HANDLING RESOURCES 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 tasks need 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 accelaration 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 resources can not be preempted at any time from resource usage without risking the correctness of the computed results.

Resources Sharing among Real-time Task.

→ Real time task need to share some resources among themselves. Often these shared resources need to be use by the individual task in exclusive mode

→ This means that a task that is using resources, can not immediately handover the resources to an other task that request the resources at any arbitary point in time. But it can do so only after it complete it's use of the Resources.

→ if a task is preempted from using a Resources before it completes it use of the Resources, then the resources can become corrupted.

→ Sharing a critical resources among task requires a different set of rules, EDF and RMA being the popular methodology, once a task gain access to a serially reusable resources, such a CPU, it uses it exclusively. That is two different task can not run once CPU at the same time

→ Another important feature of serially reusable resources is that task executing on a CPU can be preempted and restarted at a later time without any problem.

→ A non preemptable resources is also used in exclusive mode. However task using a non-preemptable resources can not be preempted from the resources usage. Otherwise the resources would become inconsistent and can lead to system failure.

→ Therefore, when lower priority task has already gained access to a non preempted resource and using it, even a higher priority task would have to wait until the lower priority task using the resources completes.

PRIORITY INVERSION

⇒ The mechanisms popularly employed to achieve mutually exclusive use of data and resources among a set of tasks include semaphores, locks, and monitors 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.

→ 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 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.

→ Fortunately, a single simple priority inversion can occur is bounded by the longest duration for which a lower priority task might need a critical resource an exclusive mode.

→ 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 inversions can upset all calculations of programmer regarding the worst case response time of a real-time task and cause it to miss its deadline.

→ 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 assume that there are several tasks $T_{I1}, T_{I2}, T_{I3}, \ldots$ that have priorities intermediate between T_H and T_L, and do

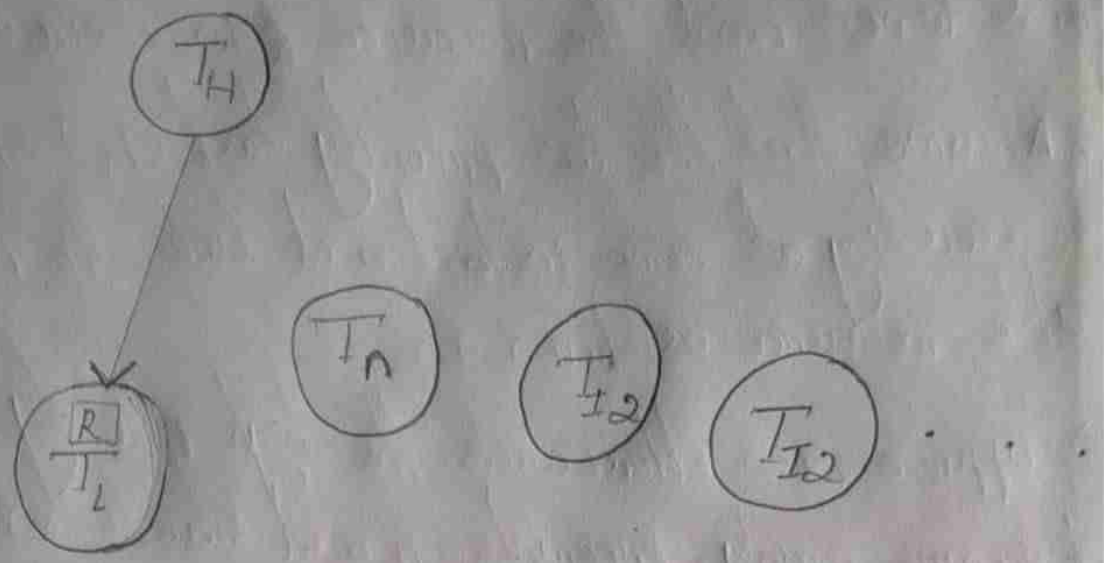


FIGURE 3.1

⇒ Unbounded priority Inversion

- Unbounded priority inversion is an important problem that real-time system developer face.

- 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).

→ A time t_0, the low priority task T_5 is executing and is using a non-preemptable resource (CR) the higher priority task T_1 arrives, preempts T_5

And do not need the resource R in their computations

→ Assume that the low priority task (T_L) starts executing at some instant and locks the non-preemptable resource as shown in

→ Suppose soon afterwards, the high priority task (T_H) becomes ready, preempts T_L and starts executing.

→ 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..

→ 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. 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. intermediate only for the low priority task (T_L) to completing

→ The high priority task having to wait for the required resource for a considerable priority task to complete.

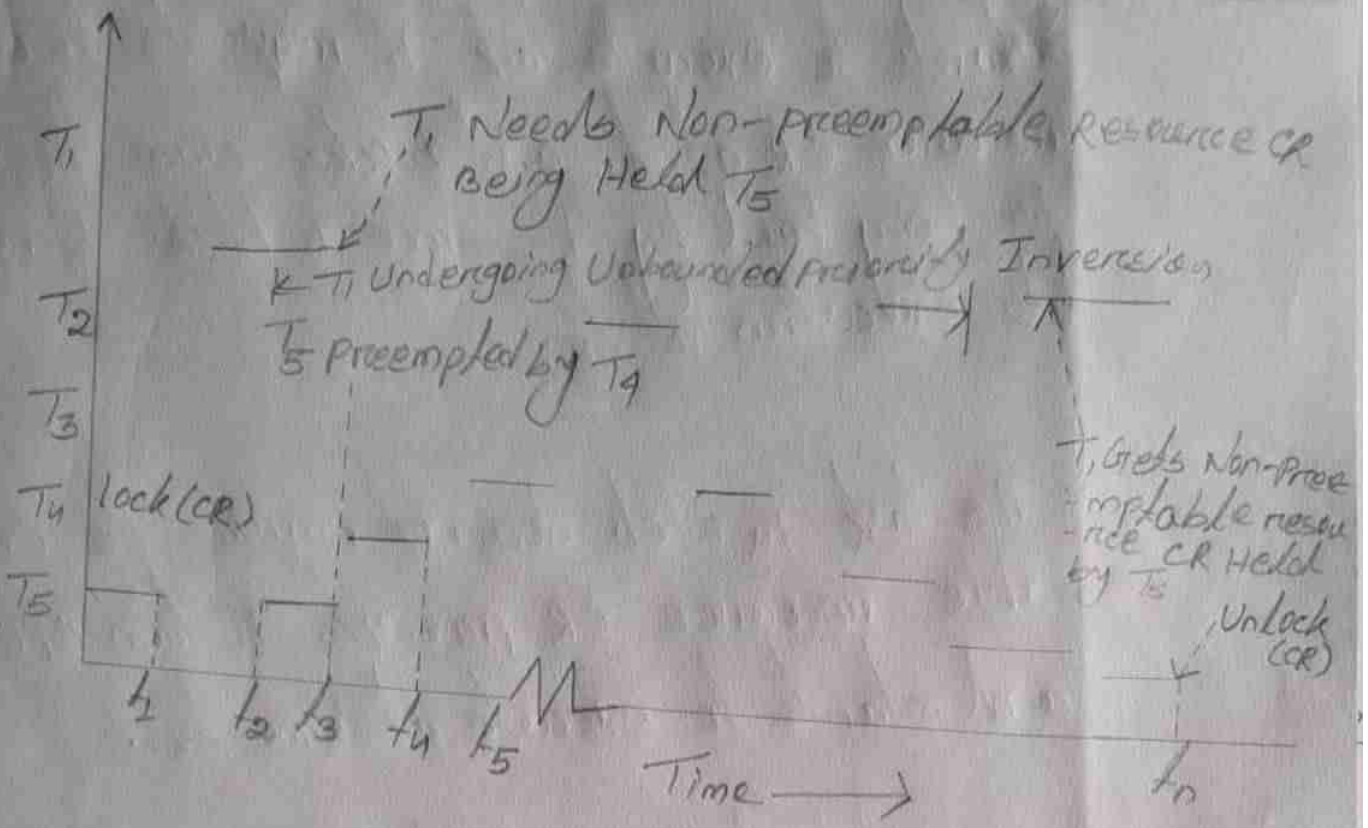


Fig 3.2

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.

→ 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).

→ 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.

→ 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

Priority Inheritance Protocol (PIP):

⇒ The basic Priority Inheritance Protocol is a simple technique to share critical resources among tasks without incurring unbounded Priority inversions.

⇒ 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 it's usage of the critical resource as early as Possible without having to suffer preemptions from the intermediate Priority task. When several tasks are waiting for a resource, the tasks holding the resource inherits the highest Priority of all tasks waiting for the resource (if this Priority is greater than it's 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 it's 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.

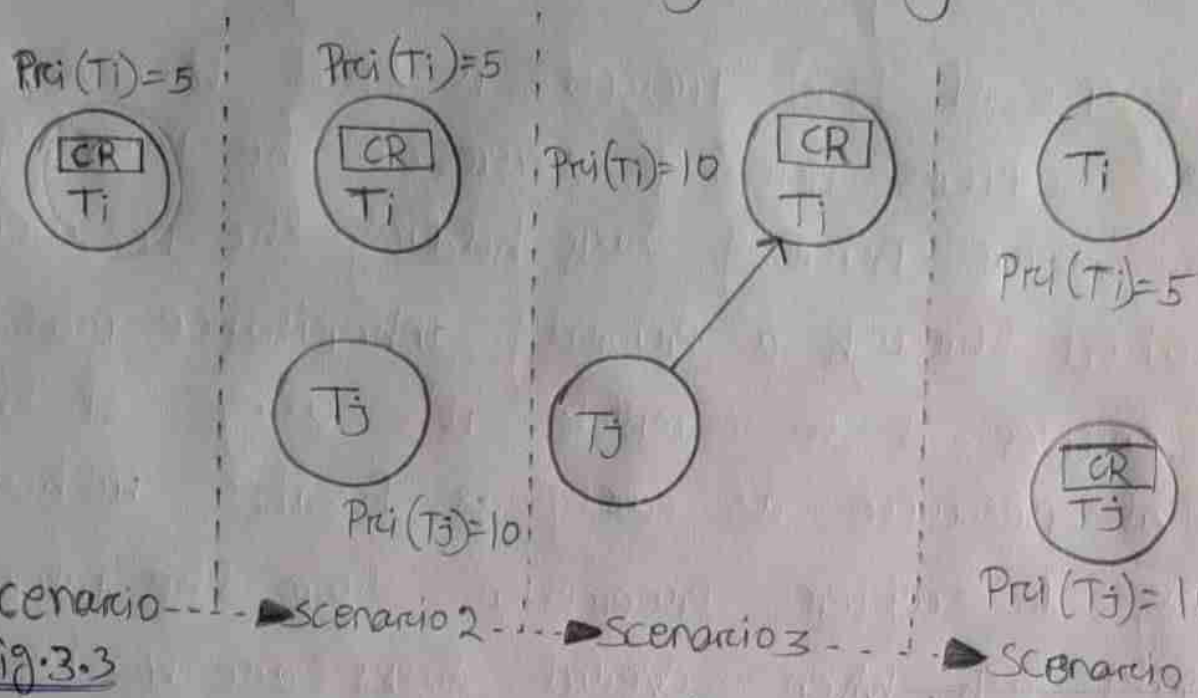


fig. 3.3

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 fig 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 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 scenario.

In fig. 3.4 the Priority changes of task is 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 are 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 it's usage of the resource at t_4 and as soon as it releases the resource, it gets back it's original Priority. It should be noted that a lower Priority task

retains the inherited Priority, untill it holds the resource required by the waiting highere Priority task. whenevere more than one highere 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 problem: deadlock and chain blocking.

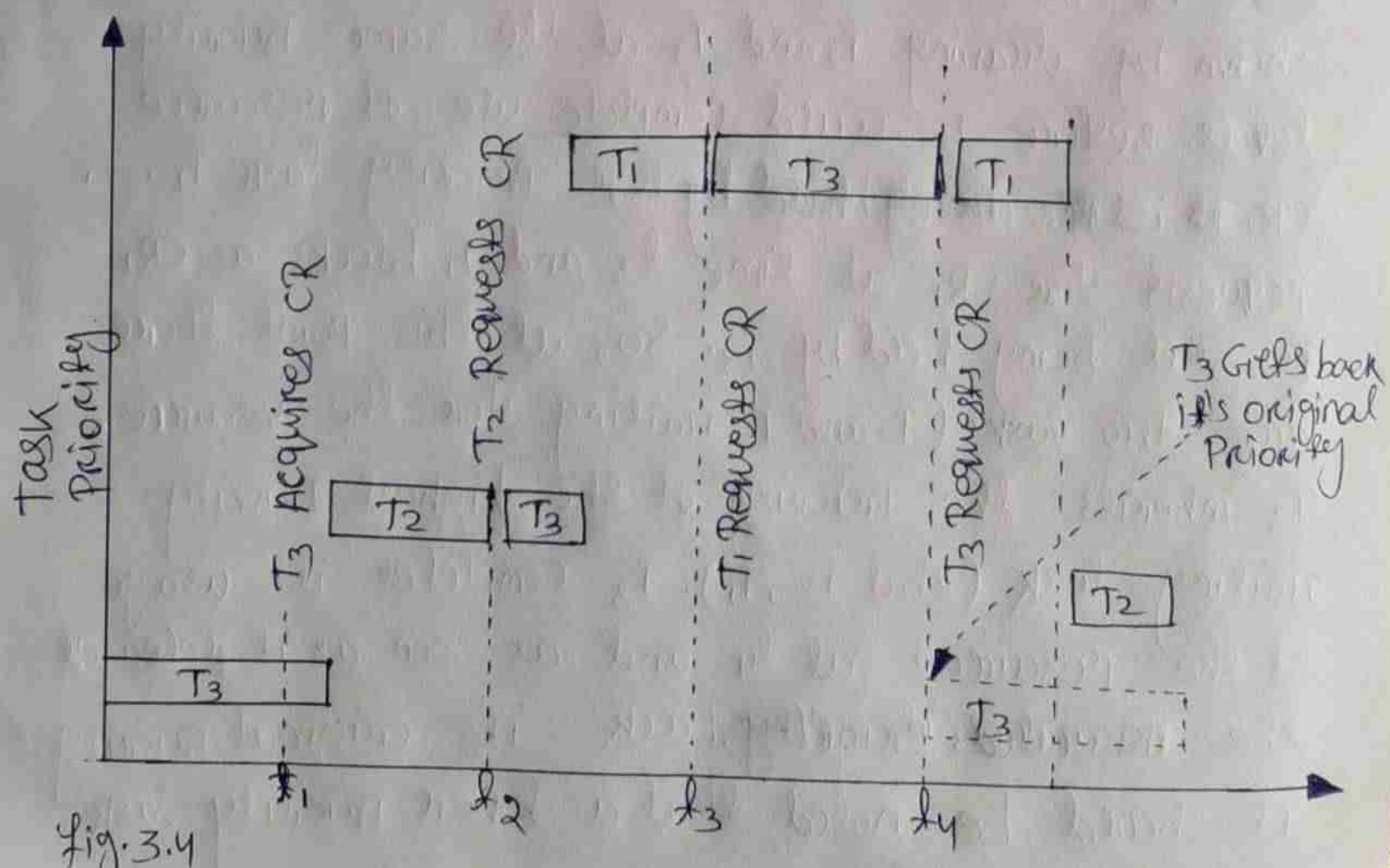


fig. 3.4

Priority changes under Priority Inheritance Protocol.

<u>Deadlock</u>:- 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 it's 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.

→ So, if a task needs 'n' no. of resources for its computed then the task might have to undergo priority inversion for 'n' times.

→ Let us now explain how chain blocking can occure using the example shown in fig. 3.5. Assume that a task, needs several resources. In the first snapshot (shown in fig. 3.5), a low priority task T_2 is holding two resources CR_1 it's priority reduces to it's original priority and T_1 is able to clock 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 it's required resources.

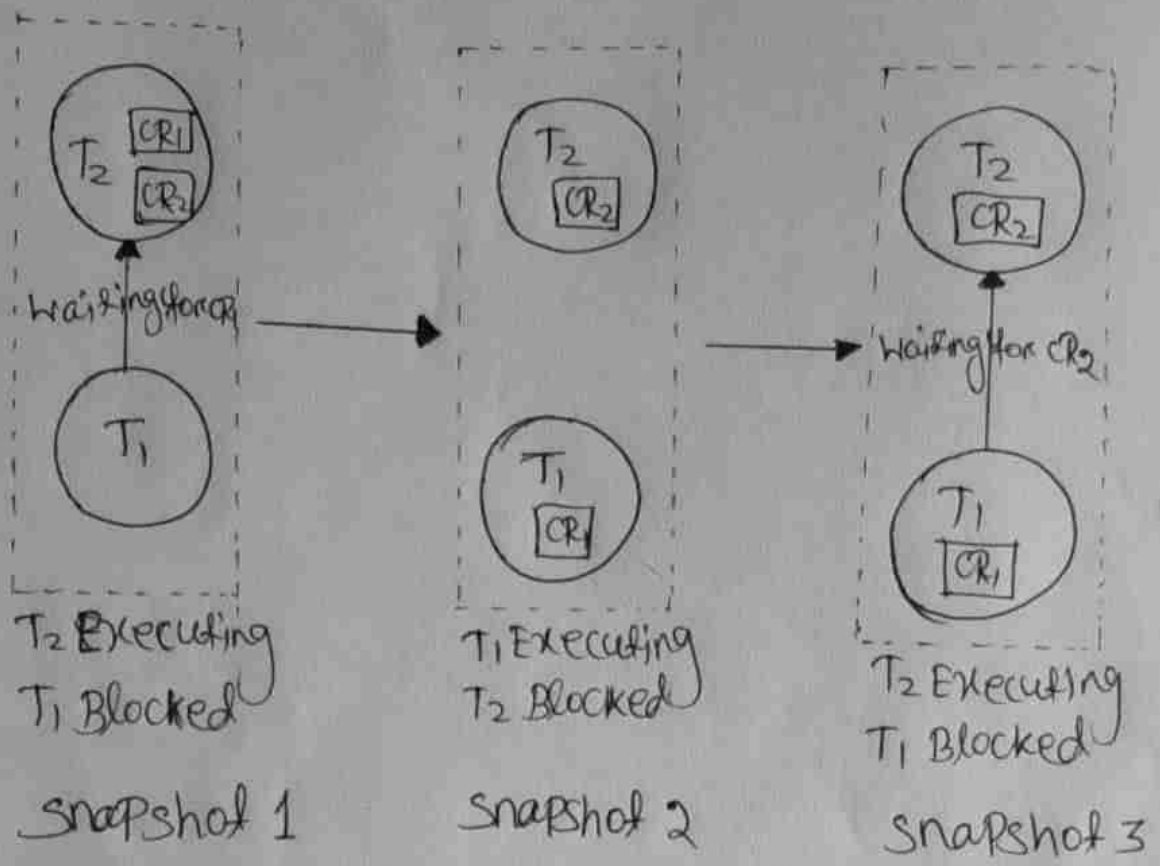


fig. 3.5

chain Blocking in Priority Inheritance Protocol

HIGHEST LOKER PROTOCOL (HLP)

⇒ HLP is an extension of PIP and overcomes some of the shortcomings of PIP. In HLP every critical resource is assigned a celling 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 schedulars 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:

$$ceil(R_i) = max(\{Pri(T_j) / T_j \text{ needs } R_i\})$$

⇒ If higher priority values indicate lower priorities (as in Unix), then the ceiling priority would by the minimum priority of all tasks needing the resource.

$$ceil(R_i) = min(\{Pri(T_j) / T_j \text{ needs } R_i\})$$

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

$$ceil(R_i) = max(\{Pri(T_j) / T_j \text{ needs } R_i\}) + 1$$

⇒ 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.

$$ceil(R_i) = min(\{Pri(T_j) / T_j \text{ needs } R_i\}) + 1$$

⇒ Example:

The system with FCFS scheduling among equal priority task. 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 ceiling Priority of CR_1 is $ceil(CR_1) = max(\{10, 5, 2\}) = 10$.

⇒ Therefore, as soon 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 ceiling Priorities of all the resources it is holding.

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.

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. so 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.

⇒ ceil (CR_2) must be greater than Pri (T_1) is one more than the maximum of the Priority of all tasks using the resource (that include (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 be assumption that T_1 is executing while T_2 is holding the resource CR_2.

⇒ 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.

⇒ So 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 blocking on account of resource sharing.

⇒ 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

1. under HLP, before a task can acquire one resource, all the resources that might be required by it must be free.
2. A task can not undergo chain blocking in HLP.

⇒ 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 source.

⇒ when a resource becomes free, the longest waiting task in the queue is granted the resources. 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 resources, it executes at the ceiling priority of the resource, and other tasks that may need this source do not even a get chance to execute and request for the resource.

Short comings 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 no needing the resource can not execute and are said to undergo inheritance-related Priority inversion.

⇒ 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, 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 CR_1 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 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 after 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, severe 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 this text as the foundation for understanding the priority ceiling protocol that is very popular and being used extensively in real-time application developments.

Priority Ceiling Protocol (PCP) :-

Priority Ceiling Protocol (PCP) of PIP and HIP to solve the Problems of unbounded priority inversion, chain blocking, deadlock and at the same time minimizing inheritance-related inversions. The basic difference between PIP and PCP is that PIP is greedy-based approach and PCP is no. In PCP, the resource is not granted to a process requesting it even if the resource is free.

As in case of HIP, the PCP associates a ceiling value ceil (CR_i) with every resource CR_i, that is the maximum of the priority values of all tasks use CR_i, We use an operating system variable called CSC (current system ceiling) to keep track of the maximum ceiling value of as 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 }).

- The PCP Protocol has 2 rules

1. Resource grant rule
2. Resource release rule

1. Resource grant rule : It has 2 clauses
   (i) Resource request clause
   (ii) Inheritance clause

(i) 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, Priority (T_i) > csc). In both (a) and (b), T_i is granted access to the resource CR_j, and,

i.e. If csc < ceil (CR_j), then, csc is set to ceil (CR_j)

(ii) 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.

2. 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 the 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.

Example:

Consider a system consisting of 4 real-time tasks T_1, T_2, T_3, T_4, sharing two non-preemptable resources CR_1 and CR_2. CR_1 is used by T_1, T_2 and T_3, and CR_2 is used by T_1 and T_4. The priority values of T_1, T_2, T_3 and T_4 are 10, 12, 15 and 20 respectively. Use FCFS scheduling among equal priority tasks where higher priority values indicates higher priorities. Calculate the ceiling priority of CR_1 and CR_2.

From the given data, the ceiling priority of the two resources are

$ceil(CR_1) = max(Pri(T_1), Pri(T_2), Pri(T_3)) = 15$

$ceil(CR_2) = max(Pri(T_1), Pri(T_4)) = 20$

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 $ceil(CR_1) = 15$

Situation 1:

Higher priority T_4, preempts T_1 execute the resource CR_2. The priority value of T_4 (20) is greater than csc (15); T_4 is granted the resource CR_2 and csc is set to 20. T_1 will get a chance to execute.

Situation 2:

Assume that T_3 is higher priority, T_3 preempts T_1 and execute, T_3 request for the resource CR_1. The priority of T_3 (15) is not greater csc; T_3 will not granted CR_1. T_3 would block, T_1 would inherit priority of T_3.

This is priority inversion problem and T_1 will inherit the priority of T_3. So T_1's priority will change from 10 to 15

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

PROOF

- consider that a task T_i needs a set of resources $SR = \{R_i\}$ The ceiling of each resource R_i in SR must be greater than or equal to $pri(T_i)$ assume that when T_i acquires some resource R_i, another task T_j was already holding a resource R_j, and that $R_i, R_j \in SR$. 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.
- Assume that a lower priority task T_k is holding some resource R_2 and a higher priority task T_h is waiting for the resource. 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. Therefore, it is not possible that a task undergoes inheritance-related inversion after it acquires a resource. We can show that a task can not suffer any avoidance inversion after acquiring a resource.

THEOREM

Corollary

- Priority ceiling protocol is free from deadlock, unbounded priority inversion, and chain blockings. In the following. We discuss these features of PCP.

How is deadlock avoided in PCP?

- Deadlock occur only when different 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, any other task can not hold a resource that may ever be needed by this task.
That is. When a task is granted one resource, all it's 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. and meanwhile intermediate priority tasks to release some of the resources preempt the low priority task from cpu usage. cpu since whenever a higher priority task waits for some resources which is made to inherit the priority of the high priority task.

- So, the intermediate priority tasks can not preempt lower priority task from CPU usage. Therefore, unbounded priority inversion can not occur under PCP.

How is chain blocking avoided?

- By Theorem 3.2, resource sharing among tasks under PCP is single blocking.

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 system. Subsequently, we discuss the situations in which the different resource sharing protocols would be useful.

1. Using PCP in Dynamic priority System.
2. Comparison of Resource Sharing protocols.

1. Using PCP in Dynamic priority System

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 - for 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.

2. Comparison of Resource Sharing protocols

We have three important resource sharing protocols for real-time tasks.

a. Priority Inheritance protocol (PIP)
b. Highest Locker protocol (HLP)
c. Priority Ceiling protocol (PCP)

a. 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. Under PIP tasks may suffer from chain blocking and PIP also does not prevent deadlocks.

b. Highest Locker protocol (HLP)

HLP requires only moderate support from the operating system. It solves the chain blocking and deadlock of PIP. However, HLP can make the intermediate priority tasks undergo large inheritance-related inversions and can, therefore, cause tasks to miss their deadlines.

C. 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.

HANDLING TASK DEPENDENCIES

- An assumption that was Implicit in all the scheduling Algorithms with we discussed in chapter 2 that the tasks in an Application are independent. That is, there is no constraint on the order which the different tasks can be taken up for execution. However, this is far from true in Practical situations, where one tasks May need results from Another task, or the task 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 do develop a satisfactory schedule for a set of tasks than 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 and precedence constraints are Store the sorted tasks in a linked list.

2. Repeat

Take up the task having largest deadline and not yet scheduling (i.e., scan the task list of step 1 from left). Schedule it as late as possible.

Untill 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 tasks with dependencies.

Examples 3.4.

Determine a feasible scheduling 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: 2, 5, 6, 10, 7 and the corresponding deadlines are 40, 8, 25, 20, 50, 40 respectively.

Solution.-

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

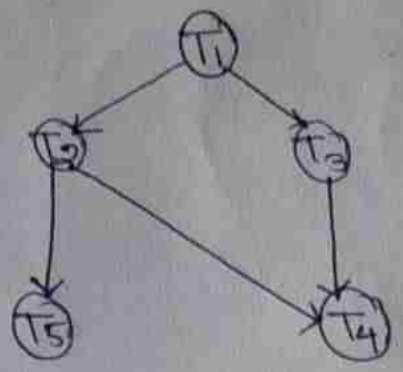


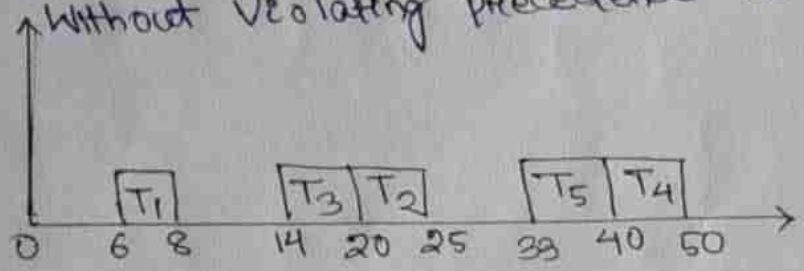
Figure 3.11:-

Precedence Relationship Among Tasks for Examples 3.4

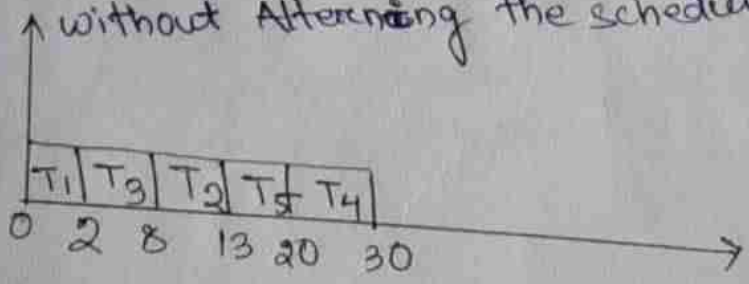
Step 1: Arrangement of Tasks in Ascending Order.

$T_1\ T_3\ T_2\ T_5\ T_4$

Step 2: Schedule Tasks as Late as possible without violating Precedence Constraints



Step 3: Move Tasks as Early as possible without Altering the schedule



EDF and RMA - based schedulers:- Precedence constraint Among tasks can be handled 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 task waiting to be enabled (on account of its predecessors completing their executions) After every tasks completes.

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

SUMMARY

- In the Many real-life Applications, realtime tasks are required to share non-preemptable resources among themselves. If traditional resources sharing technique such as semaphore 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 operatinting system. This protocol for sharing critical resources therefore, supported by almost every real time ~~system~~ operating systems. The Priority Inheritance Mechanism prevents unbounded Priority inversions. However it becomes the programmers responsibility to over-come the deadlock and chain blocking problems through careffull programming.