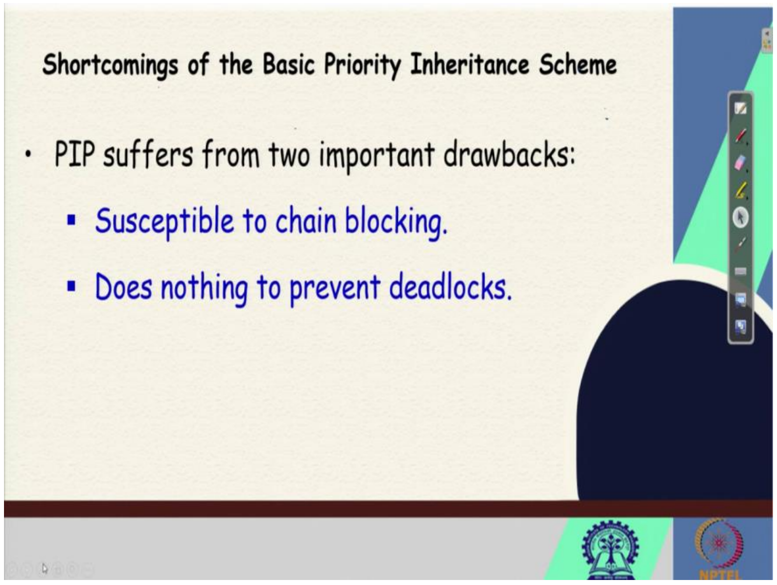
Real Time System
Professor Rajib Mall
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture: 28
Highest Locker Protocol (HLP)

Welcome to this lecture, we have so far in the last few classes examining how resource sharing among real time tasks can be supported. The traditional operating system solution of semaphores can cause tasks to miss their deadlines because of the unbounded priority inversions. So, that is a major problem that needs to be addressed when we let tasks share resources. And we were looking at the basic priority inheritance mechanism.

(Refer Slide Time: 01:02)

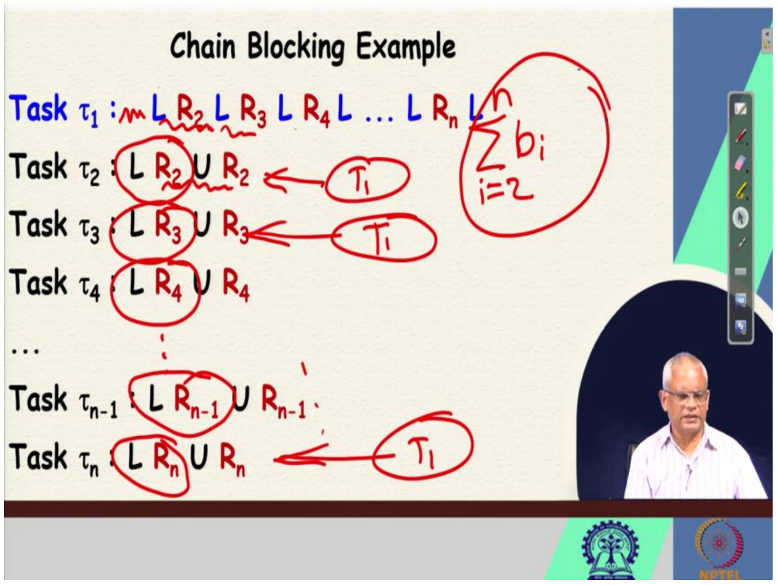


Shortcomings of the Basic Priority Inheritance Scheme

- PIP suffers from two important drawbacks:
  - Susceptible to chain blocking.
  - Does nothing to prevent deadlocks.

Now, let us proceed from there we had discussed the basic priority inheritance mechanism and we found that the solution is simple, it overcomes unbounded priority inversion, if we use the priority inheritance scheme, then unbounded priority inversion cannot occur, but there can be two other problems one is called as chain blocking and the other is that deadlocks can occur, we had looked at these two problems.

(Refer Slide Time: 01:46)



Chain Blocking Example

Task τ_1 : L R_2 L R_3 L R_4 L … L R_n L

Task τ_2 : L R_2 U R_2 $\leftarrow T_1$

Task τ_3 : L R_3 U R_3 $\leftarrow T_1$

Task τ_4 : L R_4 U R_4

…

Task τ_{n-1} : L R_{n-1} U R_{n-1}

Task τ_n : L R_n U R_n $\leftarrow T_1$

$\sum_{i=2}^{n} b_i$

But just to refresh the chain blocking problem let us say we have several tasks which need resources and let us say the task t_2, t_3, t_4 up to t_{n-1}. So, all these tasks are lower priority tasks compared to t_1 but these tasks are released earlier and they lock they execute and then lock the required resources.

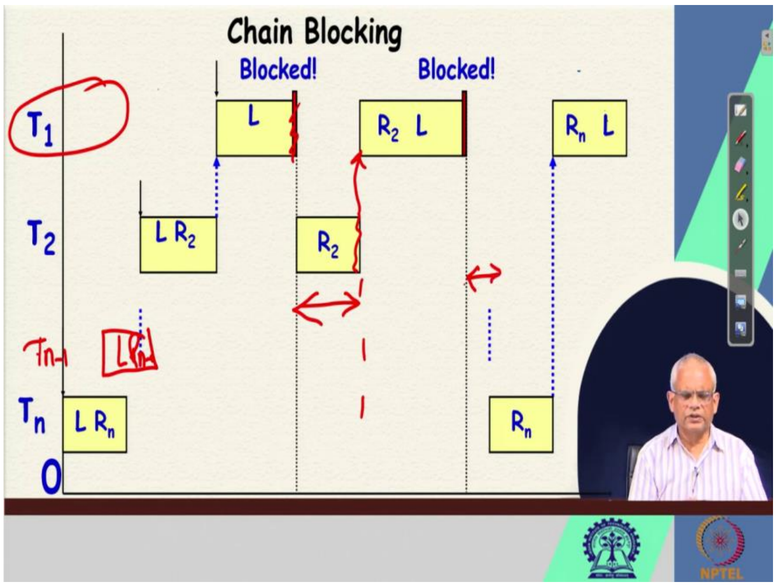
Let us say task t_2 locks R_2, t_3 locks R_3, t_4 locks R_4 and so on. And task n-1 locks resource n - 1. But at that point of time the task t_1 is released and task n also releases, it locks resource n. And at that time the task t_1 becomes ready. After all these tasks have acquired their resources and it executes some time and then it tries to lock the resource R_2.

But resource R_2 is already locked by task t_2. So, task t_1 needs to wait and as it waits task t_2 inherits the priority of t_1 because t_1 is waiting here and the t_2's priority becomes equal to t_1, t_2 inherits the priority of t_1 and it proceeds with execution and after some time releases the resource R_2 and then t_1 acquires R_2 and proceeds with execution but after some time it needs R_3. But R_3 is already locked by t_3.

So, again it blocks and again t_3 completes and so on. So, each time the task t_1 the high priority task needs a resource it undergoes blocking. So, if it needs n resources and each resource let us say the blocking time is let us say b_i so the total blocking time for the task t_1 or the total priority inversion time. For task t_1 will be $\Sigma_{i = 2..n}$.

So, this is the total blocking time or the total priority inversion time for the task t_1 and for n sources this can be substantial and the tasks can easily, the task t_1 can miss its deadline. So, there is a major problem with the simple priority inheritance scheme.

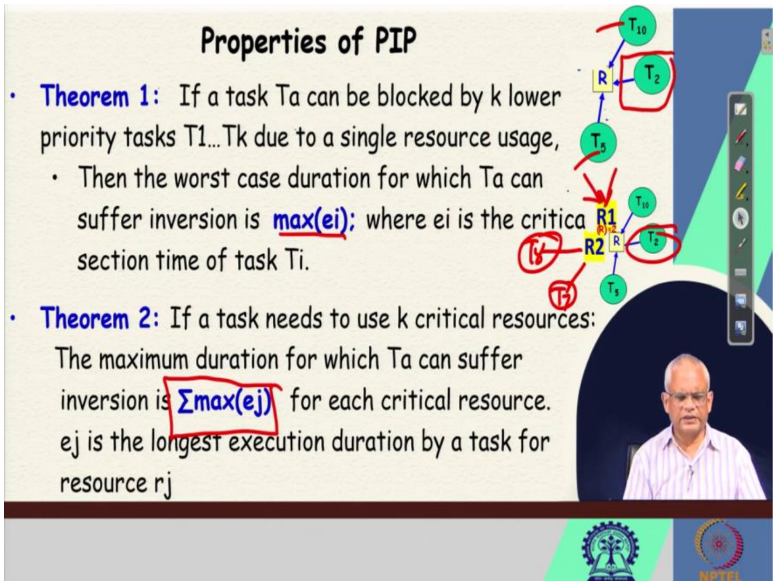
(Refer Slide Time: 05:29)



So, this is the illustration of that scheme. T_1 is the highest priority task, T_n executes first the lowest priority task locks R_n and by that time T_{n-1} is a higher priority task that gets ready preempt T_n locks R_{n-1} that is preempted by T_{n-2} and finally, T_2 locks R_2 and at that time the task T_1 becomes ready and as soon as task T_1 gets ready these are the lower priority tasks they get preempted the T_2 is the lower priority task gets preempted at that point.

But T_1 needs resource R_2 and it blocks here. And T_2 inherits the priority of T_1, it after sometime releases R_2 and as it releases R_2, T_1 starts executing, it acquires R_2 and starts executing and after some time it needs R_3 and so on. So, each time it blocks. The total block time in the worst case can be the total resource usage time of the lower priority tasks.

(Refer Slide Time: 07:18)



Properties of PIP

- **Theorem 1:** If a task Ta can be blocked by k lower priority tasks T1…Tk due to a single resource usage,
  - Then the worst case duration for which Ta can suffer inversion is max(ei); where ei is the critica[l] section time of task Ti.
- **Theorem 2:** If a task needs to use k critical resources: The maximum duration for which Ta can suffer inversion is Σmax(ej) for each critical resource. ej is the longest execution duration by a task for resource rj

So, let us look at two properties of the priority inheritance protocol. The first property gives it in the form of a theorem because you can prove it. A task T_a can be blocked by k lower priority tasks due to a single resource uses then the worst case duration for each T_a can suffer inversion is max(ei), where ei is the critical section time of T_i.

So let us say T_2 is a high priority task. And there are several low priority tasks T_{10}, T_5,etc. and all of them need the resource R. Now, what is the maximum duration for which T_2 can undergo blocking because we need to design the system conservatively. We need to know what is the worst case blocking time of T_2; on account of priority inversion?

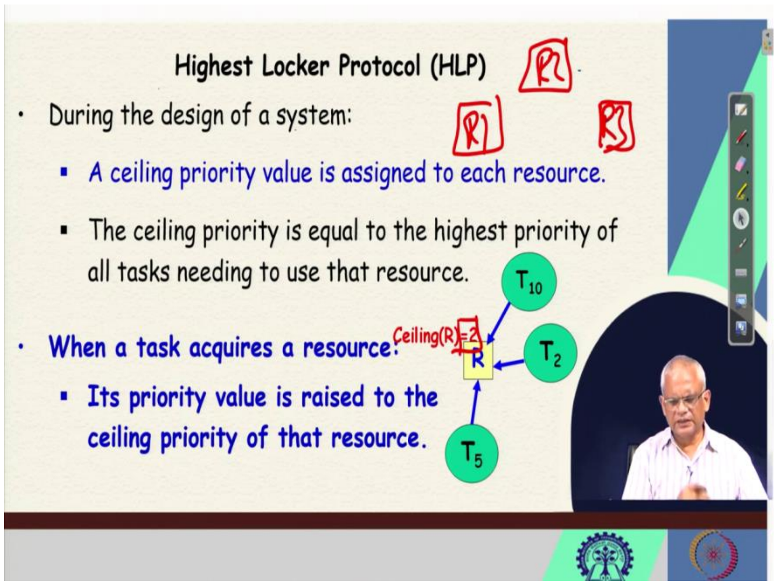
Since T_2 can be blocked by any of this at different times. So, we need to consider for instance of T_2. It can be blocked by any one of the other tasks. So, that is an important observation that T_2 can be blocked an instance of T_2 can be blocked by an instance of either T_5, T_{10}, etc. Those who need the resource R, Why is that?

The reason is that by the time T_2 gets ready and needs the resource R it might have been locked by any of these lower priority tasks, and then T_2 undergoes inversion. So, since it can be blocked by any of this, so the worst case, it will be max(ei) where ei is the critical section time of T_i. The second theorem is that if a task needs k resources, so let us say T_2 needs R_1, R_2, R_3 et c. Now the maximum duration for which the task T_a can suffer blocking is Σ max(ej) where ej is the longest duration of a task by resource r_j.

Now, to be able to prove this, we need to consider that in priority inversion protocol, the high priority task can undergo chain blocking. So, for R1 it can block, for R2 it can block, and R3 also it can block. Now, by the theorem 1, for each it will be the max of the low priority tasks needing that resource max of the critical section time.

And therefore, we need to sum it. So, this becomes max ej. So, ej for R will be let us say two tasks are there, so the maximum critical section time of T_5, T_{10}. And let us say T_7 and T_8 using R2 and also T_2. So, then it will be a max of critical section time of T_5 and T_8 and similarly for R1. So, that is the essence of the theorem, which is proved by using theorem 1.

(Refer Slide Time: 11:22)



Highest Locker Protocol (HLP)

- During the design of a system:
  - A ceiling priority value is assigned to each resource.
  - The ceiling priority is equal to the highest priority of all tasks needing to use that resource.
- When a task acquires a resource:
  - Its priority value is raised to the ceiling priority of that resource.

R2
R1
R3
T10
Ceiling(R)=2
R
T2
T5

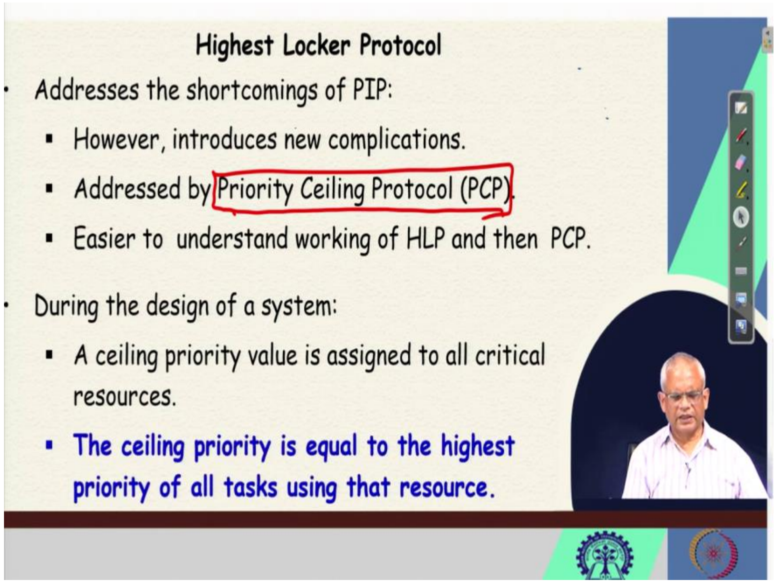
Now, the basic priority inheritance scheme has two major set comings one is the chain blocking and the other is that it does not do anything to prevent deadlocks. Deadlocks can occur in that system, it does not help in preventing the deadlocks. Now let us look at the improvement. The basic priority in priority inheritance protocol, which is the highest locker protocol; it is an improvement over the basic inheritance scheme.

And in this scheme, every resource is assigned a priority value during the system initialization. So, if there are resources R1, R2, R3 etc. So, each resource ceiling priority is computed for each of the resource during the system initialization time and these are maintained. So, what is the rule for computing the ceiling priority value for each of the resources? The ceiling priority value is equal to the highest priority of all tasks needing that resource.

We will just explain this how the ceiling priority is computed with the help of some examples. Let us look at this example. Let us say T_2 is the highest priority and also T_5 and T_{10} need resource R, in addition to T_2, T_5 and T_{10} need the resource R. So, the highest priority task needing the resource R is T_2. So the ceiling value of the resource is two.

Now, once the ceiling priority for every resource is computed, the rule is that whenever any task acquires resource, its priority is raised to that of the ceiling priority of the resource. So, if T_5 acquires resource R its priority value becomes 2, T_{10} acquires resource R then its priority value becomes 2. So, its priority becomes equal to the highest priority task needing or that can use that resource. So, that is the protocol. And of course, as soon as it releases the resource, it gets back it is own priority.

(Refer Slide Time: 14:23)



Highest Locker Protocol

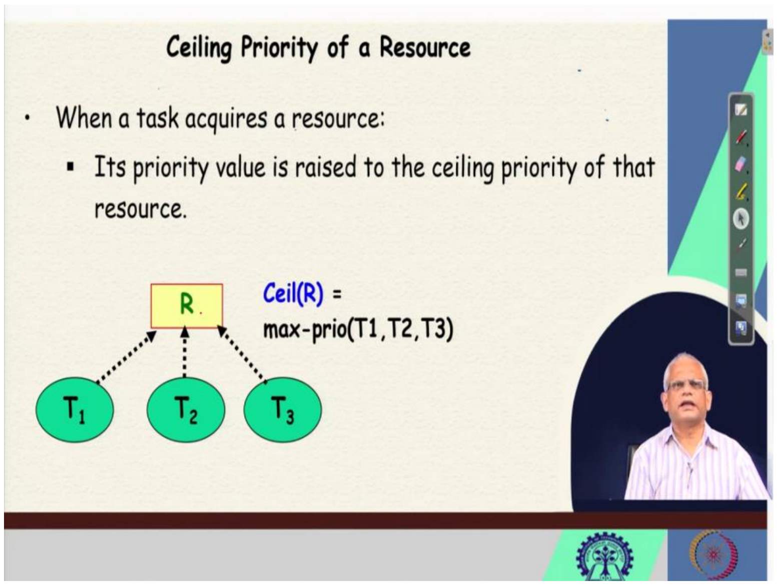
- Addresses the shortcomings of PIP:
  - However, introduces new complications.
  - Addressed by Priority Ceiling Protocol (PCP).
  - Easier to understand working of HLP and then PCP.
- During the design of a system:
  - A ceiling priority value is assigned to all critical resources.
  - **The ceiling priority is equal to the highest priority of all tasks using that resource.**

So, the priority of a task becomes the ceiling priority of the resource it acquires and as soon as it completes the resource usage and releases gets back to it is original priority. There is a simple protocol, the highest locker protocol. It will see that it addresses the problems with the basic priority inheritance protocol. But unfortunately, it creates some new complications.

And we will subsequently just after this, we will look at the priority ceiling protocol which overcomes most of the problems that have been introduced by highest locker protocol. So, the priority ceiling protocol is a bit complicated protocol, but is possibly the best protocol compared to the basic inheritance protocol and the highest locker protocol. But we will discuss the priority ceiling protocol a little later after we discuss highest locker protocol, because it is an extension of the highest locker protocol.

And if we understand the highest locker protocol, we can easily understand the priority ceiling protocol. Now in the highest locker protocol that ceiling priority value is assigned to all critical resources during system initialization. And the ceiling priority is equal to the highest priority of all tasks that might use that resource.

(Refer Slide Time: 16:15)



Ceiling Priority of a Resource

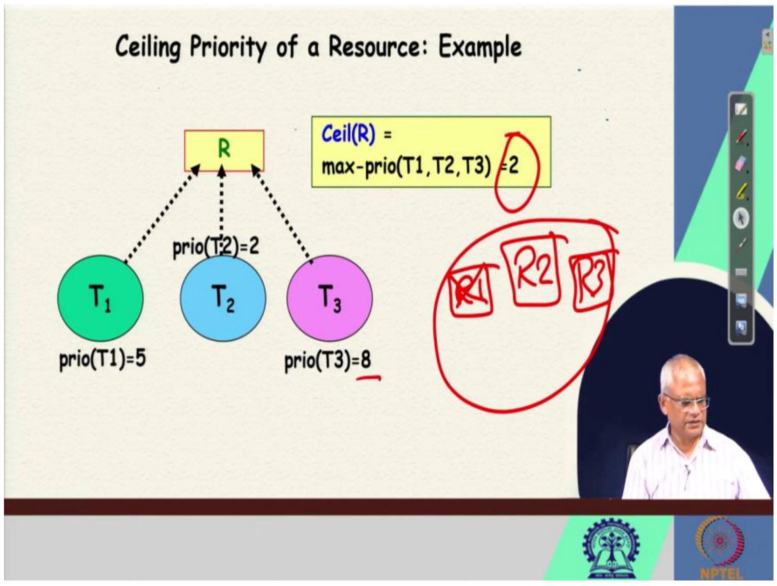
- When a task acquires a resource:
  - Its priority value is raised to the ceiling priority of that resource.

R
Ceil(R) =
max-prio(T1,T2,T3)
T_1
T_2
T_3

Just to give an example, how the ceiling priority of a resource computed let us consider the resource R which can be used by T1, T2, T3 at different points in time, sometime or other T1, T2, T3 would use the resource R. And the ceiling value that will be associated with the resource is maximum priority of T1, T2, T3.

So, if T1 happens to be the highest priority, then the resource will have the priority of T1. So, any other task that occurs the resource at some point during execution, its priority becomes equal to the ceiling priority of the resource. And as it completes the usage of resource and releases the resource, it gets back to its original priority that is the highest locker protocol.

(Refer Slide Time: 17:23)

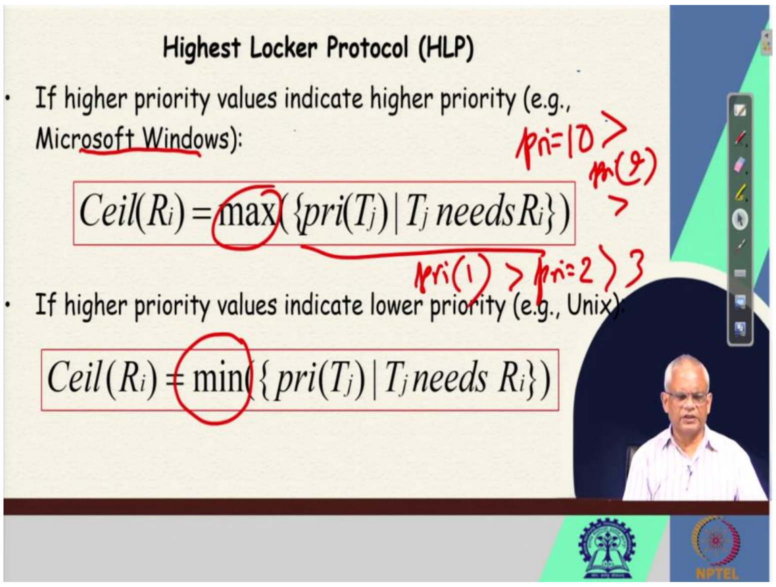


Ceiling Priority of a Resource: Example

Now here, let us say the priority of T_1 is 5, priority of T_2 is 2, and priority of T_3 is 8. Now, let us say 2 is the highest priority among this, then during system initialization, the ceiling value associated with R will be equal to 2. So, that is the important step in the highest locker protocol that the ceiling value needs to be computed for every resource. And these values are associated with that resource and whenever any task acquires, the resource, its priority becomes equal to the ceiling priority of the resource.

Now, what if the task acquires multiple resources? Let us say R1, R2 and R3 let us say task acquires all the three resources, then the priority of the task will be the maximum of the ceiling value of R1, R2, R3. Now let us say the task releases R1 and only holds R2, R3, then the priority of tasks will become maximum of the ceiling value of R2, R3. So, that is just an explanation of how the protocol will work.

(Refer Slide Time: 18:50)



Highest Locker Protocol (HLP)

- If higher priority values indicate higher priority (e.g., Microsoft Windows):

$$\boxed{Ceil(R_i) = \max(\{pri(T_j) \mid T_j\ needs\ R_i\})}$$

pri=10 > pri(9) >

pri(1) > pri=2 > 3

- If higher priority values indicate lower priority (e.g., Unix):

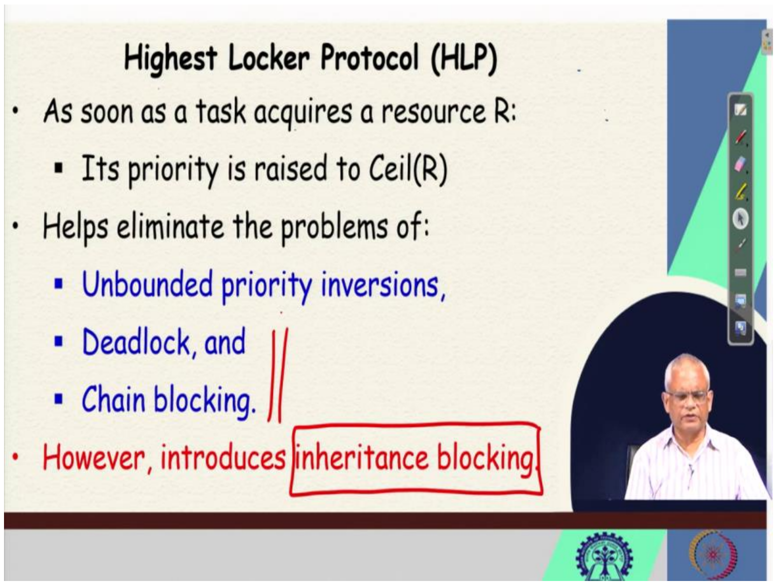
$$\boxed{Ceil(R_i) = \min(\{pri(T_j) \mid T_j\ needs\ R_i\})}$$

So, the in some systems like those which are based on the Microsoft, so their higher priority value indicates a higher priority. For example, the priority is 10. This is higher priority than priority is 9 which is higher than 8 and so on. So, 10 is the highest priority if there are 10 priority levels. And therefore, in Windows, we can say that the ceiling value of the resource is the maximum of the priority values because the higher priority value indicates a higher priority.

On the other hand, in the UNIX based systems, a lower priority value indicates a higher priority. For example, if the tasks priority is 1 indicates that it is the highest priority it is greater than a task whose priority is 2 which is greater than a task whose priority is 3 and so on. And in this case, the ceiling priority will be computed as the minimum of the priority of all tasks needing to use that resource.

So, this is a difference between the Windows based system and the UNIX based system. And therefore, we need to specify which type of system we are using, whether it is a Windows based system or a UNIX based system. And based on that the ceiling priority is computed either as the maximum of the priority or the minimum of the priority.

(Refer Slide Time: 20:55)



Highest Locker Protocol (HLP)

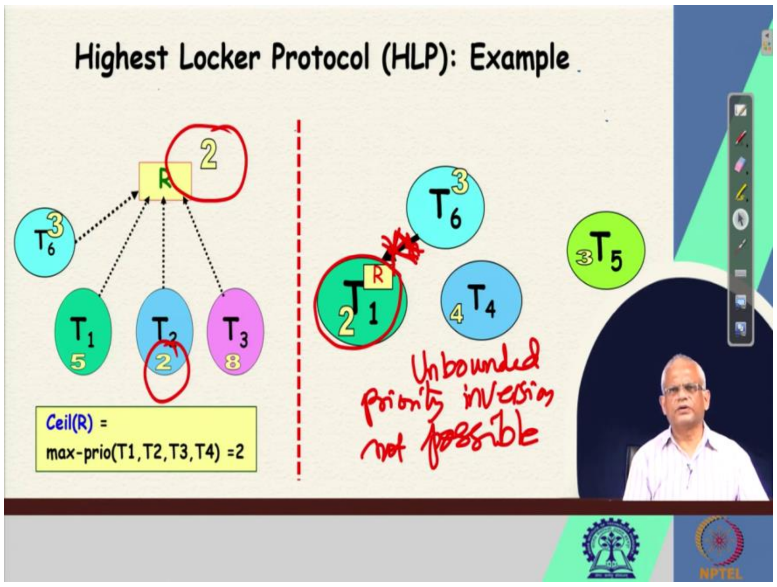
- As soon as a task acquires a resource R:
  - Its priority is raised to Ceil(R)
- Helps eliminate the problems of:
  - Unbounded priority inversions,
  - Deadlock, and
  - Chain blocking.
- However, introduces inheritance blocking

Now, this protocol, which we will discuss very simple protocol, that the task priorities raised to the ceiling priority as soon as it acquires the resource. And as soon as it releases the resource, it gets back its original priority. Now, we will see that it eliminates the unbounded priority inversion problem, which is the major problem, which even the basic inheritance scheme overcome.

But additionally, the highest locker protocol also eliminates the problem of deadlock and chain blocking. So, these two problems which were there in the basic priority inheritance protocol, they are overcome in the highest locker protocol, we will just argue about that that how the unbounded priority inversions deadlock chain blocking are overcome in the highest locker protocol. But it introduces a new problem, which is the inheritance blocking problem.

We will see what is this new problem and we will see that how it solves these three problems of unbounded priority inversions, deadlock and chain blocking.

(Refer Slide Time: 22:23)



Highest Locker Protocol (HLP): Example

Now, let us first understand the working of the protocol to start with during the system initialization for every resource which tasks need that resource is analyzed. So, this can be done statically during compile time, and the value will be assigned to the resource, statically during compile time or during system initialization time.

So, here by examining the code, a static analysis of the code will indicate the resource is being used by which tasks. Now R is being used by T_6, T_1, T_2 and T_3 and their priority values are 3, 5, 2 and 8. Now, let us assume that it is a UNIX based system where a lower priority value indicates a higher priority. So, among 3, 5, 2, 8, 2 is the highest priority and therefore, the ceiling value associated with R will be 2.

Now let us say T_1 whose priority is 5 acquired the resource R. And as soon as it acquires, the resource R its priority will change to 2. Now, let us say T_6 needs the resource T_6 is waiting for the resource whose priority value is 3 which is a higher priority task and T_4 whose priority is 4 and T_5 is three and so on, which will not be able to execute.

So, as soon as T_1 executes with priority 2 T_6, T_5, T_4, etc., cannot execute because high priority task is executing. And it cannot be preempted by other tasks T_6, T_4, T_5 etc. And therefore unbounded priority inversion cannot occur let me just repeat that as soon as the task acquires a resource, its priority becomes equal to the highest priority and therefore, other tasks cannot even block for this.

So, this cannot occur here T_6 will not get ready to run it cannot block for R, because, the priority of T_1 is already raised to 2 and this cannot even block for the resource forget about unbounded priority inversion that unbounded priority inversion was occurring when this was waiting for this and the tasks which are lower priority than this and higher priority than this, they used to execute and preempt the task holding the resource.

So, that question does not arise here, because its priority is already set to very high value. And therefore, even the other task cannot block for this first of all, and tasks that used to preempt this task which is lower than the task blocking all those questions do not arise here. So, unbounded priority inversion is not possible inversion is not possible.

Because as soon as it gets the resource its priority increases to the priority of the highest task needing the resource the other tasks will not even become ready they cannot even preempt this task to be executing and wanting to use the resource. So, that situation cannot arise and forget about the other tasks preempting these tasks so, unbounded priority inversion cannot occur in the highest locker protocol.

Similarly, we will see that the problems of chain blocking and deadlock cannot occur here. We are at the end of this lecture and we will argue that the other two problems also cannot occur. And we will see the new problem that occurs here is the inheritance related inversion a big problem which is not there in the basic priority inheritance scheme, or it was minimal there. Now, that problem is severe here in the highest locker protocol, we will examine what is that new problem that occurs. But we are at the end of the lecture and we will discuss those issues in the next lecture. Thank you.