In scenario 1, T_i is executing and has acquired a critical resource CR. In scenario 2, T_j has became ready and being of higher priority is executing. In scenario 3, T_j is blocking after requesting for the resource R and T_i inherits T_j's priority. In scenario 4, T_i has unlocked the resource and has got back its original priority and T_j is executing with the resource.

In Fig. 3.4 the priority changes of tasks are captured on a time line. The task T_3 initially locks the resource CR. After some time it is preempted by T_2. The task T_2 requests the resource CR at t_2. Since, T_3 already holds the resource, T_2 blocks and T_3 inherits the priority of T_2. This has been shown by drawing T_2 and T_3 at the same priority levels. Before T_3 could complete use of resource CR, it is preempted by a higher priority task T_1. T_1 requests for CR_1 at time t_3 and T_1 blocks as CR_1 is still being held by T_3. So, at this point there are two tasks (T_2 and T_1) waiting for the resource. T_3 inherits the priority of the highest priority waiting task (that is, T_1). T_3 completes its usage of the resource at t_4 and as soon as it releases the resource, it gets back its original priority. It should be noted that a lower priority task retains the inherited priority, until it holds the resource required by the waiting higher priority task. Whenever more than one higher priority task are waiting for the same resource, the task holding the resource inherits the maximum of the priorities of all waiting high priority tasks.

It is clear that PIP can let real-time tasks share critical resources without letting them incur unbounded priority inversions. However, it suffers from two important problems: deadlock and chain blocking.

PIP is susceptible to chain blocking and it does nothing to prevent deadlocks.



▲ **FIGURE 3.4**

Priority Changes Under Priority Inheritance Protocol

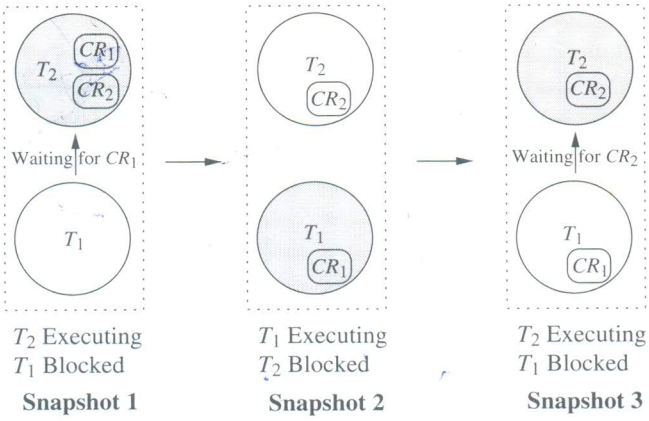
Deadlock: The basic priority inheritance protocol (PIP) leaves open the possibility of deadlocks. In the following we illustrate how deadlocks can occur in PIP. Consider the following sequence of actions by two tasks T_1 and T_2 which need access to two shared critical resources CR_1 and CR_2.

T_1: Lock CR_1, Lock CR_2, Unlock CR_2, Unlock CR_1

T_2: Lock CR_2, Lock CR_1, Unlock CR_1, Unlock CR_2

Assume that T_1 has higher priority than T_2. T_2 starts running first and locks critical resource CR_2 (T_1 was not ready at that time). After some time, T_1 arrives, preempts T_2, and starts executing. After some time, T_1 locks CR_1 and then tries to lock CR_2 which is being held by T_2. As a consequence T_1 blocks and T_2 inherits T_1's priority according to the priority inheritance protocol. T_2 resumes its execution and after some time needs to lock the resource CR_1 being held by T_1. Now, T_1 and T_2 are both deadlocked.

Chain Blocking: A task is said to undergo chain blocking, if each time it needs a resource, it undergoes priority inversion. Thus, if a task needs n resources for its computations, it might have to undergo priority inversions n times to acquire all its resources. Let us now explain how chain blocking can occur using the example shown in Fig. 3.5. Assume that a task T_1 needs several resources. In the first snapshot (shown in Fig. 3.5), a low priority task T_2 is holding two resources CR_1 and CR_2, and a high priority task T_1 arrives and requests to lock CR_1. It undergoes priority inversion and causes T_2 to inherit its priority. In the second snapshot, as soon as T_2 releases CR_1 its priority reduces to its original priority and T_1 is able to lock CR_1. In the third snapshot, after executing for some time, T_1 requests to lock CR_2. This time it again undergoes priority inversion since T_2 is holding CR_2. T_1 waits until T_2 releases CR_2. From this example, we can see that chain blocking occurs when a task undergoes multiple priority inversions to lock its required resources.



▲ **FIGURE 3.5**

Chain Blocking in Priority Inheritance Protocol

3.4 HIGHEST LOCKER PROTOCOL (HLP)

HLP is an extension of PIP and overcomes some of the shortcomings of PIP. In HLP every critical resource is assigned a *ceiling priority* value. The ceiling priority of a critical resource CR is informally defined as the maximum of the priorities of all those tasks which may request to use this resource. Under HLP, as soon as a task acquires a resource, its priority is set equal to the ceiling priority of the resource. If a task holds multiple resources, then it inherits the highest ceiling priority of all its locked resources. An assumption that is implicitly made while using HLP is that the resource requirements of every task is known before compile time.

Though we informally defined the ceiling priority of a resource as the maximum of the priorities of all those tasks which may request to use this resource, the rule for computing the ceiling priority is slightly different for the schedulers that follow FCFS (first come first served) policy among equal priority ready tasks and the schedulers that follow a time-sliced round-robin scheduling among equal priority ready task. In the FCFS policy, a task runs to completion while (any ready) equal priority tasks wait. On the other hand, in time-sliced round-robin scheduling, the equal priority tasks execute for one time slice at a time in a round-robin fashion.

Let us first consider a scheduler that follows FCFS scheduling policy among equal priority tasks. Let the ceiling priority of a resource R_i be denoted by Ceil (R_i) and the priority of a task T_j be denoted as pri(T_j). Then, Ceil (R_i) can be defined as follows:

$$\text{Ceil}\ (R_i) = \max\ (\{\text{pri}\ (T_j)/T_j \text{ needs } R_i\}) \quad \textbf{(3.1)}$$

That is, the ceiling priority of a critical resource R_i is the maximum of the priority of all tasks that may use R_i. This expression holds only when higher priority values indicate higher priority (as in Windows operating system). If higher priority values indicate lower priorities (as in Unix), then the ceiling priority would be the minimum priority of all tasks needing the resource. That is,

$$\text{Ceil}\ (R_i) = \min\ (\{\text{pri}\ (T_j)/T_j \text{ needs } R_i\}) \quad \textbf{(3.2)}$$

For operating systems supporting time-slicedround-robin scheduling among equal priority tasks and larger priority value indicates higher priority, the rule for computing the ceiling priority is:

$$\text{Ceil}(R_i) = \max(\{\text{pri}(T_j)/T_j \text{ needs } R_i\}) + 1 \quad \textbf{(3.3)}$$

For the case where larger priority value indicates lower priority (and time-sliced round-robin scheduling among equal priority tasks), we have to take the minimum of the task priorities. That is:

$$\text{Ceil}(R_i) = \min(\{\text{pri}(T_j)/T_j \text{ needs } R_i\}) + 1 \quad \textbf{(3.4)}$$

In the rest of this chapter we shall assume FCFS scheduling among equal priority tasks and also that increasing priority values indicate increasing priority of tasks. To illustrate how priority ceilings of resources are computed, consider the following example system with FCFS scheduling among equal priority tasks. In this system, a resource CR_1 is shared by the tasks T_1, T_5, and T_7 and CR_2 is shared by T_5, and T_7. Let us assume that the priority of $T_1 = 10$, that of $T_2 = 5$, priority of $T_7 = 2$. Then, the priority of CR_1 will be the maximum of the priorities of T_1, T_5, and T_7. Then, the celling priority of CR_1 is Ceil $(CR_1) = \max(\{10, 5, 2\}) = 10$. Therefore, as soon as either of T_1, T_5, or T_7 acquires CR_1, its priority will be raised to 10. The rule of inheritance of priority is that any task that acquires the resource inherits the corresponding ceiling priority. If a task is holding more than one resource, its priority will become maximum of the celling priorities of all the resources it is holding. For example, Ceil $(CR_2) = \max(\{5, 2\} = 5$. A task holding both CR_1 and CR_2 would inherit the larger of the two ceiling priorities, i.e., 10.

Under HLP, soon after acquiring a resource operates a task at the ceiling priority.

A little thinking can convince you that this helps eliminate the problems of unbounded priority inversions, deadlock, and chain blocking. However, it creates new problems. We formally analyze this point a little later in this section.

HLP solves the problems of unbounded priority inversion, chain blocking, and deadlock. Recollect that the basic PIP was susceptible to these problems. The following theorem and its corollaries substantiate these features of HLP.

THEOREM 3.1 *When HLP is used for resource sharing, once a task gets a resource required by it, it is not blocked any further.*

PROOF Let us consider two tasks T_1 and T_2 that need to share two critical resources CR_1 and CR_2. Let us assume that a task T_1 acquires CR_1. Then, T_1's priority becomes Ceil (CR_1) by HLP. Assume that subsequently it also requires a resource CR_2. Suppose T_2 is already holding CR_2. If T_2 is holding CR_2, T_2's priority should have been raised to Ceil (CR_2) by HLP rule. Obviously, Ceil (CR_2) must be greater than pri(T_1), because Ceil (CR_2) is one more than the maximum of the priority of all tasks using the resource (that includes T_1). Therefore, T_2 being of higher priority should have been executing, and T_1 should not have got a chance to execute. This is a contradiction to the assumption that T_1 is executing while T_2 is holding the resource CR_2. Therefore, T_2 could not be holding a resource requested by T_1 when T_1 is executing. Using a similar reasoning, we can show that when T_1 acquires one resource, all resources required by it must be free. From this we can conclude that a task blocks at best once for all its resource requirements for any set of tasks and resource requirements.

It follows from Theorem 3.1 that under HLP tasks are blocked at most once for all their resource requirements. That is, tasks are *single blocking*. However, we should remember that once a task after getting a resource may be preempted (from CPU usage) by a higher priority task which does not share any resources with the task. But, the "single blocking" we discussed, is blocking on account of resource sharing.

Another point which we must remember is that the deadlock and chain blocking results of HLP (as well as that of PCP discussed in the next section) are valid only under the assumption that once a task releases a resource, it does not acquire any further resources. That is, the request and release of resources by a task can be divided into two clear phases: it first acquires all resources it needs and then releases the resources.

The following are two important corollaries which easily follow from Theorem 3.1.

Corollary 1. *Under HLP, before a task can acquire one resource, all the resources that might be required by it must be free.*

Corollary 2. *A task can not undergo chain blocking in HLP.*

An interesting observation regarding HLP is the following. In PIP whenever several tasks request a critical resource that is already in use, they are maintained in a queue in the order in which they requested the resource. When a resource becomes free, the longest waiting task in the queue is granted the resource. Thus, every critical resource is associated with a queue of waiting tasks. However, in HLP no such queue is needed. The reason is that whenever a task acquires a resource, it executes at the ceiling priority of the resource, and other tasks that may need this resource do not even a get chance to execute and request for the resource.

HLP prevents deadlocks from occurring because when a task gets a single resource, all other resources required by it must be free (by Corollary 2).

Shortcomings of HLP: Though HLP solves the problems of unbounded priority inversion, deadlock, and chain blocking, it opens up the possibility for inheritance-related inversion. Inheritance-related inversion occurs when the priority value of a low priority task holding a resource is raised to a high value by the ceiling rule, the intermediate priority tasks not needing the resource can not execute and are said to undergo inheritance-related priority inversion.

We now illustrate inheritance-related inversion through an example. Consider a system consisting of five tasks: T_1, T_2, T_3, T_4, and T_5, and their priority values be 1, 2, 3, 4, and 5, respectively. Also, assume that the higher the priority value, the higher is the priority of the task. That is, 5 is the highest and 1 is the lowest priority value. Let T_1, T_2, *and* T_3 need the resource CR1 and T_1, T_4, and T_5 need the resource CR_2. Then, Ceil (CR_1) is max (1, 2, 3) = 3, and Ceil (CR_2) is max (1, 4, 5) = 5. When T_1 acquires the resource CR_2 its priority would become 5. After T_1 acquires CR_2, T_2 and T_3 would not be able to execute even though they may be ready, since their priority is less than the inherited priority of T_1. In this situation, T_2 and T_3 are said to be undergoing inheritance-related inversion.

HLP is rarely used in real-life applications as the problem of inheritance-related inversion often become so severe as to make the protocol unusable. This arises because the priority of even very low priority tasks might be raised to very high values when it acquires any resource. As a result, several intermediate priority tasks not needing any resource might undergo inheritance-related inversion and miss their respective deadlines. In spite of this major handicap of HLP, we study this protocol in this text as the foundation for understanding the *priority ceiling protocol* that is very popular and is being used extensively in real-time application developments.

3.5 PRIORITY CEILING PROTOCOL (PCP)

Priority Ceiling Protocol (PCP) extends the ideas of PIP and HLP to solve the problems of unbounded priority inversion, chain blocking, and deadlocks, while at the same time minimizing inheritance-related inversions. A fundamental difference between PIP and PCP is that the former is a greedy approach whereas the latter is not. In PIP whenever a request for a resource is made, the resource will be allocated to the requesting task if it is free. However, in PCP a resource may not be granted to a requesting task even if the resource is free.

Just as HLP does, PCP associates a ceiling value Ceil (CR_i) with every resource CR_i, that is the maximum of the priority values of all tasks that might use CR_i. An operating system variable called CSC (Current System Ceiling) is used to keep track of the maximum ceiling value of all the resources that are in use at any instant of time. Thus, at any time CSC = max ({Ceil (CR_i)/CR_i is currently in use}). At system start, CSC is initialized to 0 (lower priority than the least priority task in the system).

Resource sharing among tasks under PCP is regulated using two rules for handling resource requests: resource grant and resource release.

Resource Grant Rule: Resource grant rule consists of two clauses. These two clauses are applied when a task T_i requests to lock a resource.

1. **Resource request clause:**
   (a) If the task T_i is holding a resource whose ceiling priority equals CSC, then the task is granted access to the resource.
   (b) Otherwise, T_i will not be granted CR_j, unless its priority is greater than CSC (i.e., pri (T_i) > CSC). In both (a) and (b), if T_i is granted access to the resource CR_j, and, if CSC < Ceil (CR_j), then CSC is set to Ceil (CR_j)
2. **Inheritance clause:** When a task is prevented from locking a resource by failing to meet the resource grant clause, it blocks and the task holding the resource inherits the priority of the blocked task if the priority of the task holding the resource is less than that of the blocked task.

Resource Release Rule: If a task releases a critical resource it was holding and if the ceiling priority of this resource equals CSC, then CSC is made equal to the maximum of the ceiling value of all other resources in use; else CSC remains unchanged. The task releasing the resource either gets back its original priority or the highest priority of all tasks waiting for any resources which it might still be holding, whichever is higher.

PCP is very similar to HLP except that in PCP a task when granted a resource does not immediately acquire the ceiling priority of the resource. In fact, under PCP the priority of a task does not change upon acquiring a resource, only the value of a system variable CSC changes. The priority of a task changes by the inheritance clause of PCP only when one or more tasks wait for a resource it is holding. Tasks requesting a resource block almost under identical situations under PCP and HLP. The only difference with PCP is that a task T_i can also be blocked from entering a critical section, if there exists any resource currently held by some other task whose priority ceiling is greater or equal to that of T_i. A little thought would show that this arrangement prevents the unnecessary inheritance blockings that could be caused due to the priority of a task

acquiring a resource being raised to very high values (ceiling priority) at the instant it acquires a resource. In PCP, instead of actually raising the priority of the task acquiring a resource, only the value of a system variable (CSC) is raised to the ceiling value. By comparing the value of CSC against the priority of a task requesting a resource, the possibility of deadlocks is avoided. If no comparison with CSC would have been made (as in PIP), a higher priority task may later lock some resource required by this task leading to a potential deadlock situation where each task holds a part of the resources required by the other task.

We now explain the working of PCP through an example.

Example 3.1

Consider a system consisting of four real-time tasks T_1, T_2, T_3, T_4. These four tasks share two non-preemptable resources CR_1 and CR_2. Assume CR_1 is used by T_1, T_2, and T_3, and CR_2 is used by T_1 and T_4. Assume that the priority values of T_1, T_2, T_3, and T_4 are 10, 12, 15, and 20, respectively. Assume FCFS scheduling among equal priority tasks, and that higher priority values indicate higher priorities.

From the given data, the ceiling priority of the two resources can be easily computed.

Ceil (CR_1) = max (pri (T_1), pri (T_2), pri (T_3)) = 15.

Ceil (CR_2) = max (pri (T_1), Pri (T_4)) = 20.

Let us consider an instant in the execution of the system in which T_1 is executing after acquiring the resource CR_1.

When T_1 acquires CR_1, CSC is set to Ceil (CR_1) = 15.

Now consider the following two alternate situations that might arise in the course of execution of the tasks.

Situation 1

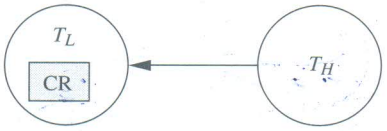
Assume that T_4 becomes ready. Being of higher priority T_4, preempts T_1 and starts to execute. After some time, T_4 requests the resource CR_2. Since the priority value of T_4 (given to be 20) is greater than CSC (which is 15); T_4 is granted the resource CR_2 (by the resource request clause) and CSC is set to 20. When T_4 completes its execution, T_1 will get a chance to execute.

Situation 2

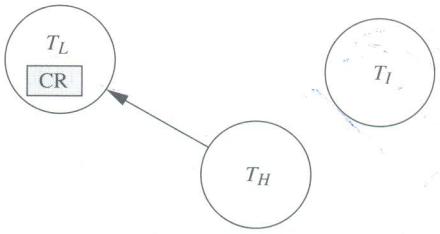
Assume that T_3 becomes ready. Being of higher priority, T_3 preempts T_1 and starts to execute. After some time, T_3 requests for the resource CR_1. As the priority of T_3 (given to be 15) is not greater than CSC (which is 15); T_3 will not be granted CR_1. T_3 would block, and T_1 would inherit the priority of T_3 (by the PCP inheritance clause). So, T_1's priority would change from 10 to 15.

3.6 DIFFERENT TYPES OF PRIORITY INVERSIONS UNDER PCP

Tasks sharing a set of resources using PCP may undergo three important types of priority inversions: direct inversion, inheritance-related inversion, and avoidance inversion.



▲ **FIGURE 3.6**
Direct Priority Inversion



▲ **FIGURE 3.7**
Inheritance–Related Inversion

1. Direct Inversion: Direct inversion occurs when a higher priority task waits for a lower priority task to release a resource that it needs.

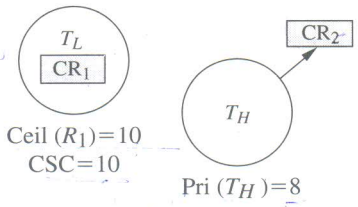
Consider the example shown in Fig. 3.6. Suppose a low priority task T_L is holding a critical resource named CR. Now if a higher priority task T_H needs this resource, then it would have to wait till T_L finishes using CR and releases it. We can see that in this type of inversion, a lower priority task directly causes a higher priority task to undergo inversion by holding the resource it needs.

2. Inheritance-Related Inversion: Consider a situation where a lower priority task is holding a resource and a higher priority task is waiting for it. Then, the priority of the lower priority task is raised to that of the waiting higher priority task by the inheritance clause of PCP. As a result, the intermediate priority tasks not needing the resource undergo inheritance-related inversion.

Inheritance-related inversion has been illustrated in Fig. 3.7. T_H is a higher priority task than T_L, and T_I has priority intermediate between T_L and T_H. T_H and T_L both need the critical resource CR, whereas T_I has no resource requirements. At some point of time in the execution, a low priority task T_L has acquired a resource CR. Therefore, CSC value is set to Ceil (CR) by the resource request clause. A little later, a high priority task T_H requests the resource CR. T_H would block on CR by the resource request clause. By the inheritance clause, T_L would inherit the priority of T_H. Now, consider the intermediate priority task T_I. Though T_I needs no resource, it can not execute due to the raised priority of T_L. In this case, the task T_I is said to undergo inheritance-related inversion.

3. Avoidance-Related Inversion: In PCP, when a task requests a resource its priority is checked against CSC. The requesting task is granted use of the resource only when its priority is greater than CSC. Therefore, even when a resource that a task is requesting is idle, the requesting task may be denied access to the resource if the requesting task's priority is less than CSC. A task whose priority is greater than the currently executing task, but greater than CSC and needs a resource that is currently not in use, is said to undergo avoidance-related inversion.

Avoidance-related inversion is also sometimes referred to as priority ceiling-related inversion, since a higher priority task is not allowed to execute, not because it requests a resource that is already locked by another task, but because its priority is less than the value of CSC. In avoidance-related inversion, a higher priority task blocks for a resource that is not being used by any of the tasks. Though this restriction might appear to be too restrictive and wasteful, a little thought would show that this restriction is necessary to prevent deadlocks. This type of inversion is, therefore, popularly called deadlock avoidance inversion, or simply avoidance inversion; the reason being that the blocking caused due to the priority ceiling rule is the cost for avoidance of deadlock among tasks.



▲ **FIGURE 3.8**
Avoidance-Related Inversion

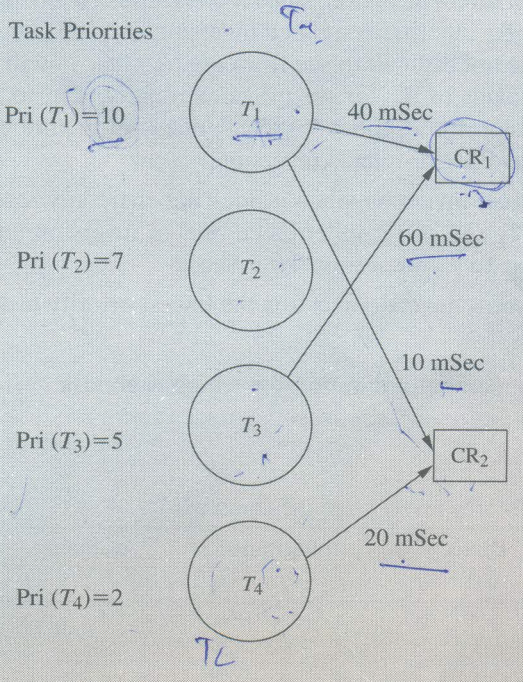
We now illustrate deadlock avoidance inversion using an example. Consider the example shown in Fig. 3.8. In Fig. 3.8, a low priority task (T_L) and a high priority task (T_H) both need two resources R_1 and R_2 during their computations. Assume that the low priority task T_L is presently using a critical resource CR_1. This means that when the resource was granted, CSC must have been set equal to the Ceil (CR_1). Now, when a high priority task T_H requests to use the resource CR_2, it blocks because its priority is less than CSC. As already pointed out, this provision has been incorporated in PCP in order to avoid deadlocks. Not allowing the task T_H to access R_2 precludes the possibility that T_H may later request R_1 and T_L may request R_2 leading to a deadlock. Also, note that if T_H's priority is higher than CSC, then it does mean that T_H will never need R_1 and T_H in that case can safely be allowed to access R_2. The example shows that avoidance-related inversion is the price to pay to avoid deadlocks. However, a task may be denied access to a resource even when a grant of the resource could in no way would have caused a deadlock. That is, it is possible that under PCP a task undergoes avoidance inversion, though its request for a resource in no way could have caused any deadlocks. We illustrate this aspect in the following using an example.

Consider a real-time system in which there are four tasks T_1, T_2, T_3, and T_4. Assume that the priorities of these three tasks are 2, 4, 5, and 10, respectively. Further assume that the tasks T_1 and T_4 share a critical resource CR_1 and that the tasks T_2 and T_3 share a critical resource CR_2. The ceiling priorities of the resources CR_1 and CR_2 are 10 and 5, respectively. Assume that the task T_1 first acquires the resource CR_1. Then, the CSC would be set to 10 (ceiling priority of CR_1) by the resource grant clause. If task T_2 requests CR_2 next, then it would be refused access because T_2's priority is less than CSC. In this case, T_2 would suffer avoidance inversion though there is no possibility of a deadlock even if T_2 was permitted access to CR_2.

Now, let us try to quantitatively determine the duration for which a task may undergo the different types of inversions when resources are shared under PCP. To illustrate how this can be done, we compute the task inversions due to resource sharing through the following two examples. In the analysis regarding priority inversions due to resource sharing, we have assumed that once a task releases a resource, it does not acquire any other resource. In other words, tasks execute in two phases, resource acquire phase followed by resource release phase. In the resource acquire phase, tasks keep on acquiring the resources they need (no release) and in the resource release phase they only release resources (no acquiring). Unless the tasks are enforced to follow this discipline, determination of a quantitative bound on the inversion time would become difficult.

Example 3.2

A system has four tasks: T_1, T_2, T_3, T_4. These tasks need two critical resources CR_1 and CR_2. Assume that the priorities of the four tasks are as follows: pri (T_1) = 10, pri (T_2) = 7, pri (T_3) = 5, pri (T_4) = 2. The four tasks: T_1, T_2, T_3 and T_4 have been arranged in decreasing order of their priorities. That is, pri (T_1) > pri (T_2) > . . . > pri (T_4). The exact resource requirements of these tasks and the duration for which the tasks need the two resources have been shown in Fig. 3.9. Compute the different types of inversions that each task might have to undergo in the worst case.



▲ **FIGURE 3.9**

Task Graph of Example 3.2

Solution. As shown in Fig. 3.9, the task T_1 would require the resource CR_1 for 40 mSec and CR_2 for 10 mSec. The task T_3 would require resource CR_1 for 60 mSec and task T_4 would require CR_2 for 20 mSec. Let us assume FCFS scheduling among equal priority tasks. Now the ceiling of any resource CR can be calculated using Expr. 3.1 as:

$$\text{Ceil (CR)} = \max(\{\text{pri}\{T_i\}/T_i \text{ may need CR}\})$$

Using this, we get Ceil (CR_1) = max ({10, 5}) = 10 and Ceil (CR_2) = max ({10, 2}) = 10. Let us now determine the different types of inversions that each task might suffer, and the duration of the inversions. Considering the tasks one by one:

- Task T_1 can suffer direct inversion (due to the resource CR_1) by task T_3 or (due to the resource CR_2) by T_4. From an inspection of Fig. 3.9 it can be easily seen that T_1 can suffer direct inversion due to T_3 for 60 mSec and due to T_4 for 20 mSec. T_1 will not suffer any inheritance-related inversion, since it is the highest priority task. From a similar reasoning, it is easy to see that T_1 can not undergo deadlock avoidance inversion.
- Task T_2 will not suffer any direct inversion, since it does not need any resource for its computations. But, T_2 can suffer inheritance-related inversion due to T_3 when T_3 acquires CR_1 and T_1 waits for CR_1. It is not difficult to see from Fig. 3.9 that T_2 will suffer inheritance-related inversion on account of T_3 for 60 mSec. Similarly, due to T_4 it can undergo inheritance-related inversion for 20 mSec. Since T_2 does not require any resource, it can not suffer any deadlock avoidance-related inversion.
- T_3 will not suffer any direct inversion since it does not share any resource with a lower priority task. However, T_3 can suffer inheritance related inversion and deadlock avoidance related inversion due to T_4 for at most 20 mSec.
- T_4 will not suffer any priority inversion, as it is the lowest priority task.

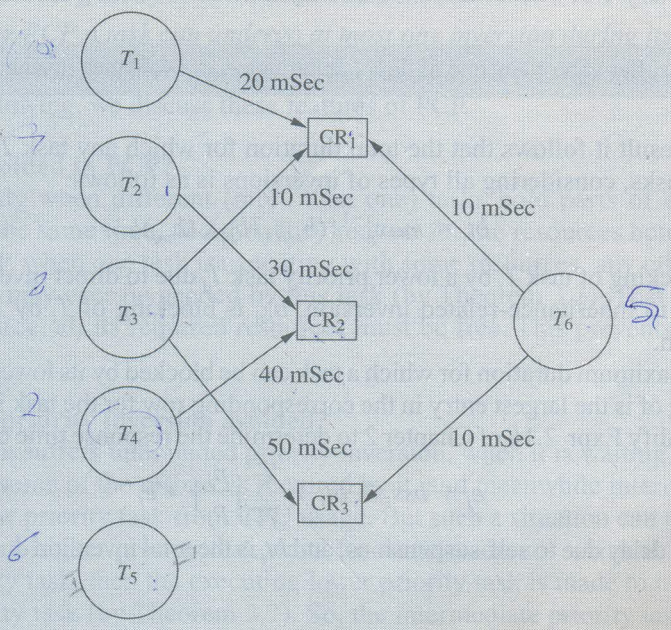
We now represent the maximum duration for which each task suffers from each type of inversion in Table 3.1.

TABLE 3.1 Priority Inversions for Example 3.2

Task	Direct			Inheritance			Avoidance		
	T_2	T_3	T_4	T_2	T_3	T_4	T_2	T_3	T_4
T_1	x	60	20	x	x	x	x	x	x
T_2	x	x	x	x	60	20	x	x	x
T_3	x	x	x	x	x	20	x	x	20

Example 3.3

Let us consider a system with a set of periodic real-time tasks $T_1 \ldots T_6$. The resource and computing requirements of these tasks have been shown in Fig. 3.10. Assume that the tasks $T_1 \ldots T_6$ have been arranged in decreasing order of their priorities. Compute the different types of inversions that a task might have to undergo.



▲ **FIGURE 3.10**
Resource Sharing Among Tasks of Example 3.3

Solution. For each of the given tasks we have computed the maximum duration for which it might have to suffer from different types of inversions and have represented those in Table 3.2.

TABLE 3.2 Different Types of Inversions for Example 3.3

Task	Direct					Inheritance					Avoidance				
	T_2	T_3	T_4	T_5	T_6	T_2	T_3	T_4	T_5	T_6	T_2	T_3	T_4	T_5	T_6
T_1	x	10	x	x	10	x	x	x	x	x	x	x	x	x	x
T_2	x	x	40	x	x	x	10	x	x	10	x	10	x	x	10
T_3	x	x	x	x	10	x	x	40	x	10	x	x	40	x	10
T_4	x	x	x	x	x	x	x	x	x	10	x	x	x	x	10
T_5	x	x	x	x	x	x	x	x	x	10	x	x	x	x	x

Let us now examine some of the properties of an inversion table.

- Each inversion table is an upper triangular matrix. The lower triangular items of each table are all zero. This structure of the inversion tables is expected since a lower priority task can not suffer any inversions due to a higher priority task.
- The avoidance table is similar to inheritance table except for the case of a task that does not need any resource. The reason for this is again not very far to seek. The tasks not needing any resource, never request for any resource, consequently, they can not suffer avoidance inversions.

From Corollary 1 of Theorem 3.2 we already had the following result:

A task can suffer at best any one of direct, inheritance, or avoidance inversions.

From this result it follows that the total duration for which any task T_i may be blocked by lower priority tasks, considering all types of inversions is as follows:

$$bt_i = max_j^{i-1}\{(b_{idj}), (b_{iij}), (b_{iaj})\} \quad (3.5)$$

where b_{idj} is blocking of task T_i by a lower priority task T_j due to direct inversion, b_{iij} is blocking of T_i by T_j due to inheritance-related inversion, b_{iaj} is blocking of T_i by T_j due to avoidance-related inversion.

Thus, the maximum duration for which a task can be blocked by its lower priority tasks on account of sharing of is the largest entry in the corresponding row for the task in the inversion table. We can now modify Expr. 2.16 of Chapter 2 to determine the response time of a task T_i as follows:

$$e_i + bss_i + bt_i + \sum_{j=1}^{i-1}\left\lceil \frac{p_i}{p_j} \right\rceil * e_j \quad (3.6)$$

where bss_i is the delay due to self-suspensions, and bt_i is the total inversion due to resource sharing.

3.7 IMPORTANT FEATURES OF PCP

In this section, we discuss some important features of PCP. We first prove that tasks are single blocking on account of resource usage.

THEOREM 3.2 *Tasks are single blocking under PCP.*

PROOF Consider that a task T_i needs a set of resources SR = $\{R_i\}$. Obviously, the ceiling of each resource R_i in SR must be greater than or equal to pri (T_i). Now, assume that when T_i acquires some resource R_i, another task T_j was already holding a resource R_j, and that R_i, $R_j \in$ SR. Such a situation would lead T_i to block after acquiring a resource. But, when T_j locked R_j CSC should have been set to at least pri(T_i) by the resource grant clause of PCP and T_i could not have been granted R_i. This is a contradiction with the premise. Therefore, when T_i acquires one resource all resources required by it must be free.

We can in a similar way show that once a task T_i acquires a resource, it can not undergo any inheritance-related inversion. Assume that a lower priority task T_k is holding some resource R_z and a higher priority task T_h is waiting for the resource. But, this is not possible as Ceil(R_2) must be at least as much as pri (T_h). The CSC should, therefore, have been set to a value that is at least as much as pri (T_h) and T_i would have been prevented from accessing the resource R_i in the first place. This is a contradiction with the premise we started with. Therefore, it is not possible that a task undergoes inheritance-related inversion after it acquires a resource.

Using a similar reasoning, we can show that a task can not suffer any avoidance inversion after acquiring a resource.

Thus, once a task acquires a resource it can not undergo any inversion. It is, therefore, clear that tasks under PCP are single blocking.

The following corollary easily follows from Theorem 3.2.

Corollary 1. *Under PCP a task can undergo at most one inversion during its execution.*

Priority Ceiling Protocol is free from deadlocks, unbounded priority inversions, and chain blockings. In the following, we discuss these features of PCP.

How is deadlock avoided in PCP?
Deadlocks occur only when different (more than one) tasks hold parts of each other's required resources at the same time, and then they request for the resources being held by each other. But under PCP, when one task is executing with some resources, any other task can not hold a resource that may ever be needed by this task (by Theorem 3.2). That is, when a task is granted one resource, all its required resources must be free. This prevents the possibility of any deadlock.

How is unbounded priority inversion avoided?
A higher priority task suffers unbounded priority inversion, when it is waiting for a lower priority task to release some of the resources required by it, and meanwhile intermediate priority tasks preempt the low priority task from CPU usage. But such a situation can never happen in PCP since whenever a higher priority task waits for some resources which is currently being used by a low priority task, then the executing lower priority task is made to inherit the priority of the high priority task (by Theorem 3.2). So, the intermediate priority tasks can not preempt lower priority task from CPU usage. Therefore, unbounded priority inversions can not occur under PCP.

How is chain blocking avoided?
By Theorem 3.2, resource sharing among tasks under PCP is single blocking.

3.8 SOME ISSUES IN USING A RESOURCE SHARING PROTOCOL

In this section we discuss some issues that may arise while using resource sharing protocols to develop a real-time application requiring tasks to share non-preemptable resources. The first issue that we discuss is how to use PCP in dynamic priority systems. Subsequently, we discuss the situations in which the different resource sharing protocols would be useful.

3.8.1 Using PCP in Dynamic Priority Systems

So far in all our discussions regarding usage of PCP, we had implicitly assumed that the task priorities are static. That is, a task's priority does not change for its entire lifetime—from the time it arrives to the time it completes. In dynamic priority systems, the priority of a task might change with time. As a consequence, the priority ceilings of every resource needs to be appropriately recomputed each time a task's priority changes. In addition, the value of the CSC and the inherited priority values of the tasks holding resources also need to be changed. This represents a high runtime overhead. The high runtime overhead makes use of PCP in dynamic priority systems unattractive.

3.8.2 Comparison of Resource Sharing Protocols

We have so far discussed three important resource sharing protocols for real-time tasks: priority inheritance protocol (PIP), highest locker protocol (HLP) and priority ceiling protocol (PCP). The shortcomings and advantages of these protocols are discussed below.

- Priority Inheritance Protocol (PIP): This is a simple protocol and effectively overcomes the unbounded priority inversion problem of traditional resource sharing techniques. This protocol requires the minimal support from the operating system among all the resource sharing protocols we discussed. However, under PIP tasks may suffer from chain blocking and PIP also does not prevent deadlocks.
- Highest Locker Protocol (HLP): HLP requires only moderate support from the operating system. It solves the chain blocking and deadlock problems of PIP. However, HLP can make the intermediate priority tasks undergo large inheritance-related inversions and can, therefore, cause tasks to miss their deadlines.
- Priority Ceiling Protocol (PCP): PCP overcomes the shortcomings of the basic PIP as well as HLP protocols. PCP protocol is free from deadlock and chain blocking. In PCP priority of a task is not changed until a higher priority task requests the resource. It, therefore, suffers much lower inheritance-related inversions than HLP.

From the above discussions we can infer that PCP is well-suited for use in applications having large number of tasks and many critical resources, while PIP being a very simple protocol is suitable to be used in small applications.

3.9 HANDLING TASK DEPENDENCIES

An assumption that was implicit in all the scheduling algorithms we discussed in Chapter 2 is that the tasks in an application are independent. That is, there is no constraint on the order in which the different tasks can be taken up for execution. However, this is far from true in practical situations, where one task may need results from another task, or the tasks might have to be carried out in certain order for the proper functioning of the system. When such dependencies among tasks exist, the scheduling techniques discussed in Chapter 2 turn out to be insufficient and need to be suitably modified.

We first discuss how to develop a satisfactory schedule for a set of tasks that can be used in table-driven scheduling.

Table-driven algorithm: The following are the main steps of a simple algorithm for determining a feasible schedule for a set of periodic real-time tasks whose dependencies are given:

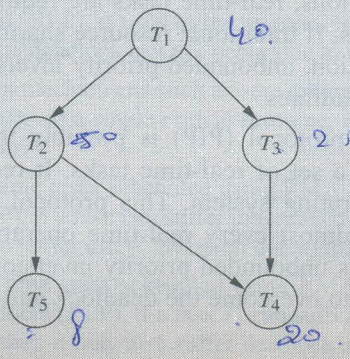
```
1. Sort task in increasing order of their deadlines, without violating any
   precedence constraints and store the sorted tasks in a linked list.
2. Repeat
        Take up the task having largest deadline and not yet
        scheduled (i.e., scan the task list of step 1 from left).
        Schedule it as late as possible.
   until all tasks are scheduled.
3. Move the schedule of all tasks to as much left (i.e., early) as
   possible without disturbing
   their relative positions in the schedule.
```

We now illustrate the use of the above algorithm to compute a feasible schedule for a set of tasks with dependencies.

Example 3.4

Determine a feasible schedule for the real-time tasks of a task set $\{T_1, T_2, \ldots, T_5\}$ for which the precedence relations have been shown in Fig. 3.11 for use with a table-driven scheduler. The execution times of the tasks $T_1, T_2, \ldots, T_5$ are: 7, 10, 5, 6, 2 and the corresponding deadlines are 40, 50, 25, 20, 8, respectively.

Solution. The different steps of the solution have been worked out and shown in Fig. 3.12.

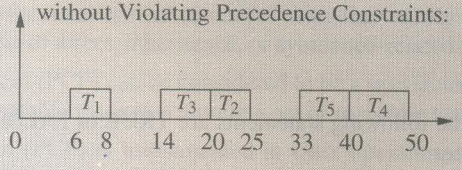


▲ **FIGURE 3.11**
Precedence Relationship Among Tasks for Example 3.4

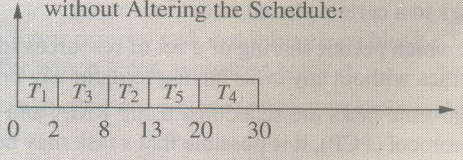
Step 1: Arrangement of Tasks in Ascending Order:

T_1 T_3 T_2 T_5 T_4

Step 2: Schedule Tasks as Late as Possible without Violating Precedence Constraints:



Step 3: Move Tasks as Early as Possible without Altering the Schedule:



▲ **FIGURE 3.12**
Solution of Example 3.4

EDF and RMA-based Schedulers: Precedence constraints among tasks can be handled in both EDF and RMA through the following modification to the algorithm.

- Do not enable a task until all its predecessors complete their execution.
- Check the tasks waiting to be enabled (on account of its predecessors completing their executions) after every task completes.

We, however, must remember that the achievable schedulable utilization of tasks with dependencies would be lower compared to when the tasks are independent. Therefore, the schedulability results worked out in the last chapter would not be applicable when task dependencies exist.

SUMMARY

- In many real-life applications, real-time tasks are required to share non-preemptable resources among themselves. If traditional resource sharing techniques such as semaphores are deployed in this situation, unbounded priority inversions might occur leading to real-time tasks to miss their deadlines.
- The priority inheritance protocol (PIP) is possibly the simplest protocol for sharing critical resources among a set of real-time tasks. It requires minimal support from the underlying real-time operating system. This protocol for sharing critical resources is, therefore, supported by almost every real-time operating systems. The priority inheritance mechanism prevents unbounded priority inversions. However, it becomes the programmer's responsibility to overcome the deadlock and chain blocking problems through careful programming.
- PCP incorporates additional rules in the basic PIP and overcomes the deadlock, unbounded priority inversion, and chain blocking problems of PIP. However, under PCP tasks might still suffer moderate inversions. PCP has been widely accepted as a popular resource sharing protocol and is normally used for developing moderate and large real-time applications. PCP is supported by many modern commercial real-time operating systems.
- Task dependencies can be handled through minor alterations to the basic task scheduling algorithms. However, the achievable utilization of the schedulers drop when tasks are not independent.

EXERCISES

1. Determine whether the following statements are TRUE or FALSE. In each case justify your choice in one or two sentences.
   (a) Rate monotonic scheduling can satisfactorily be used for scheduling access of several hard real-time periodic tasks to a certain shared critical resource.
   (b) Algorithms exist which permit sharing of a set of resources in the exclusive mode among tasks of different priorities without any tasks having to incur any priority inversion.
   (c) When a set of real-time tasks are scheduled using RMA and share critical resources using the priority ceiling protocol (PCP), it is possible that a task may be denied access to a resource even when the resource is not required by any other task.
   (d) Highest locker protocol (HLP) maintains the tasks waiting for a shared non-preemptable resource in FIFO order.

(e) When a set of periodic real-time tasks scheduled using RMA share certain critical resources using priority inheritance protocol (PIP), some of the tasks might suffer unbounded priority inversions.

(f) In the highest locker protocol (HLP), for each critical resource a separate queue needs to be maintained for the tasks waiting for the resources.

(g) Using the basic priority inheritance protocol (PIP) for scheduling access of a set of real-time tasks to a set of non-preemptable resources would result in tasks incurring unbounded priority inversions.

(h) When a set of static priority periodic real-time tasks scheduled using RMA share some critical resources using the priority ceiling protocol (PCP), the time duration for which a task can undergo inheritance-related inversion is exactly the same as the duration for which it can undergo deadlock avoidance-related inversion.

(i) Suppose a task needs three non-preemptable shared resources CR1, CR2, and CR3 during its computation. Under highest locker protocol (HLP), once the task acquires one of these resources, it is guaranteed not to block for acquiring the other required resources.

(j) When highest locker protocol (HLP), is used as the protocol for sharing some critical resources among a set of real-time tasks, deadlock can not occur due to resource sharing.

(k) If traditional resource sharing mechanisms such as semaphores and monitors are used to share access of several real-time tasks to a single critical resource, neither unbounded priority inversions, nor deadlocks can occur.

(l) When a set of periodic real-time tasks scheduled using RMA share critical resources using the priority ceiling protocol (PCP), if a task can suffer inheritance blocking by another task for certain duration, then it may also suffer direct inversion due to that task for the same duration.

(m) When a set of periodic real-time tasks scheduled using RMA share certain critical resources using the priority ceiling protocol (PCP), if a task T_1 can suffer inheritance blocking by another task T_2 for certain duration, then it may also suffer deadlock avoidance inversion for the same duration due to T_2 excepting if T_1 does not need any resource.

(n) When a set of real-time tasks share certain critical resources using the priority ceiling protocol (PCP), the highest priority task does not suffer any inversions.

(o) When a set of real-time tasks share critical resources using the priority ceiling protocol (PCP), the lowest priority task does not suffer any inversions.

(p) It is possible that in a real-time system even the lowest priority task may suffer unbounded priority inversions unless a suitable resource-sharing protocol is used.

(q) When a set of real-time tasks share critical resources using the priority ceiling protocol (PCP), a task can undergo at best one of direct, inheritance, or avoidance-related inversion due to any task.

(r) The priority ceiling protocol (PCP) can be considered to be a satisfactory protocol to share a set of serially reusable preemptable resources among a set of real-time tasks.

(s) If priority ceiling protocol (PCP) is implemented in Unix operating systems, then the ceiling priority value of a critical resource would be the maximum of the priority values of all the tasks using this resource.

(t) The duration for which a lower priority task can inheritance block a higher priority task is also identical to the duration for which it can avoidance block it.

(u) Under PCP even a task which does not require any resource can undergo priority inversion for some duration.

2. Explain the problems that might arise if hard real-time tasks are made to share critical resources among themselves using traditional operating system primitives such as semaphores or monitors. Briefly explain how these problems can be overcome.

3. Explain using an appropriate example as to why a critical resource can get corrupted if the task using it is preempted, and then another task is granted use of the resource.
4. What do you understand by the term "priority inversion" in the context of real-time task scheduling? When several tasks share a set of critical resources, is it possible to avoid priority inversion altogether by using a suitable task scheduling algorithm? Explain your answer.
5. Explain the operation of priority ceiling protocol (PCP) in sharing critical resources among real-time tasks. Explain how PCP is able to avoid deadlock, unbounded priority inversions, and chain blockings.
6. Explain the different types of priority inversions that a task might suffer due to a lower priority task when the priority ceiling protocol (PCP) is used to share critical resources among a set of real-time task. Can a task suffer both inheritance-related inversion and direct inversion due to some lower priority task? If you answer in the affirmative, construct a suitable example to corroborate your answer. If you answer in the negative, explain why.
7. Define the terms *priority inversion* and *unbounded priority inversion* as used in real-time operating systems. Is it possible to devise a resource-sharing protocol which can guarantee that no task undergoes: (i) priority inversion? (ii) unbounded priority inversion? Justify your answers.
8. What do you understand by *inheritance-related inversion?* Explain how it can arise when resources are shared in exclusive mode in a real-time environment. Can inheritance-related inversion be eliminated altogether? If your answer is "yes", explain how? If your answer is "no", then explain how inheritance-related inversions can be contained?
9. Using two or three sentences explain how PCP can be used for resource sharing among a set of tasks when the tasks are scheduled using EDF. Can your solution be used in practical situations? If not, why?
10. When EDF is used for task scheduling in a real-time application, explain a scheme by which sharing of critical resources among tasks can be supported. Give an algorithm in pseudo-code notation to describe the steps to handle resource grant and release.
11. A set of hard real-time periodic tasks need to be scheduled on a uniprocessor using RMA. The following table contains the details of these periodic tasks and their use of three non-preemptable shared resources. Can the tasks T_2 and T_3 meet their respective deadlines when priority ceiling protocol (PCP) is used for resource scheduling?

Task	p_i	e_i	R_1	R_2	R_3
T_1	400	30	15	20	
T_2	200	25	—	20	10
T_3	300	40	—	—	—
T_4	250	35	10	10	10
T_5	450	50	—	—	5

p_i indicates the period of task T_i and e_i indicates its computation time. The period of each task is the same as its deadline. The entries in the R_1, R_2, and R_3 columns indicate the time duration for which a task needs the named resource in non-preemptive mode. Assume that after a task releases a resource, it does not acquire the same or any other resource.

12. While it is generally true that avoidance inversion in PCP is the price paid to avoid situations leading to deadlocks, sometimes tasks might undergo avoidance inversion when their request for a resource can in no way cause deadlocks. Illustrate this by constructing an example to show that even when

there is no chance of any deadlocks being caused, a task under PCP might still undergo avoidance inversion.

13. A set of periodic tasks need to be scheduled on a uniprocessor. The following table contains the details of these periodic tasks and their use of non-preemptable shared resources R_1, R_2, and R_3. Can task T_3 meet its deadline when the tasks are scheduled under RMA and priority ceiling protocol (PCP) is used for resource scheduling?

Task	p_i	e_i	R_1	R_2	R_3	s_i
T_1	400	30	15	20		10
T_2	200	25	—	20	10	20
T_3	300	40	—	—	—	10
T_4	250	35	10	10	10	40
T_5	450	50	—	—	5	55

p_i indicates the period of task T_i and e_i indicates its computation time. The entries in the R_1, R_2, and R_3 columns indicate the time duration for which a task needs the named resource in non-preemptive mode, s_i is the self suspension time of task T_i. All time units have been specified in milliseconds.

14. A set of periodic tasks need to be scheduled on a uniprocessor. The following table contains the details of these periodic tasks and their use of non-preemptable shared resources. Can the task T_3 meet its deadline when the tasks are scheduled under RMA and the priority ceiling protocol (PCP) is used for resource scheduling?

Task	p_i	e_i	R_1	R_2	R_3
T_1	400	30	15	20	
T_2	200	25	—	20	10
T_3	300	40	—	—	—
T_4	250	35	10	10	10
T_5	450	50	—	—	5

p_i indicates the period of task T_i and e_i indicates its computation time. The entries in the R_1, R_2, and R3 columns indicate the time duration for which a task needs the named resource in non-preemptive mode. All time units have been specified in milliseconds.

15. Consider a real-time system whose task characteristics and dependencies are described in the following table. Assume that the tasks have zero phasing and repeat with a period of 90 mSec. Determine a feasible schedule which could be used by a table-driven scheduler.

Task	Computation Time (e_i) mSec	Deadline (d_i) mSec	Dependency
T_1	30	90	—
T_2	15	40	T_1, T_3
T_3	20	40	T_1
T_4	10	70	T_2

16. Consider a real-time system in which five real-time periodic tasks $T_1 \ldots T_5$ have zero phasing and repeat with a period of 150 mSec. The task characteristics and dependencies are described in the following table. Determine a feasible schedule which could be used by a table-driven scheduler.

Task	Computation Time (e_i) mSec	Deadline (d_i) mSec	Dependency
T_1	15	40	—
T_2	30	70	T_1
T_3	10	90	T_2
T_4	20	40	T_1

17. Consider a real-time system whose task characteristics and dependencies are described in the following table. These tasks repeat every 150 mSec. Determine a feasible schedule which could be used by a table-driven scheduler.

Task	Computation Time (e_i) mSec	Deadline (d_i) mSec	Dependency
T_1	10	50	—
T_2	10	80	T_1
T_3	30	60	T_1
T_4	50	150	T_3, T_2
T_5	35	140	T_2