Real-Time Systems

Prof. Dr. Rajib Mall

Department of Computer Science and Engineering

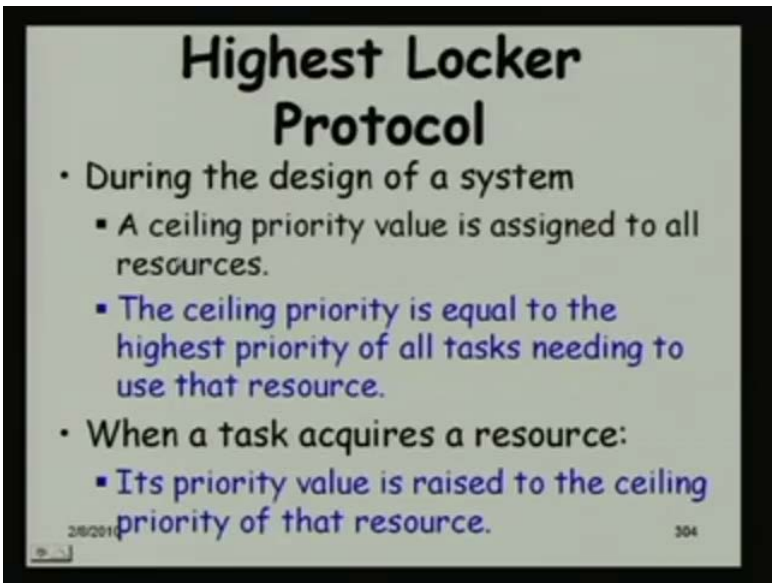
Indian Institute of Technology, Kharagpur

Lecture No. # 14

Highest Locker and Priority Ceiling Protocols

Good morning, let us get stated from where we left last time. So, we were discussing about resource sharing among tasks - real time tasks and seen, we examined what kind of problems that might arise. We saw the basic problem of priority inversion, and then, the unbounded priority inversion. And we had seen the simple protocol, the priority inheritance mechanisms, the basic mechanism for supporting resource sharing among real time tasks. So, let us proceed from that point onwards. So, today we will look at highest locker protocol and priority ceiling protocols.

(Refer Slide Time: 01:04)

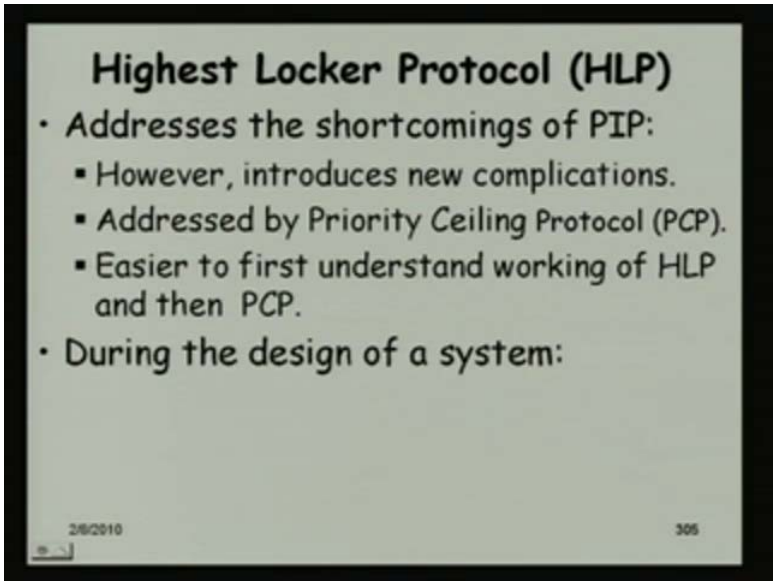


Let us first, look at the basic idea behind the highest locker protocol. So here, the resources have a ceiling priority value assign to them; and the ceiling priority value of a resource is the highest priority of all tasks needing to use that resource.

So, you know, before and what are the tasks that would be using a resource, and then set the ceiling priority of that resource equal to the highest of the task priorities that are

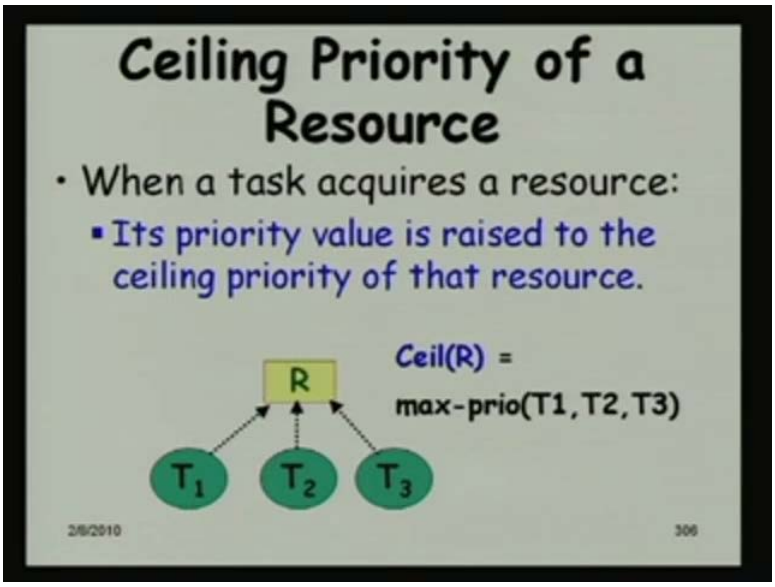
going to use the resource. And then, whenever a task acquires a resource, its priority value is automatically raised to the ceiling priority of that resource. If it acquires multiple resources then it will be naturally the highest of all these priorities. That that the basic mechanism. And we will see that - it addresses the soft comings of the basic inheritance scheme. We had identified the basic problems of the inheritance scheme. Does anybody remember what are the basic? So, there are possibility of dead lock and chain blocking. Yes.

(Refer Slide Time: 02:03)



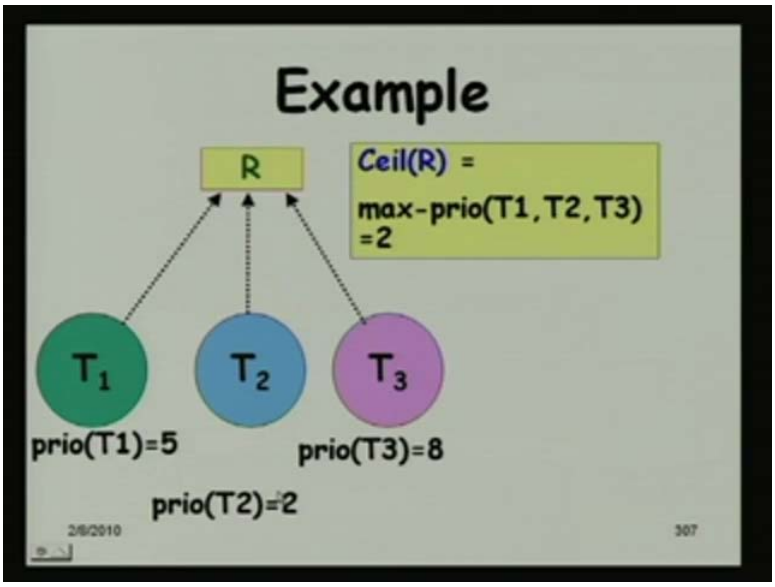
So, let us see how it will overcome those problems and unfortunately, the highest locker protocol introduces new problems. And therefore, highest locker protocol by itself is not used. But it gives us the basic understanding based on which we can develop the ceiling protocol - the priority ceiling protocol which is the one which is actually used. So, if we understand the highest locker protocol, we will understand what are the issues that the priority ceiling protocol is required to address. And therefore, why certain clauses etcetera are added there. So, let us look at that.

(Refer Slide Time: 03:27)



So, during the design of the system, the ceiling priority is assigned to all critical resources and the priority is equal to the highest priority of all tasks that might use the resource. And when the task acquires the resource, its priority raised to the ceiling priority. So, this is the situation where resource R is being used by tasks T 1, T 2, T 3. So, R will be associated with a ceiling R, which is the maximum priority of T 1, T 2, T3.

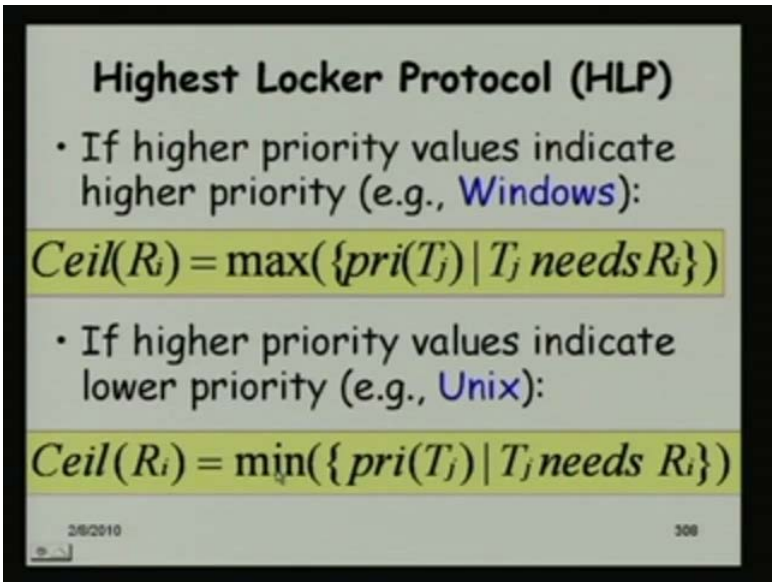
(Refer Slide Time: 03:50)



Just an example: if we have priority of T 1 is 5, and the priority of T 3 is 8, and priority of T 2 is 2. And assume that - the lower the priority value, the higher is the priority let us

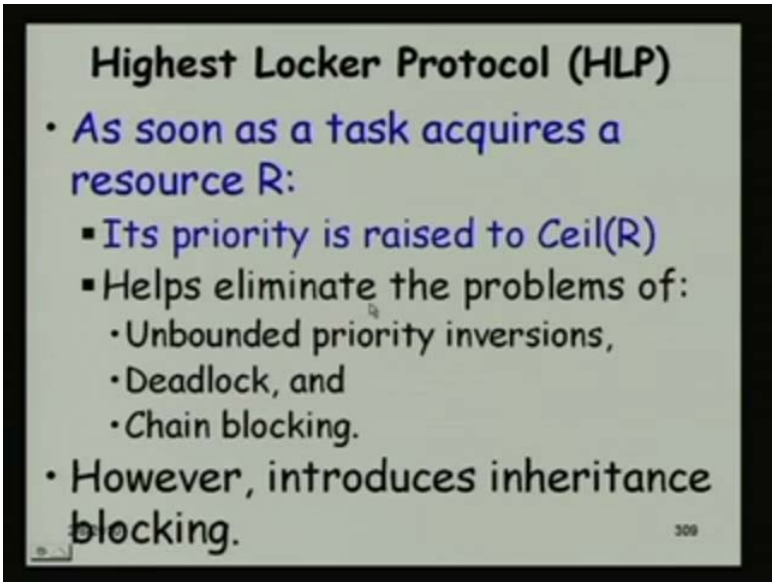
assume that. And then, the maximum priority will be 2 and ceiling value associated with R will be 2. So, whenever a task acquires R, if T 3 acquires R, its priority will become 2.

(Refer Slide Time: 04:33)



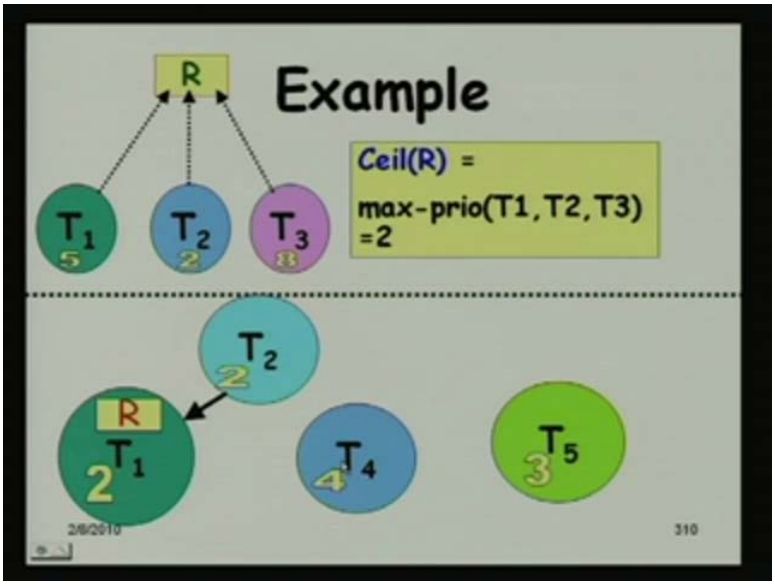
But one thing we need to remember that for windows, the higher priority values indicate higher priorities; whereas, in UNIX it is just the other way round, the lower priority values like 2 is higher priority than 8, here 8 will be higher priority than 2. So, let us just remember that - what kind of because conventions are different for different operating systems. So, we need to appropriately select the priority value.

(Refer Slide Time: 05:05)



Now, as soon as the task acquires the resource, its priority is raised to the ceiling value, and we will see that the simple mechanism will help eliminate the unbounded priority inversions, the dead lock and the chain blocking. But it would introduce a new problem which will call as the inheritance blocking.

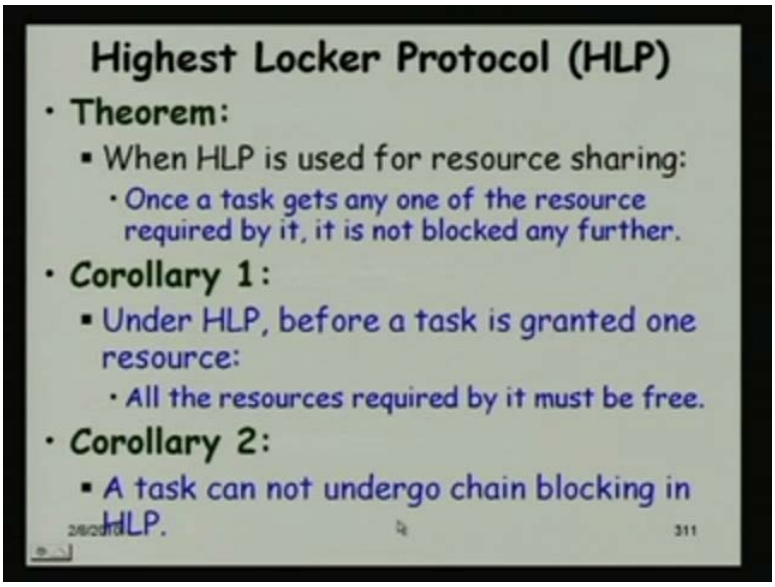
(Refer Slide Time: 05:28)



So, this is an example of the working of the algorithm. T 1, T 2, T 3 are sharing the resource R, the ceiling value associated with R is 2, because these have priorities 5 to 8 and the maximum of this is 2. Now, let us say T 1 acquires R, T 1's own priority is 5 and as soon as it requires R, its priority would become 2. And if T 2 gets ready by that time, and then it would have to wait.

But the thing is that the intermediate priority tasks, so, that is the inversion occurring no doubt here. Let us see here, that the task 5, its priority has become 2 it is executing, whereas, T2 whose priority is 2 is waiting, but the thing is that unbounded priority inversions are avoided. So, T 4 does not need any resource cannot preempt T 1 from CPU uses, because T 1's priority has become more, similarly, T 5.

(Refer Slide Time: 06:56)

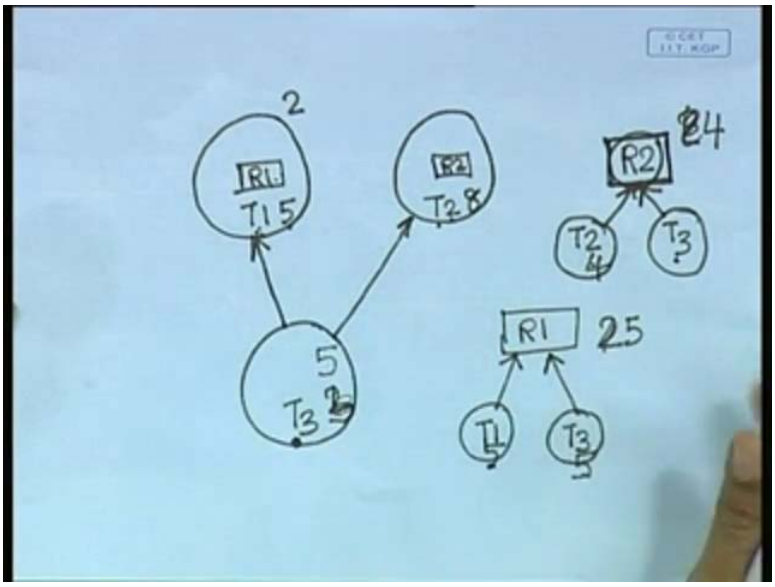


Now, let us look at a theorem. When highest locker protocol is used for resource sharing, once a task gets any one of the resource, it will not block any further. Or in other words, there is no chain blocking. At this theorem says that under highest locker protocol there is no chain blocking. So, we will just look at the outline of the proof, very simple. You can argue out that there would not be any chain blocking. And then, we look at 2 corollaries of this theorem.

Let us, once a task is granted a resource, all resources required by it must be free. So, when a task is granted one resource under HLP when none of the resources could have been held by any other task. That is a corollary. Because know, it does not block any further. From here, you can easily so that if task gets one resource, then other resources must be free at that time. And the other corollary is that cannot undergo chain blocking. So, let us see the outline of the proof for why that cannot be any chain blocking, and once it is get a resource - one resource, then it will not be blocked any further.

So, I do not have a slide for that. I will just outline the proof on the paper. So, just observe simple concept. Say let us assume… We will prove it by contradiction. So, let us assume that there are two tasks who are holding resources R1 and R2. And let us say, we have a higher priority task. So, let us say, this is T 1 T 2 and T 3. Let us assume T 3 is higher priority that way the priority inheritance sorry the priority inversion will occur.

(Refer Slide Time: 08:43)



Now, let us say T 3. So, as soon as T 1 acquires R1 its resource will be its priority will be raised to the ceiling priority menu, is it not? That is the basic protocol. So, according to this scheme, the resource R1 is required by T 1 and T 3 right. Similarly, resource R2 is required by T 2 and T 3. Sorry, need to show it, we are saying it a rectangle, let us follow that convention. So, R2 - the resource R 2 is required by T 2 and T 3, is it not? So, the ceiling priority of R1… Let us assign some arbitrary priority value just for our understanding. We can say it for any arbitrary priorities. So, let us assume that its priority let us say some 5, its priority is 8 and its priority is let us say 2. Now, the ceiling priority of R2 will be how much, maximum of 2 and 8 is it not. 2 and 8, because R2 is used by T 2 and T 3. So, the ceiling of R2 will be T2, is it not? Similarly, the ceiling of R1 will be 2; will be atleast 2, because there might be other tasks which might be using whose priority might be 1. So, it is priority is atleast 2.

Now, let us assume that T 2 first blocks for resource sorry T3 first blocks for resource R1, and then it gets the resource, and then computes and by that time T 2 is holding R2, and then it again as it requires R2 it again blocks. Let us assume this, and we will show that it is not possible. So, when T 1 acquires R1, its priority must have been set to 2 right. And if its priority were set to 2, then it is not possible that another task can acquire a resource R2, because it is a lower priority task to acquire R2, because it is already a higher priority, its priority has already increased to 2 after acquiring the resource. Is it not?

So, as soon as its priority, would have it would have acquired the resource its priority would have got raised to 2, there is no chance that T 2 could have acquired R2. Similarly, we can show that if T 2 started executing and acquired R2. R1 could not have possibly been acquired by R1 sorry R1 could not have possibly been acquired by T 1.

Sir, for example, these are two different resources.

Yes, two different resources, let us just please understand again, I am just repeating that resource R1, the ceiling value is 2 and by the highest locker protocol, as soon as any task acquires that resource, its priority automatically becomes 2 right. Now, similar is the case with R2 sorry R1. As soon as, the task acquires R1 it is priority will become 2 right. Now, we are assuming that the situation exist, where R1 is held by some task - the task T 1, its priority was 5, and R2 is being held by T 2 whose priority is 8. Right?

But let us assume that first R1 was holding sorry T 1 was holding R1. Then T 1's priority will be raised to 2, it is high priority. So, at that time it is not possible that T 2 can execute, because T2 is a lower priority, unless a higher priority task completes how will it get chance to execute. Right? It has got its resource, it is executing. So, T 2 could not have possibly executed and acquired R2. And similarly, you can argue that if T 2 had first acquired R2, T 2 started executing and first acquired R2, T 2's priority would have increased to 2. And T 1 being a priority 5 could not have executed and got the resource R1.

Now, we can show a through similar argument. So, this is a contradiction basically.

Sir, it may be a case that T 1 the ceiling priority of R1 is, say 5.

R1, see, R1 should be ceiling priority - R1 should be at least as much as T3 which is the high priority task.

Suppose it is also 5.

Yes.

Then the ceiling priority will be 5.

Suppose this is 5, is it. Right.

Then the ceiling…

Ceiling priority is 5. So, these two are equal priority you are saying.

Yes.

Right. So, this is…

T 1 and T 3.

T 1 and T 3 are 5.

And say, suppose T 2's priority is 4.

T 2's priority is 4. So, its priority will be 4, yes.

R1 started executing, T 1 started executing first.

T 1 started executing first, yes.

It has acquired R1.

Yes.

And its priority is still 5.

Yes.

But now, T 2 has come.

Yes.

T 2's priority is 4.

4 yes, T 2's priority will be made 4 yes.

Yes. So, T 2 has to execute.

T 2 will execute and complete.

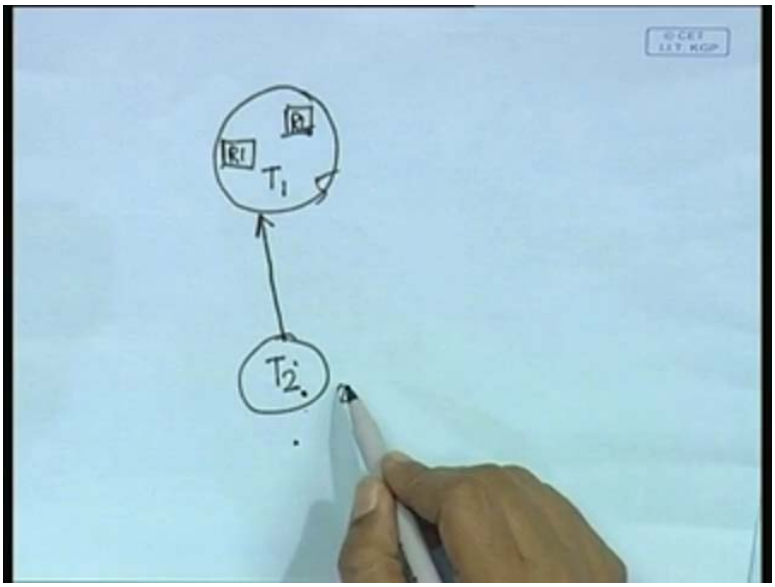
Yes sir, it will preempt T 1.

It will preempt T 1 and then, T 2 will start execute and start executing and T 2 will complete.

It would have acquired R2.

It would have acquired R2, and then it will complete. Because T 3 priority is 5, so, T 3 cannot execute right. 4 is a higher priority, T 2 T 2 has priority has become 4. So, T 2's priority, T 2 would complete before T 3 can start get a chance to execute. So, you can construct all possible scenarios and see that this holds and we can prove it actually formally by considering all possible values of priorities - priority combinations i j in terms of i j, you can prove which is worked out in the book.

(Refer Slide Time: 16:41)



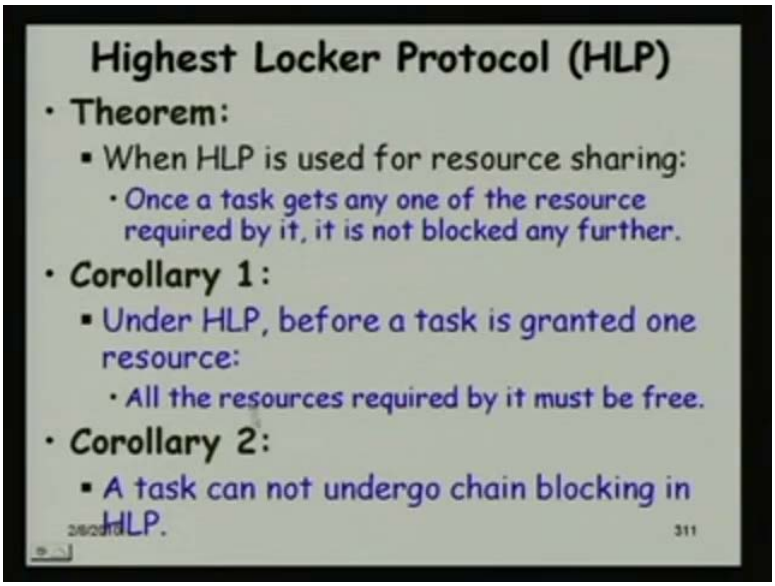
And we can also show in a through very similar argument. That if a task is holding both the resources, that is also a possibility that - a task T 1 was holding both R1, and R2 right. And then T 2 which is a higher priority task. Let us say T 2's priority is 2, and T 1's priority is let us say 5. So, T 1 was holding both R1 and R2, and T 2 was blocking on R1.

Now, let us say T 1. So, first time it has blocked for R1, and then after some time T 1 completed using R1, and then T 2 started executing and again it block for R2, because it

also needed R2. So, again it started blocking for R2, so 2 times it has blocked. But thatselves are not possibly, exactly that is not possible. Because as long as it has it is holding some resource R2 which is required by R2, any resource required by T 2 its priority will become at least as much as this. Right? So, unless it releases both these resources, it completes execution of both these resources, T 2 will not get a chance to execute. Right?

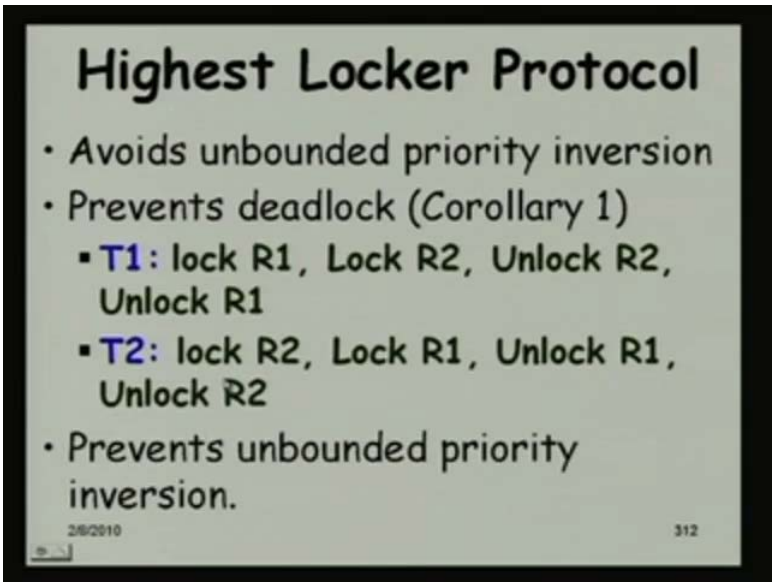
So, once it gets one of its resources, it will not block any further for any other resource. Or in other words, chain blocking is not possible under highest locker protocol. So, you can try it more examples, and then look at the formal proof. See, that the proof outline is just similar excepting that - we just generalize instead of taking instances 5, 2, etcetera we just generalize right. Or if you find the better proof - easier proof - you just report it to me. We will consider that for a bonus mark. So, let us now proceed.

(Refer Slide Time: 18:46)



So, we know that under highest locker protocol, chain blocking is not possible. And once it gets one resource, other resources must be available. So, this much we know.

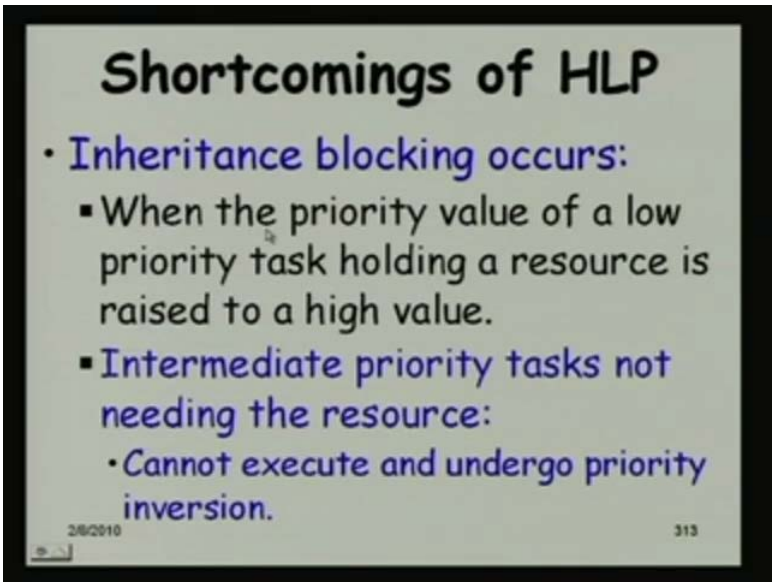
(Refer Slide Time: 19:00)



Now, we have seen that it avoids unbounded priority inversion. We can easily show that it prevents deadlock by the corollary 1. From corollary 1 where you said that see once it is gets one resource, other resources must be free right. So, based on that we can show that they are cannot be a deadlock. Because it has got one resource, all other resources are free. So, where is the chance of deadlock?

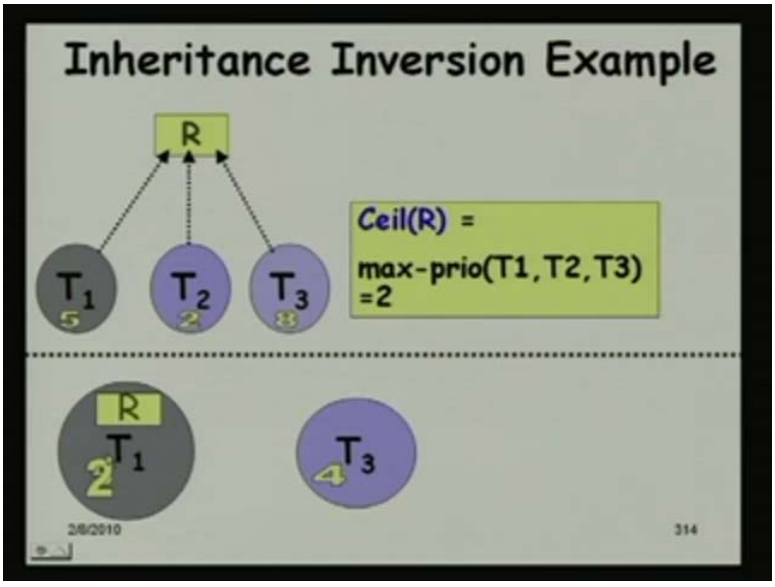
We can even show that through some examples. See, this is the example where the basic inheritance scheme was getting in to deadlock right. So, let us say T1 and T2 are two tasks requiring resources R1 and R2, and let us say T2 is a lower priority, T1 is a higher priority. And let us say T2 started executing locked R2, and then T1 started executing and issue the command lock R1, but that is not possible. Because as soon as T2 has acquired R2, the ceiling priority of R2 must be as as much as the priority of T1 and therefore, this assumption that it will lock R1 is not possible right. So, dead lock is prevented.

(Refer Slide Time: 20:28)



Now, the main problem of the highest locker protocol is the inheritance-related inversion or inheritance blocking. So, here when a low priority task gets a resource its priority is raised to high value, and then the tasks whose priority is more than this low priority task which are the intermediate priority tasks, even though they do not need any resource. They will be prevented from execution, because the low priority tasks priority has increased right. So, this is called as the inheritance blocking.

(Refer Slide Time: 21:10)



So, this is just an example of an inheritance blocking. We have these three tasks with priority 5 to 8. So, the ceiling priority of the resource is 2. Now, let us say T 1 whose priority is 5 acquired R, priority becomes 2 by the highest locker protocol, and then T 3 priority is 4 it is more than T 1 cannot execute, even though it is ready cannot execute. So, right. (Audio not clear. Refer Time: 21:48) So, we should make… Right. There is a problem here; we should have name this task as T 4 or T 5 something right, because T 3's priority is 8. That is correct. So, it should be T 4 or T 5 something whose priority is 4 cannot execute. So, it is undergoing inheritance-related inversion. Similarly T 5, its priority is 3 cannot execute, because its priority has become 2. So, these tasks are undergoing inheritance inversion until T 1 completes. So, what is the duration for which T 3 and T 5 might undergo inheritance inversion?

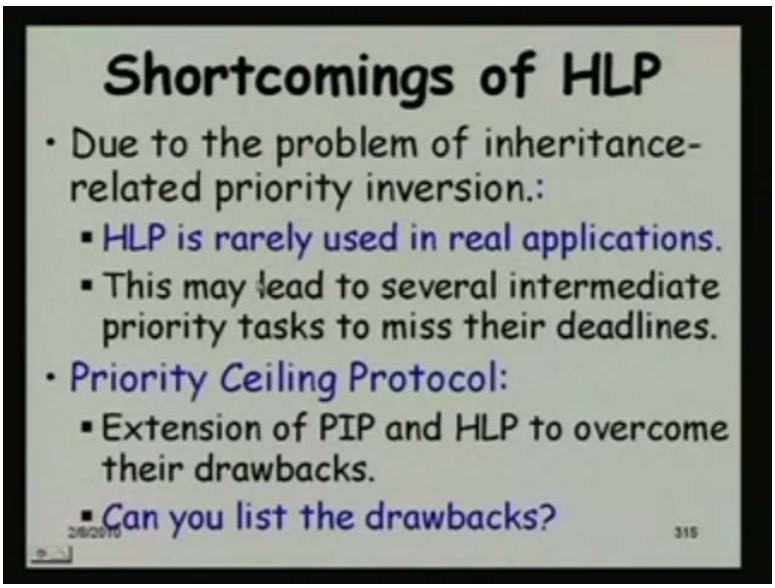
(Audio not clear. Refer Time: 22:30)

Yes. So, the duration for which T 1 needs the resource R - that is the duration for which T 3 and T 5 will undergo inheritance inversion.

Sir, actually it is T 1 blocks for input, output or something.

If it blocks for input-output, then the situation is more complex we need to modify our simple assumption. Right? The duration we need to calculate accordingly. But right now we are assuming that once it gets the resource continuous without blocking right.

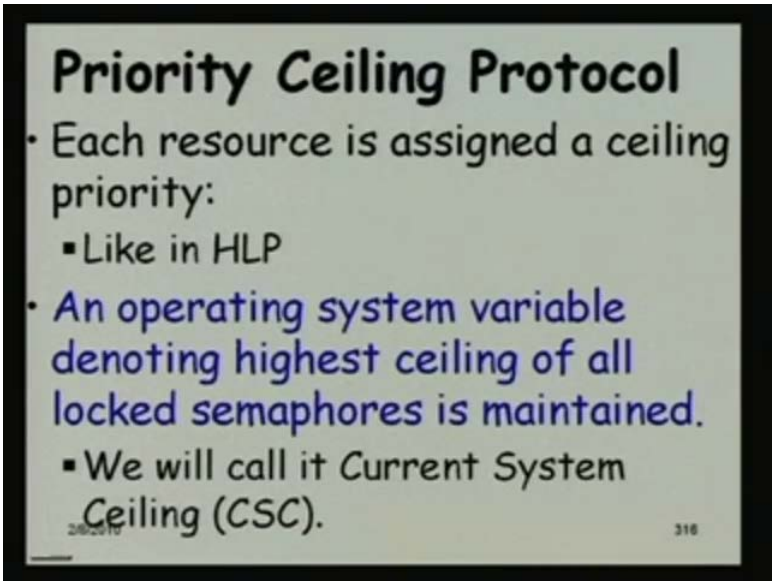
(Refer Slide Time: 23:11)



Now, because of the inheritance-related inversion is rarely used in applications. Because just imagine the lowest priority task, there is a chance that it will become the highest priority task, and then it blocks everything else. For example, a logging task which you know just keeps the logs, it can become the highest priority task, and even the most sensitive tasks will be blocked. And since, this can occur for many low priority tasks, rarely used in real applications.

So, the problems of highest locker protocol is overcome by the ceiling protocol to the priority ceiling protocol. So, it is a extension of the highest locker protocol. And we now know the draw backs of the basic inheritance scheme. Even though it avoided the unbounded inversion, and deadlock, and chain blocking. And the highest locker protocol, it overcame the deadlock and the chain blocking problem. But it had a new problem - the inheritance-related inversion. So, let us see how the priority ceiling protocol overcomes that.

(Refer Slide Time: 24:28)



Here, just like the highest locker protocol, a ceiling priority is computed for every resource. But in the highest locker protocol, whenever a resource was acquired by a task its priority was raised to the ceiling value. Here, it does not occur like that. Here, we have an operating system variable which denotes the highest ceiling of all locked semaphores.

So, if R1, R2 are two resources that are been used and they have ceiling value of C 1 and C 2. Then, the operating system variable will get maximum of C 1 and C 2. So, the operating system variable which we will call as the current system ceiling or CSC, it stores the highest of all ceiling protocols of the resources under use.

Ceiling of say semaphore, what is that mean ceiling of semaphore.

Locked semaphore, see, the terms that are used here like we sometimes we will call it as mutex - locked mutex.

See semaphore.

Called locked semaphore - locked resource. So, these are terminologies we will just keep using equivalently.

Sir, semaphore will have a value?

Yes.

Some 1, 2 and minus 1, minus 2.

Exactly.

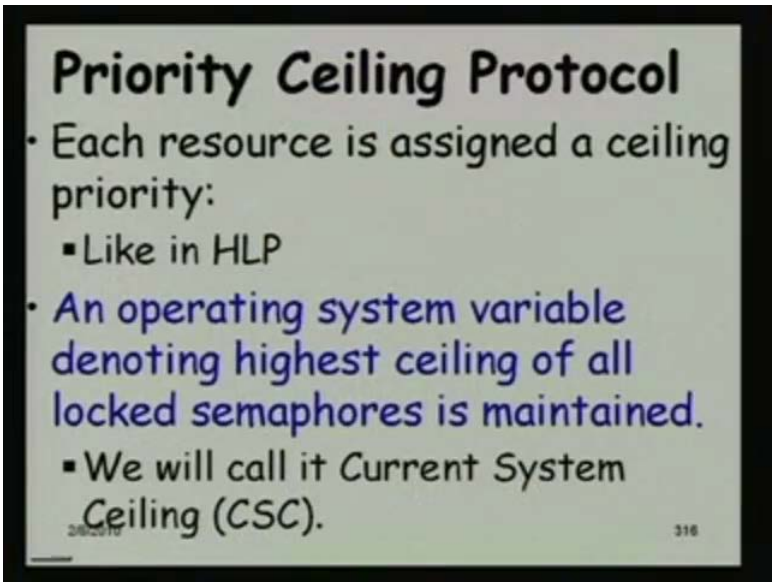
And the priority is different thing I mean the ceiling of the semaphore value or the priority value it leads to.

No, no, this is a locked semaphore, and locked semaphore is the semaphore is set right. And, we instead of saying that, the ceiling is associated with a resource we just used it as ceiling is associated with a semaphore, right? Because the semaphore…

Highest of all that priorities means, ceiling priority.

Exactly, that is all; highest of all ceiling priorities of various locked resources. So that is the terminology we are using so far. So, possibly, because we just used locked semaphore. So, may be that is the causing the confusion, but these are some of the terms that are used interchangeably right. Let us proceed.

(Refer Slide Time: 26:51)



But what is an operating system variable, I mean how do you check the operating, let us say, you have all of you have used UNIX. So, how do you check what are the current environment, the the system the set of system variables is called as the environment variables right.

It could one of the variable. In the order of path.

Path, no, no, path is just one example, path is an environment variable. It is a system variable - the current path. But how do you check all the environment variables that are there.

You have to (Audio not clear. Refer Time: 27:28 t0 27:32) all the input that are set include in a file.

Any one wants to give. See, all of you have used UNIX how do you check the environment variables.

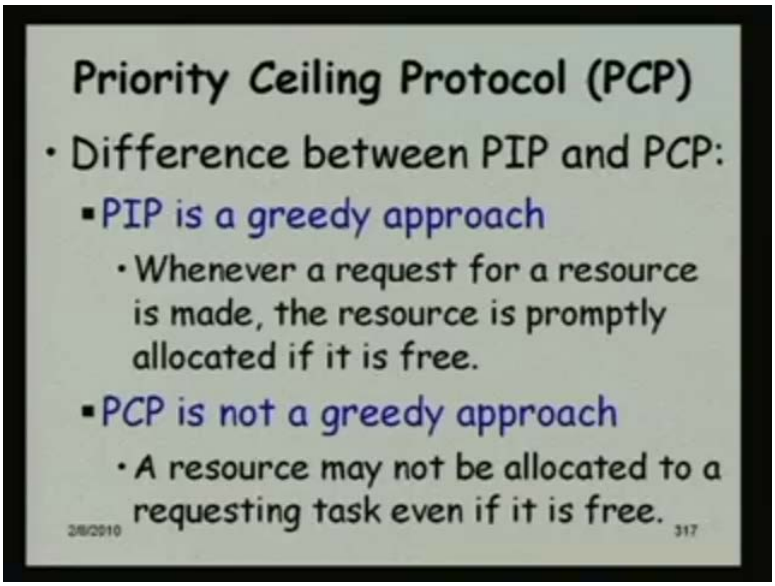
For receiving this… (Audio not clear. Refer Time: 27:44)

No, no, see, as operating system to function, see, operating system is configurable you know it the path can be changed a current path. Do not have to set the path each time right, or…

If you just (Audio not clear. Refer Time: 27:57)

Linux we have used all know, UNIX we have used. So, what is the command for UNIX – env; if you give a command env, you will see the set of environment variables just try that out. It will list at atleast two dozens of variables, just check what are the variables why they are used.

(Refer Slide Time: 28:09)

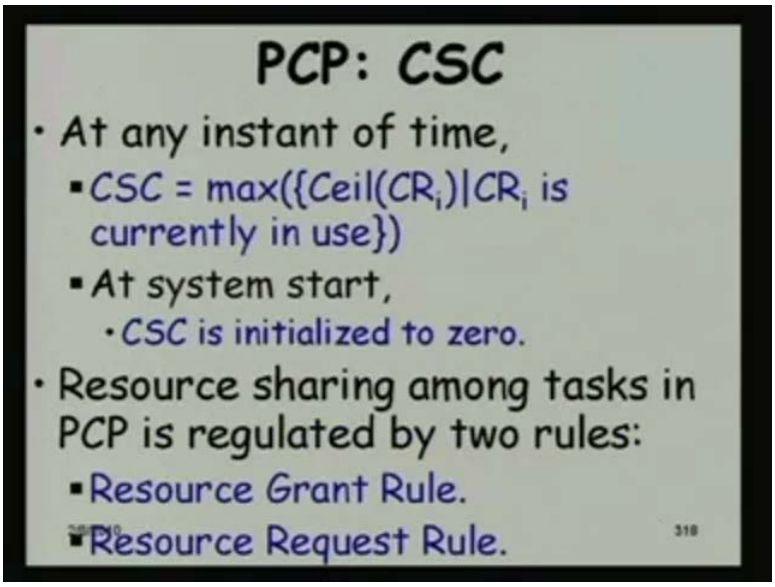


So, the priority inheritance protocol is basically a greedy approach. Because whenever a request to a resource is made as long as the resource is available, it will be allocated. See, if it is being used by some task, then it might block, and then the inheritance scheme applies and so on.

But as long as the resource is there and some task requests it, it gets it; is it not? So that is a greedy approach. But here you will see that in the priority ceiling protocol, a resource may be there - free resource, but the task will be prevented from acquiring it. So, it is not a greedy approach.

Later, I will just observe as we go through the ceiling protocol, just observe why it is not a greedy approach. And later, I will just ask you a question, just give us an example, why the ceiling protocol is not a greedy approach.

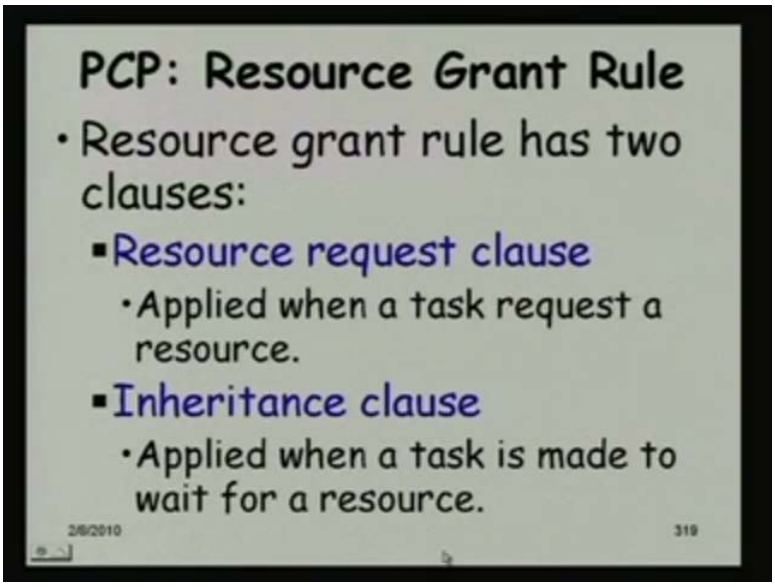
(Refer Slide Time: 29:22)



So, the current system ceiling as we were saying is set to the maximum of all ceiling values of the resources in use, we just a formal writing that - the system the current system ceiling is equal to the maximum of all ceiling values of resources that are currently in use. And to start with, the current system ceiling is initialized to zero, because no resources are being used. Assuming that zero means that nothing is being used. Or we can even set it to a very large value depends on the kind of convention we are following.

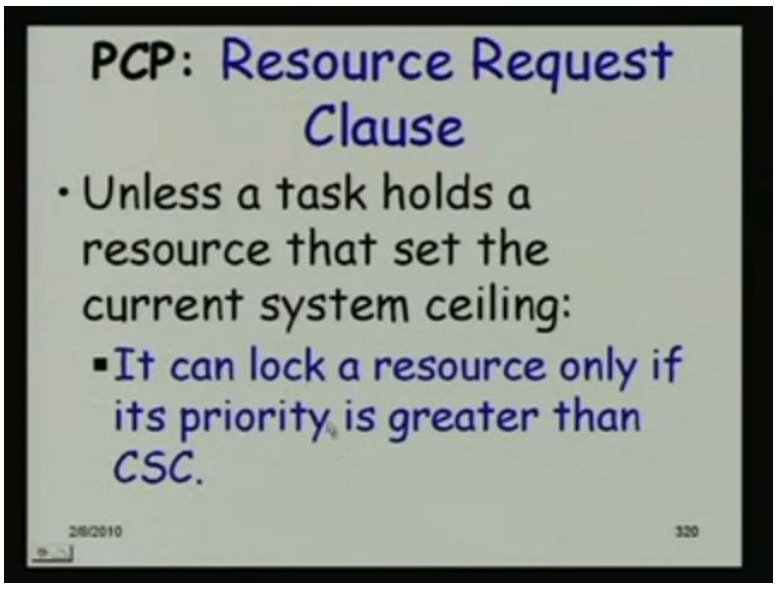
So, the priority ceiling protocol basically has two rules. One is the resource grant rule and the resource release rule actually; it should have been release mistake here, grant rule and the release rule.

(Refer Slide Time: 30:31)



So, first let us look at the grant rule. The grant rule has two clauses here. So, when a resource is to be granted first the request occurs, and then there is an inheritance clause that is applied. So, let us first concentrate on the grant rule, and then we will look at the release rule.

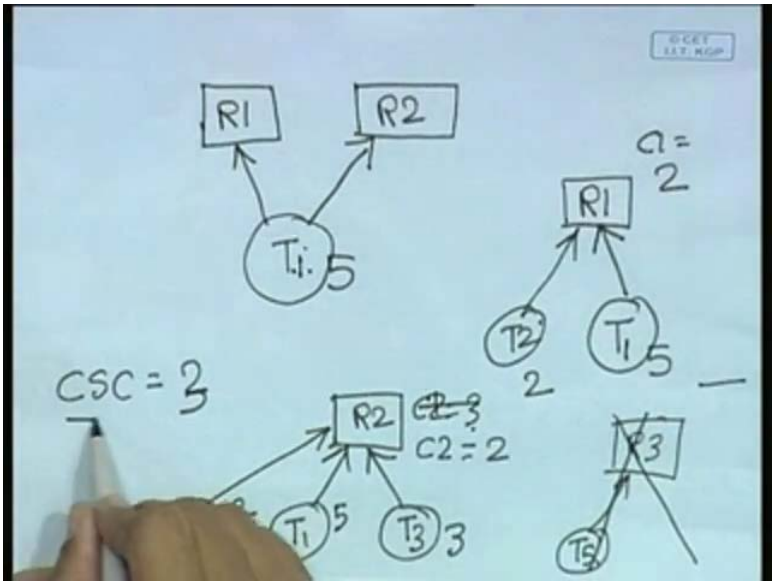
(Refer Slide Time: 31:01)



So here, unless a task holds a resource that set the current system ceiling, it cannot lock a resource, unless its priority is more than the current system ceiling. So, let us assume, I will just do not think there is example here, I will just take an example. See, this is one of

the very basic working of the ceiling protocol. So, you just read the statement once again, then I will just give you an example. Unless a task holds a resource that set the current system ceiling, it can lock a resource only if it is priority is greater than the CSC - current system ceiling.

(Refer Slide Time: 31:55)



So, let us say, we have a situation, where we have a task, which needs two resources R1 and R2. R1 and R2 are needed by T 1's. And let us say, the R1 is also used by T 2. So, T 1's priority let us say 5, and T 2's priority let us say 2. So, the ceiling priority of R1 will become 2. Is it not? Now, R2 let us say is being used by T 1 and also T 3, and T 1's priority is 5 and T 3's priority is let us say 3 or may be you can consider 8 or something. So, R2's ceiling priority. So, C1 is 2 and here C1 is equal to 3 right.

Now, let us say, T1 first acquired R2. So, the current system ceiling will be set to 2. No, sorry it it will be set to 3. See, T 1 first acquired R2, and R2 ceiling priority sorry this C2, R2 ceiling priority is 3. So, the current system ceiling will be made 3. Now, after sometime T 1 wanted to lock R1, and then it will be directly granted access to R1, the the current system ceiling will not be looked at. It will because it is it is the one which had set to the current system ceiling.

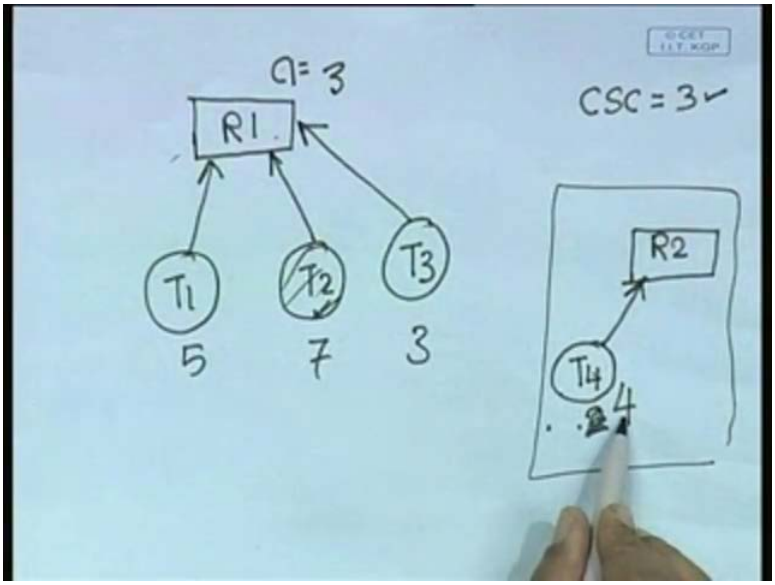
The CSC is not a consequence, but T 1. But let us say, we have the task T 2, let us say another task, let me just give R3 which is being used by T 5. So, T 5 if it request R3, its

current its priority will be checked - T 5's priority will be checked with the current system ceiling. And if it is priority is less than the current system ceiling, it will be prevented from acquiring the resource that it needs. And let us say, just to explain it little bit further. Let us say just rather than complicating with R3, let us just assume that T 2 whose priority is 2 also needs R2. I think that will. So, T 2's priority is 2. So, R2's ceiling will become 2, right maximum of all these priorities. Now, let us say T 1 first acquired R2, its priority will become 2.

Sir, this CSC will become…

Sorry, not, the CSC will become 2. The priority is do not change, the priority of T 1 do not change. Please remember that under CSC the priority does not change upon acquiring a resource, unlike the highest locker protocol. So, only the current system ceiling is set to 2. And let us say T 1 sorry T 3 wants to acquire R2, because its priority was not increased - T 1's priority was not increased by acquiring R2. Let us say T 3 is started executing, because it is priority is 3. And after sometime, it wanted to use resource R2, then T 3's priority will be checked against 2, it is higher priority than 3. So, T 3 will be prevented from locking. Is that ok or it needs to explain with another example. Anybody thinks it is slightly more complex so that we need to take one more example, yes.

(Refer Slide Time: 37:01)



Let me just take another example, just to explain this resource request clause in the grant procedure. So, let us say we have a resource R1 and the resource R1 is needed by three tasks. Let me just write T 1, T 2 and T 3. And let us give some arbitrary priority values 5, 7, 3 or you can give some values to me I can write that.

So, the ceiling value of R 1 will be how much, 3. C1 the ceiling associated with R1 will be 3. Now, as soon as any task here acquires the resource, this current system ceiling will be set to the ceiling value here. So, CSC will be made 3, let us say T2 has acquired the resource. But T 2's priority does not change; so, only the environment variable which has got changed.

Now, it is possible that another task T 4 whose priority is let us say 2 or something or let us say 4, it will start executing, because it is priority is more than T 2 right. Let us say T 2 has acquired R1 and T 2's priority has not changed, T 2 has acquired R1 and the current system ceiling is set to 3, and T 4 priority 4 more than 7, T 4 starts executing right.

Now, let us consider the example where T 4 also needs resource. Let us say R2, and T 4 requests for R2 then T 4's priority will be checked with current system ceiling, and then if T 4's priority is larger than the current system ceiling it will be granted R2. But it is not here their priorities only 4. So, it will be prevented from locking R2.

Means before the request R1 and if T 4 priority is greater than T 1 or T 2.

No, if T 4 requests R1 we have to write T 4 first and its ceiling priority will become 4 right. So, do can you just imagine or think, why this clause is required; checking against the current system ceiling. And see here, the resources idle, no one is needing that resource. But still T 4 is prevented from getting R2. When you think, why this clause is introduced in the ceiling protocol?

So, that all the resources do not get, like are available at a time given it.

Ok.

It does not get blocked.

Exactly, exactly, see here, the basic idea here, like he said, see, the basic idea is that intermediate priority tasks are not not blocked. They are allow to execute; unlike, the basic the highest locker protocol where these tasks were undergoing inheritance-related inversion. Here they do not undergo inheritance related-inversion; no priority inheritance is applied till now. It is only the current system ceiling is changed.

But the thing is that any related resource that it might need should be available. Otherwise, there will be problem of deadlock, chain blocking all those problems will come; and to overcome those problem, the current system ceiling is set. But the either possibility that R2 is never needed by T 2, but still T 4 is prevented from accessing it. So that we will call as a problem here, call it as some avoidance-related inversion later. But this problem can arise that some resource is never going to be required by anybody excepting these task and some other task - T 7.

But still, they are blocked. These are shared resources basically. So, I written T 4, T 7, it is are unrelated to this and this never needs these resource, but still they will be prevented; unless, this as a priority 1 - T 4's priority is 1 and if it is T 4's priority is 1 then we know that T 1 is never going need a resource whose priority is 1 whose ceiling priority is 1. And therefore, T 4 will be allowed to execute R2 right. So, this will prevent the deadlock, and T 4 will execute and complete. So, let us see the other clauses.

Sir,

Yes,

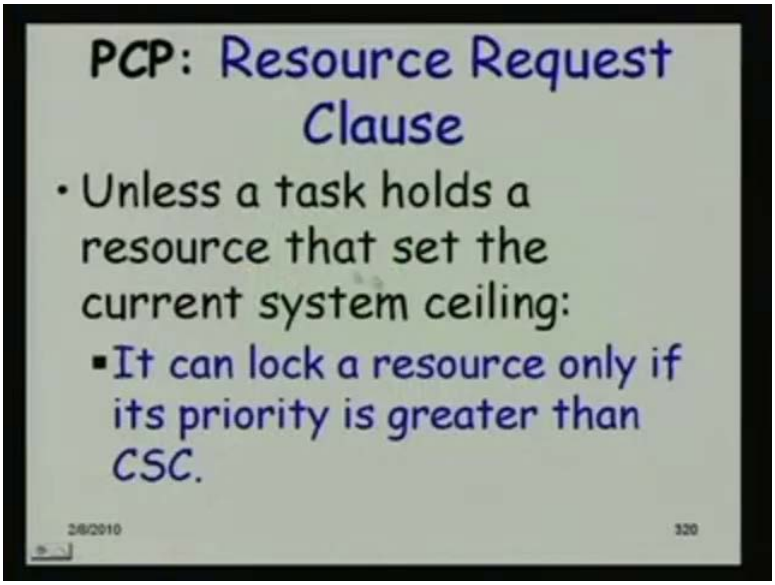
One more problem in this protocol is that, we need to statically convince this priority right, prior only we to determine what all the task use this resource, dynamically a task cannot request any resource.

Of course, see any embedded real-time system, we will know what are the resources required by a task. It is not the dynamically they request the resources, does not happen. We should know that what exactly is the data structures, what are the devices etcetera used by a resource before hand for any...

Now, I may make the request as…

Dynamic requests are not allowed. Let us see, some devices been added, and then how to use it etcetera that will create problem or some new task which has a new resource - data structure, how to access no those, these are statically decided during design time, what are going to the tasks and resources.

(Refer Slide Time: 43:24)



Now, let us proceed with the other clauses here. So, the first one was the request clause, and we saw that unless a task holds a resource that set the current system ceiling. It will be prevented from locking any resource except when its priority is greater than CSC. Now, the request there is… sorry.

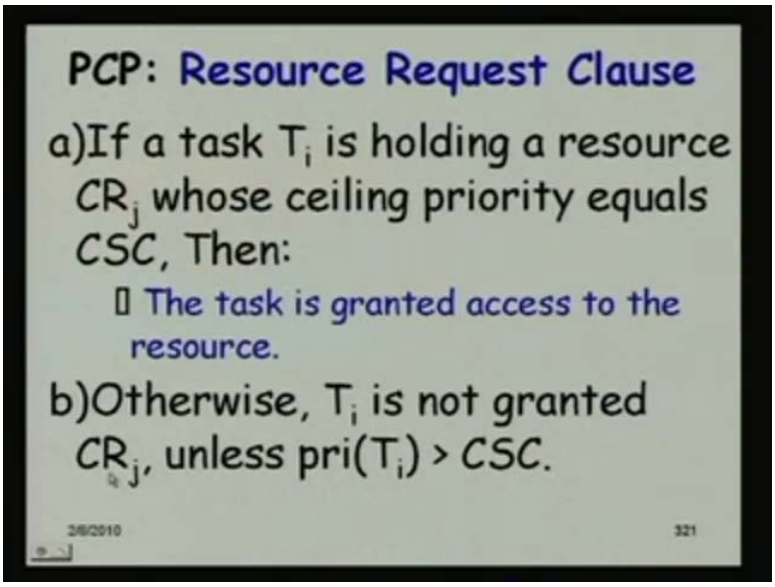
That point you left that thing, if it has a resource we set this system ceiling.

Yes.

Then, it can acquire any resource.

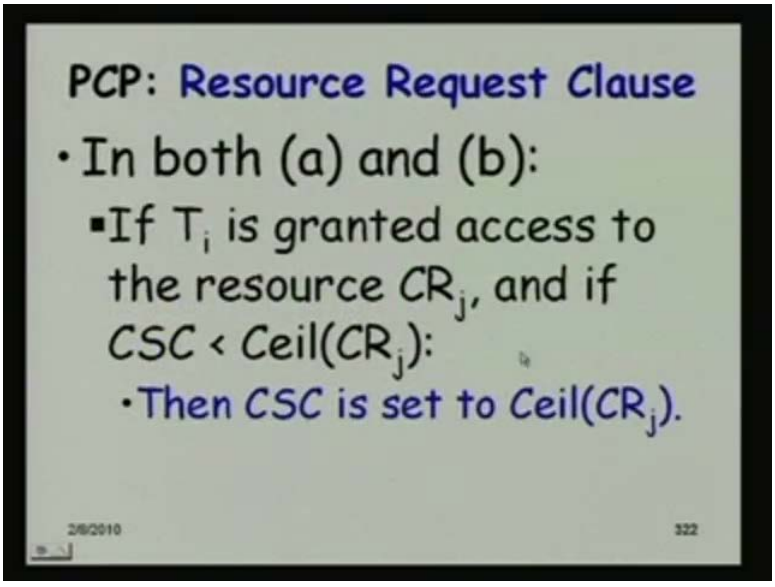
Exactly, if it has set to the system ceiling, then any request by it will not be checked against CSC. It can lock any request that it needs. So, let us look at the resource request clause more formally.

(Refer Slide Time: 44:02)



So, we can say that task if a task is holding a resource with ceiling priority equals the CSC, I mean that is the task which had set the CSC, then the task is granted access to the resources. Otherwise, T i is not granted some requested CR j, unless the priority of T i sorry T j it should have been (Audio not clear. Refer Time: 44:29 to 44:35) no T i it is… Otherwise, otherwise T i is not granted the resource. Unless, the priority of T i is greater than CSC; whatever we said just written slightly in the pseudo code form right.

(Refer Slide Time: 44:51)

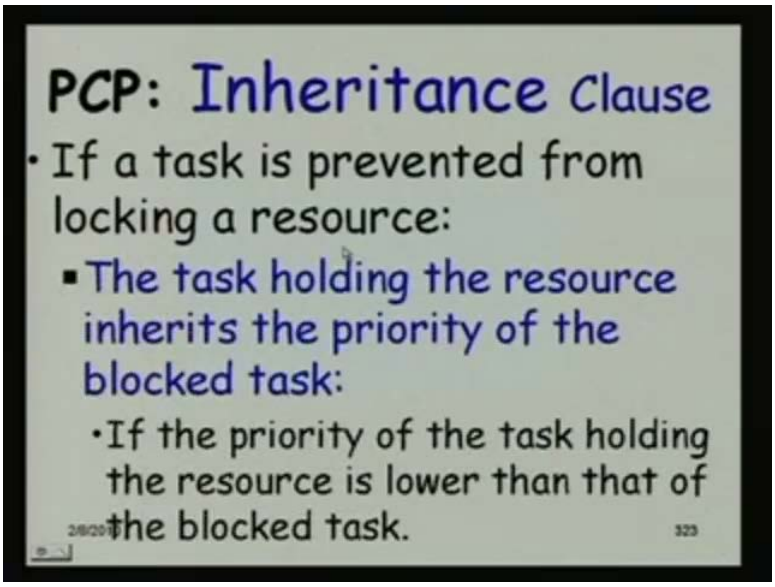


Now, in both a and b, if T i is granted access to the resource, then CSC is set to the ceiling CR j. So, whatever is the maximum ceiling value, the CSC gets changed. As soon as there is a resource allocation, the ceiling value - will the CSC will get updated. Because CSC will keep track of the highest ceiling value that is of the resource currently in use right. So, whenever we grant a resource we have to change the CSC if required.

Sir, CSC can decrease by the course of time.

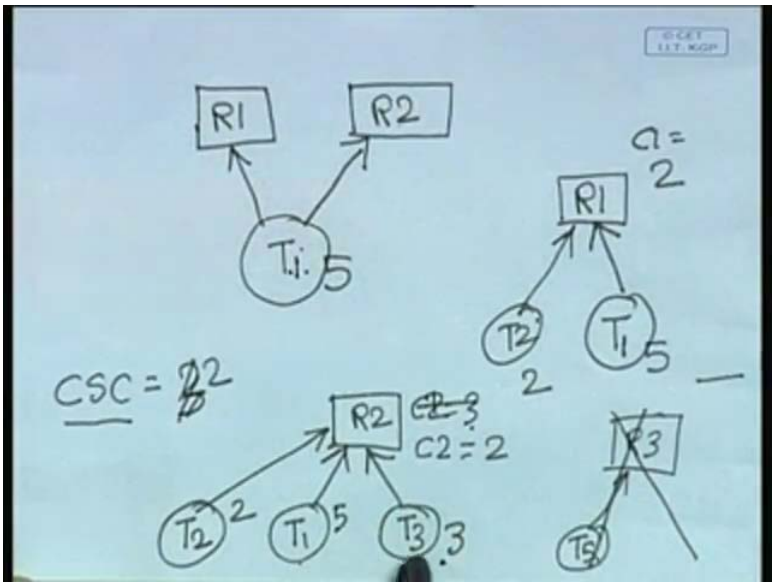
Yes, when there is a release of resource it will decrease. So, that also that clause you will see. Right now, we are just looking at the grant clause, then we will look at the release clause.

(Refer Slide Time: 45:44)



Now, let us look at the second clause in the grant scheme that is the inheritance clause. If a task is prevented from locking a resource, it had requested some resource, but it is prevented, and then the inheritance clause applies. The task holding the resource will inherit the priority of the blocked task. So, let us look at the example that we are constructing. So, the first example we had constructed is I think this one where we had said that say two tasks sorry three tasks are there. And they are, there are two resources R1 and R2, and T 1 requested R2, and CSC is set to 2.

(Refer Slide Time: 46:23)



Now, let us say T 2 requests R1 at that time, then T 2 will get blocked. T 2 started was ready, and then T 2 is blocked just because the resource is not available. And then the priority of T 1 will be raised to 2 by the inheritance clause right. So, just listen to the scenario again, T 1 started executing, priority is 5 and then acquired the resource R2, once it acquired the resource R2 its priority did not change; it is still priority 5, but the only thing that changed was the CSC value was set to 2. Now T 2, because its priority 2 it is started executing. As soon as it became ready started executing preempted T 1, the low priority task started executing, after sometime it needed R1.

Say R2, sir, T 2.

T 2, it needed R1 sorry R2, does not matter whether it is R1 or R2.

Sir, it should, it be on same resource.

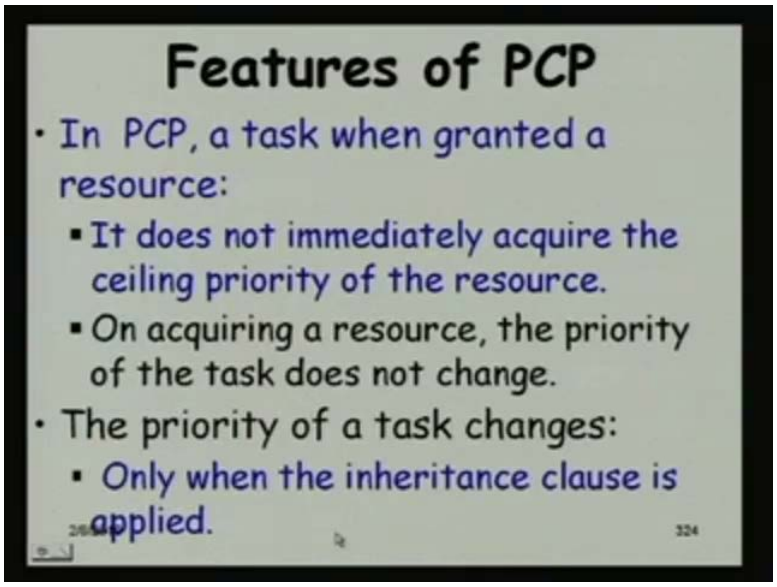
No, not necessary, it can be any resource. I hope that point is clear; that any resource request is governed by that rule. So, T 2 may request R1 also. So, let us say T 2 requested R1, and then this request the way it will be handled is, because it is not the one which set the system ceiling, the set by T 1. So, T 2 priority will be checked against current system ceiling already 2. So, CSC will block sorry the T 2 will block. Its priority is not greater than CSC. So, T 2 cannot acquire R1, and it will block. And when it blocks T 1's priority will get increased by the inheritance clause. Is that ok?

So, whenever a higher priority task, see initially when it acquired a resource, its priority does not change only the CSC value changes. And whenever any other task is prevented from acquiring any resource and is blocks, than the task that was holding the resource its priority increases by the inheritance clause.

The task that is blocked has got lower priority that will.

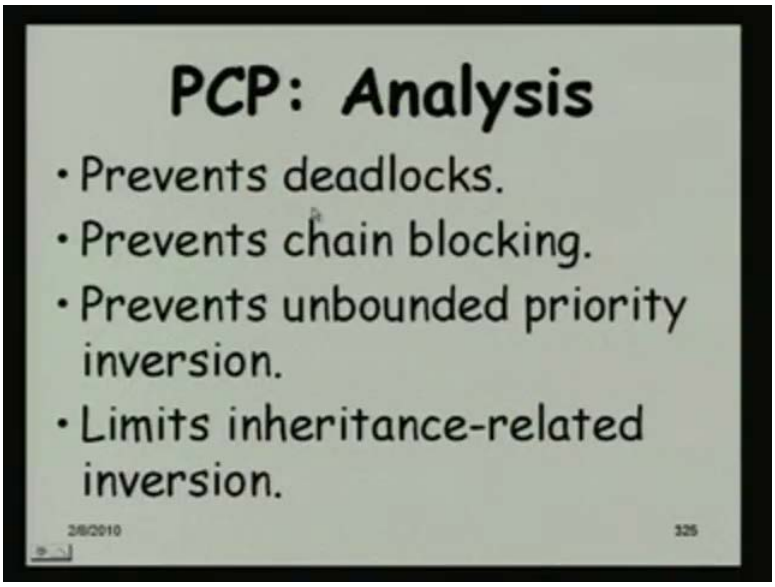
Then nothing will happen. See, inheritance clause applies only when the priority is the way the blocked task has a higher priority than the task that is executing. See, if it was a lower priority, then even this situation will not arise. Like it started executing and then requested resource that scenario will not exist at all. So, we are talking of only higher priority task requesting a resource. So, let us look at some of the features of the priority ceiling protocol. Look at 1 or 2 one example and then, we will look at the release clause.

(Refer Slide Time: 49:58)



So, as you we have already seen that - a task when it gets a resource, its priority does not change. It does not become the ceiling priority like it use to happen in the highest locker protocol. In fact, it does not change at all. Only when a higher priority task waits for being denied a resource, then the inheritance clause is applied.

(Refer Slide Time: 50:43)



So, we will see that this simple scheme not as simple as highest locker slightly more complex, because we have a system variable which is set, and when a resource is requested there are two clauses one is request clause where it is checks against the ceiling priority. And then, if it is the task which set had set the CSC then it is it is not checked against CSC. So, and then the inheritance clause. We will see that this will prevent the deadlock situation, it will prevent chain blocking, it will prevent unbounded priority inversion and it will reduce the inheritance-related inversion to a large extent. Cannot say that totally eliminated inheritance-related inversion will still be there, but substantially reduced. So, let us look at this.

Sir,

Yes, please.

Previous slide.

This one inheritance plus…

Sir, what is the use of this?

Why the inheritance clause is required is it.

Yes.

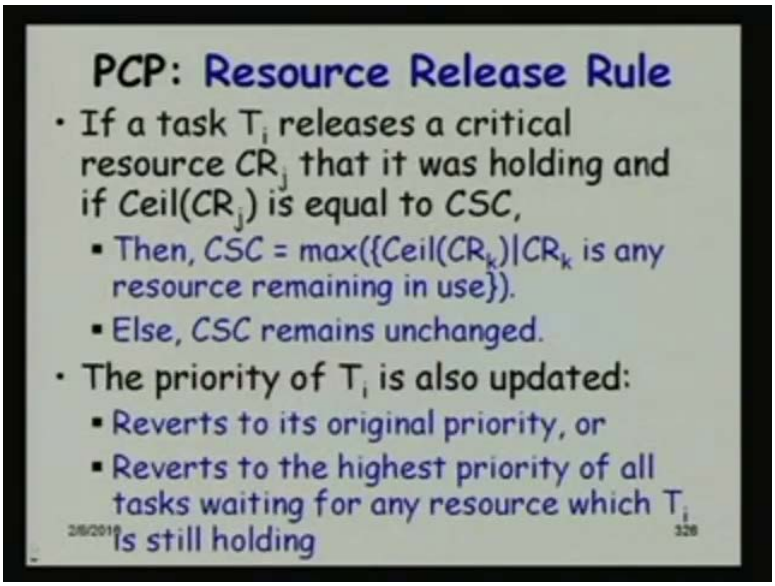
So, his question is why inheritance clause is required. Can anybody answer this question? Why do I need the inheritance clause?

Is it for unbounded.

Exactly, exactly. See, unless you have the inheritance clause, the task whose priority is higher right, it will just intermediate priority tasks will keep executing, and the higher priority task which is waiting will keep on waiting. That is the unbounded priority inversion.

So, the inheritance clause is there to prevent unbounded priority inversion, but it is not really raised to ceiling priority, but to the priority of the task that is waiting for the resource. So, without inheritance clause there will be unbounded priority inversion. Is that ok? Now, let us proceed further.

(Refer Slide Time: 52:56)



There is the resource release clause also. When a resource is released by a task which was holding the ceiling, then if this was the maximum of all ceilings that are of the resources currently in use, maximum ceiling of all resources that are current that are currently in use, then the CSC is changed to the one - that is current maximum. If it is not the maximum then nothing changes it just releases the resource. And of course, the if it had inherited a priority, if an inheritance clause was applied then that also needs to be reverts. Is it not?

So, if it inheritance clause was applied, then it will revert to its original priority. If it was it is not holding any resource, if it is holding some other resource, then of course, it will get the highest priority of all the blocked tasks on those resources; just the opposite of the resource release - resource grant clause.

Here, the CSC value will get updated if applicable and it will revert to its old priority if applicable right. So, we will just stop here, and we will just take a break and meet for the next class.