Real-Time Systems

Prof. Dr. Rajib Mall

Department of Computer Science and Engineering

Indian Institute of Technology, Kharagpur

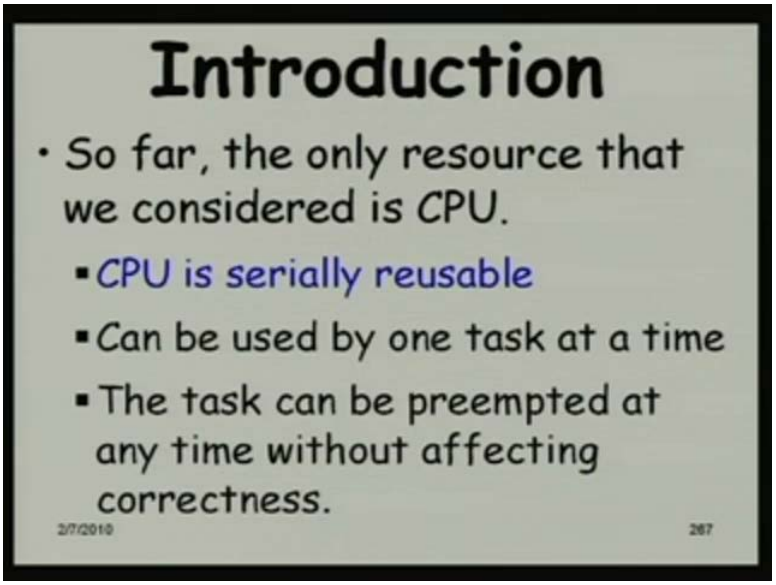
Module No. # 01

Lecture No. # 13

Resource Sharing among Real-Time Tasks

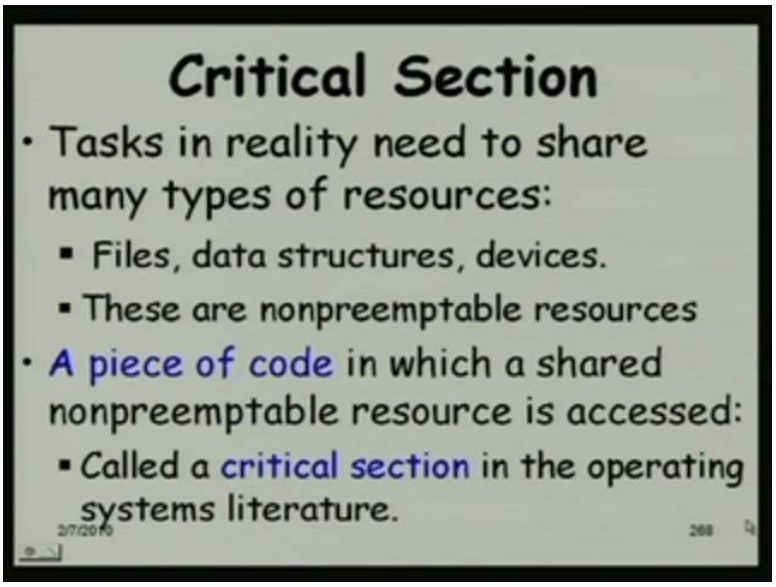
Good morning, so let us get started. Today, we will discuss about resource sharing among real-time tasks; because, so far, we have considered the situation where tasks share no resource, but that was unrealistic. In a real situation - practical situation - tasks will be sharing some resources or other, and we will just discuss about the issues that arise, and the difficulties, and how these are addressed in the commercial operating systems, real-time operating systems. So, let us start with that.

(Refer Slide Time: 01:01)



So far, we considered CPU is the only resource that is shared among tasks. And CPU is a serially reusable resource, in the sense that, it is used by only one task at a time, just like the other resources we will discuss about memory and devices, a CPU is also used by one task in a uniprocessor system, but a difference is that the tasks can be preempted at any time without affecting the correctness.

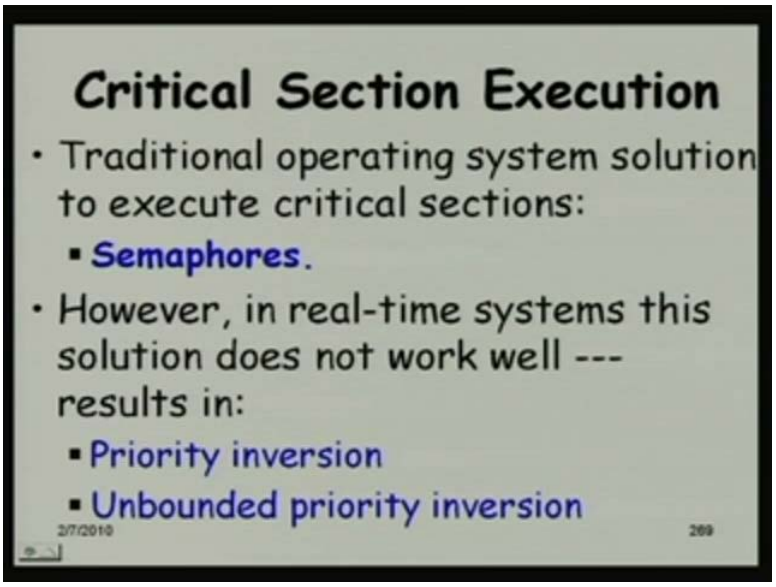
(Refer Slide Time: 01:42)



So, as long as there is only one task, it is all right, but the tasks need to share many other types of resources other than the CPU. For example, the files, data structures and devices are common resources, that are shared, and many of these resources need to be used in a preemptable mode.

Actually, the part of the code in which a nonpreemptable resource is accessed is called as the critical section in the operating system literature. So, it is the part of the code where once the code is entered, execution starts. It should complete without releasing the resource, right. So, that is the idea of a critical section. Even though, sometimes we loosely talk of critical section as a resource, but it is actually a piece of code.

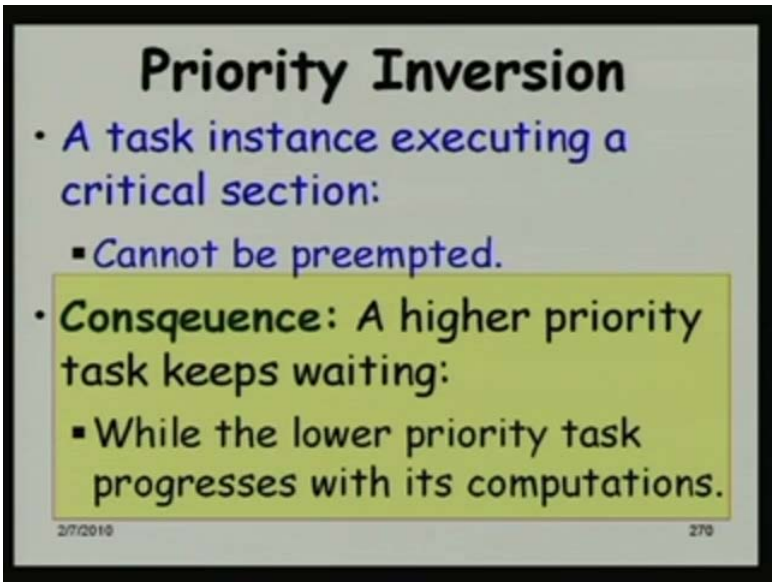
(Refer Slide Time: 02:45)



In the traditional operating systems, the typical solution is to use semaphores. But in case of real-time tasks, if we try to use semaphores, we will get problems. The problems are known as priority inversion and unbounded priority inversion.

Let us understand this problem, because very fundamental problem, very basic thing. Even if, you know, tell somebody that I have done a course on real-time system, they are likely to ask, what do you mean by priority inversion and unbounded priority inversion? Possibly the first question, right? So, that way it is very fundamental question.

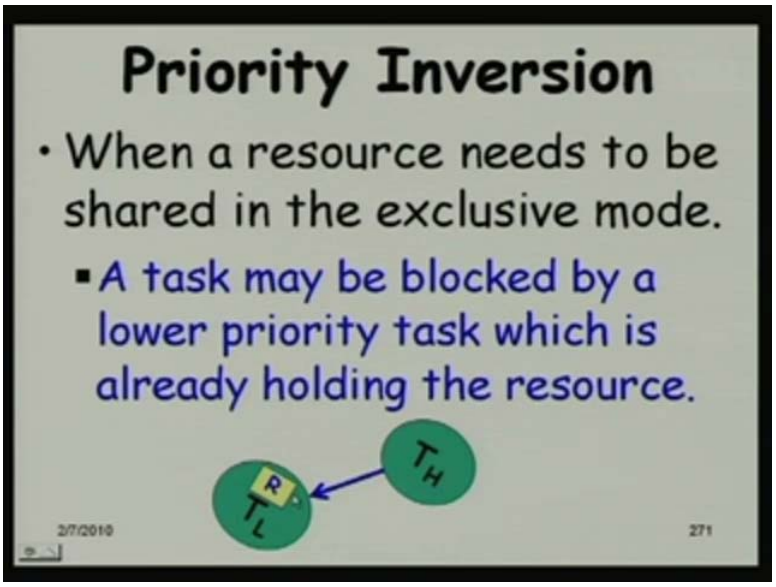
(Refer Slide Time: 03:36)



So, first let us understand, what is meant by priority inversion. The idea is that, when a task is executing its critical section, it should not be preempted from resource uses. I am not saying CPU preemption; I am saying that when a task is executing its critical section, it should not be preempted, because then the resource becomes inconsistent right. These things we know from the first course on operating system.

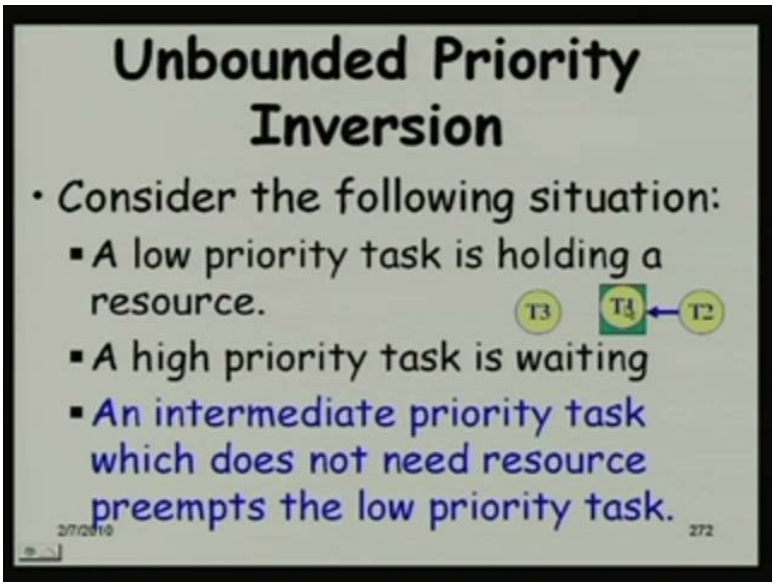
But the consequence in a real-time application is that if a task cannot be preempted - a lower priority task - then higher priority task will keep on waiting, and we will have a situation, when the real lower priority task is making progress with its computation, and higher priority task is waiting, and this is against the basic assumption we had in the scheduling algorithms; as, as long as there is a higher priority task, it should have execute; the lower priority task should yield. So, this is the priority inversion occurring where a lower priority task is executing and the higher priority task is waiting.

(Refer Slide Time: 04:59)



Now, when a task is executing and has accessed the resource in the exclusive mode, then it can block a higher priority task. Let us say, that this is a lower priority task T L, which has acquired the resource R, and after sometime, the high priority task was released, and it waits for the resource R, and the lower priority task until it completes its execution, the resource will not be available to higher priority task, it cannot execute right, that is the main problem.

(Refer Slide Time: 05:51)

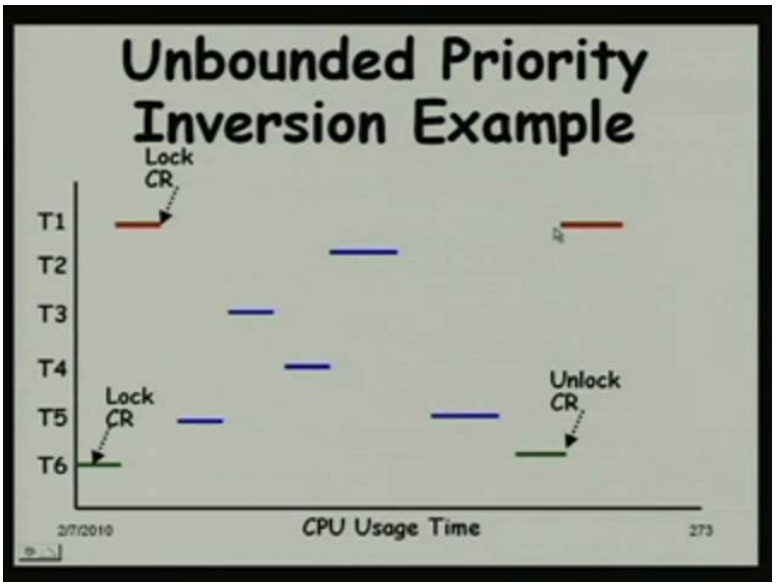


But we will see that priority inversion by itself is not a big problem. A simple priority inversion can be handled, we will see how. But the one that is very difficult to handle is unbounded priority inversion. Let us understand, what is meant by unbounded priority inversion.

Consider the situation, that a low priority task is holding a resource. So, there is a resource and a task was executing, after sometime, it just acquired the resource, and meanwhile, a higher priority task was released, and is waiting for the resource right. This is the simple priority inversion situation. But what if some tasks, which do not need the resource, that are of higher priority than T 1, but lower priority than T 2, also get ready and start executing, because they do not need the resource, they will preempt; they will not wait for T 1 right; they will preempt T1.

So let us say, we have a task T 3, which does not need the resource and it preempts T 1; so, T 2 will wait for T 3 to complete, and then T 1 to complete, but the thing is that there may not be just to one intermediate priority task, there might be multiple of them, and as a result, T 2 keeps waiting for all these intermediate priority tasks to complete, so that T 1 will finally execute, and release the resource, and then T 2 will proceed; but that may be too late.

(Refer Time Slide: 07:45)



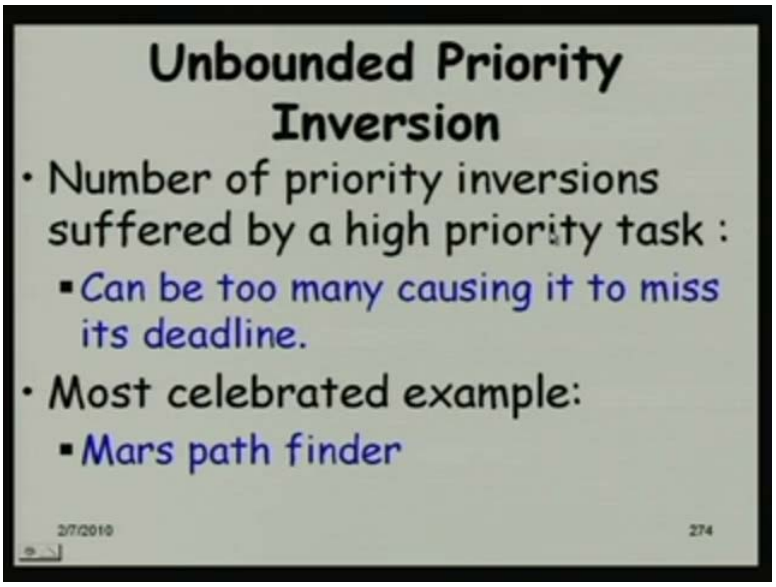
So, this just an example of a unbounded priority inversion. Let us say, we have six tasks and T 1 has the highest priority, and T 6 the lowest priority, and this is the time line, and we will just show the tasks, which is using the CPU.

Let us say, T6 is the lowest priority task executing at some instant, and then, it just locked a resource - critical resource - and after sometime, a high priority task has become ready and immediately preempts the lower priority task - T 6 - starts executing, but after sometime, it needs the resource CR; whenever, it is use a lock CR call, it gets blocked.

Now, T 6 would execute, but what has happened meanwhile, that other tasks have become ready and we are just waiting. So, T 6 does not get the CPU, T 5 gets it, executes for some time, and T 3 meanwhile is waiting is executing, T 4 is executing, T 2 executes for some time, T 5 again gets back the CPU, starts executing, and after all this, T 6 gets to use the CPU and completes resource usage, unlocks the resource, and then only, the task T 1 can start executing.

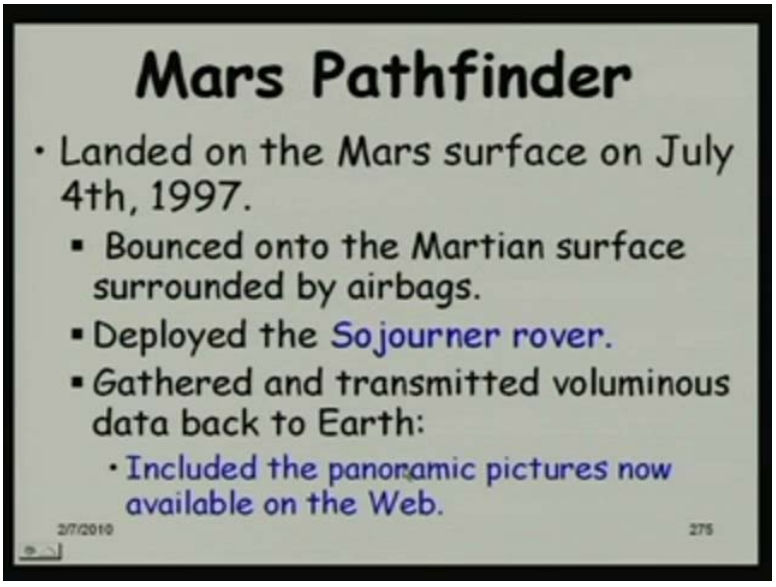
So, the priority inversion that T 1 suffered is from this point up to this point right, or until this point, it was suffering priority inversion and is likely to miss its deadline.

(Refer Slide Time: 09:56)



So, the number of priority inversion suffered by a high priority task can be too many, and is likely to miss deadline, whenever unbounded priority situation, priority inversion situations occur, the high priority task are very likely to miss its deadline, and there are many situations where systems have malfunctioned, because of the priority inversion problem. In the most celebrated example is the Mars Pathfinder; you find widely documented in the internet and in the book.

(Refer Slide Time: 10:32)



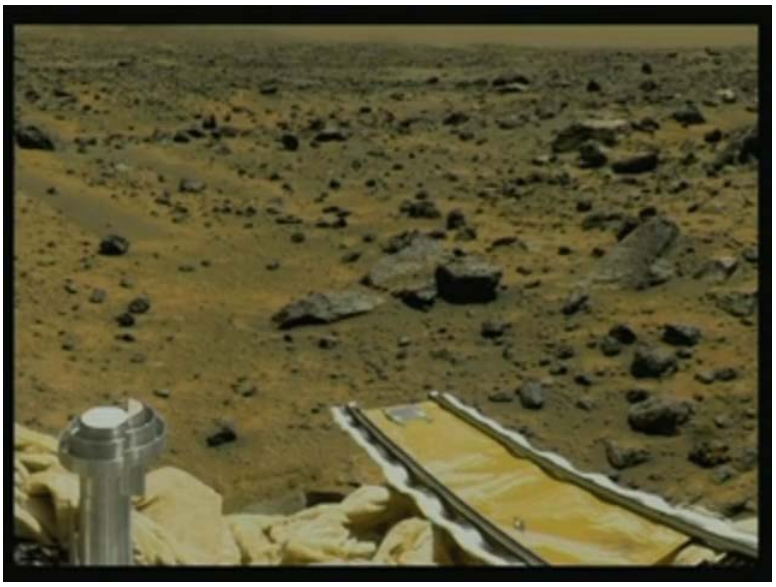
So, this was sent to the Mars, and it landed actually on 4th July 1997, and it bounced onto the Mars surface by airbags, so that, it will not get damaged, and then, it had a vehicle which will roam around the Mars surface called as the Sojourner.

So, the Sojourner rover is the one, which was released to roam around the moon surface. The Sojourner actually, roamed around, gathered transmitted data - voluminous data - started transmitting to the Earth, and there are many pictures that were sent, many of these pictures are available on the internet, you can easily find them, just a simple Google search will give you.

(Refer Slide Time: 11:35)

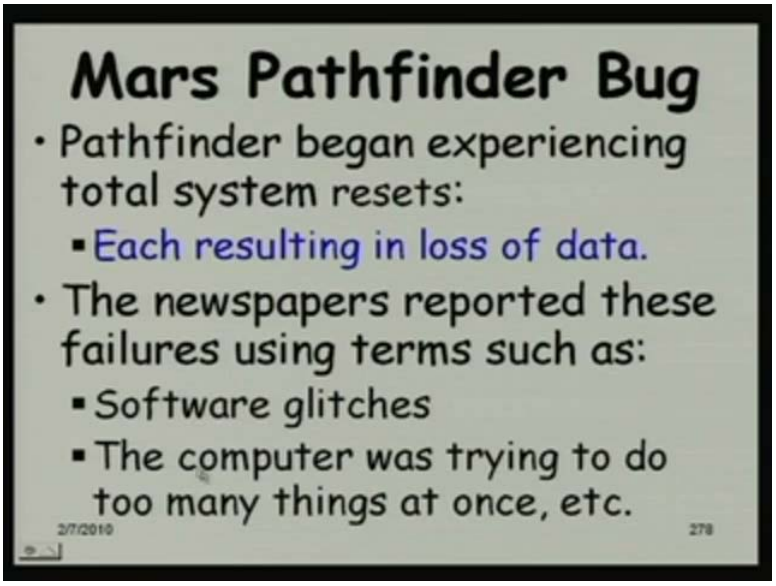
This is one of the picture; see the rover vehicle here, the land rover, sorry, the Sojourner rover is on this Pathfinder; it is sitting on the pathfinder.

(Refer Slide Time: 11:55)



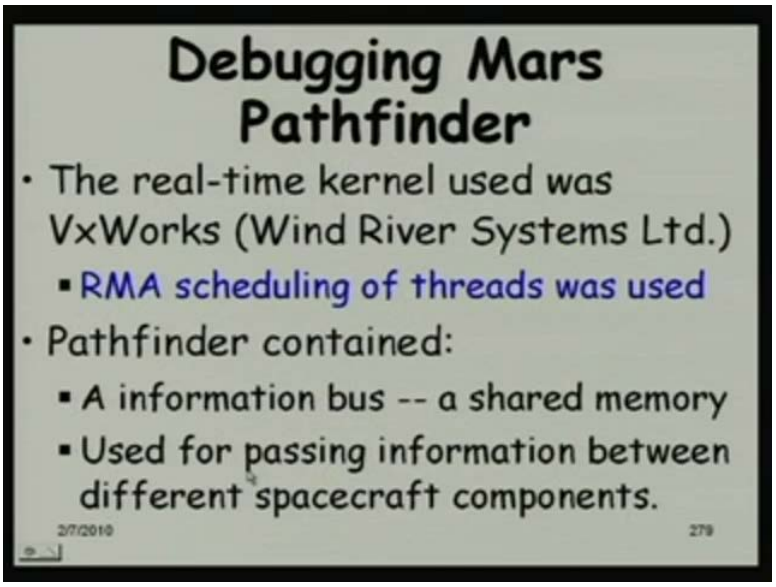
And this is one of the pictures sent; just see the surface of the Mars taken by the rover.

(Refer Slide Time: 12:06)



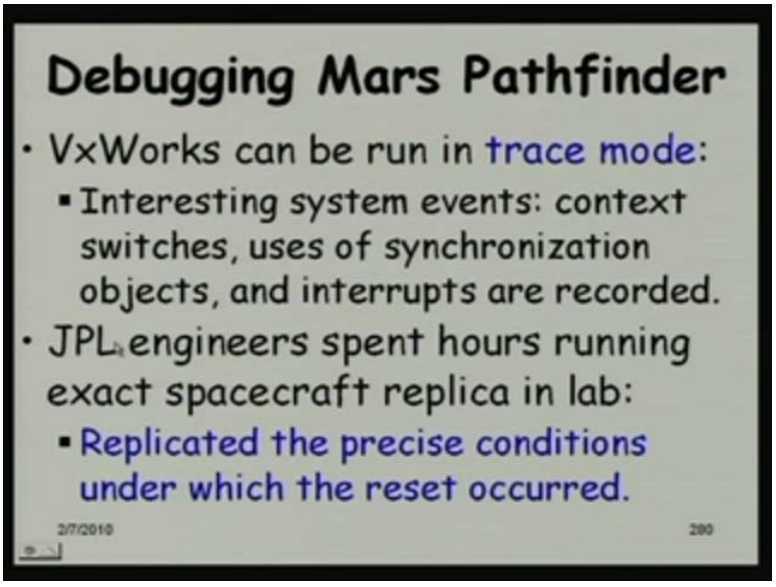
But unfortunately, soon it started experiencing total system resets. Each time the system reset, huge amount of data was lost, and it appeared in all news papers, saying that Mars Pathfinder is getting into problem, and they used many terms; some said software glitches have occurred in the Mars Pathfinder, it is not taking commands, it is resetting itself, and some papers, they said that the computer was trying to do too many things at once and it was crashing, that was other reports.

(Refer Slide Time: 12:48)



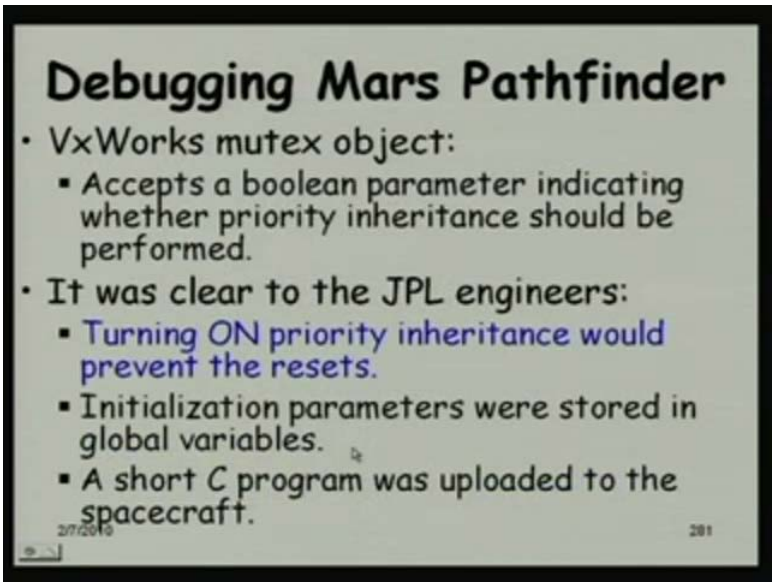
And, the pathfinder actually, was using a real-time kernel, which we will discuss later as a type of commercial operating system - the VxWorks - which is the vendor is Wind River Systems, who sale this VxWorks real-time operating system and RMA scheduling of the threads was used, and the pathfinder contained an information bus, basically a communication channel, which was a shared memory to communicate among various tasks that execute on the Pathfinder and land rover, sorry the rover.

(Refer Slide Time: 13:34)



Now, once it started resetting itself, one thing that was done is that the VxWorks, it provides a facility to run it in the trace mode. In the trace mode, we, we will also discuss when we discuss about these operating systems, in the trace mode, the important system events like when context switches occur, when synchronization primitives are used, when interrupts occur, the different events are recorded. And, the the Jet Propulsion Laboratory engineers, they spent hours trying on a replica of the space craft that they had in the lab, and where they had set the trace mode on, and they were trying to replicate the precise conditions under which the reset was occurring.

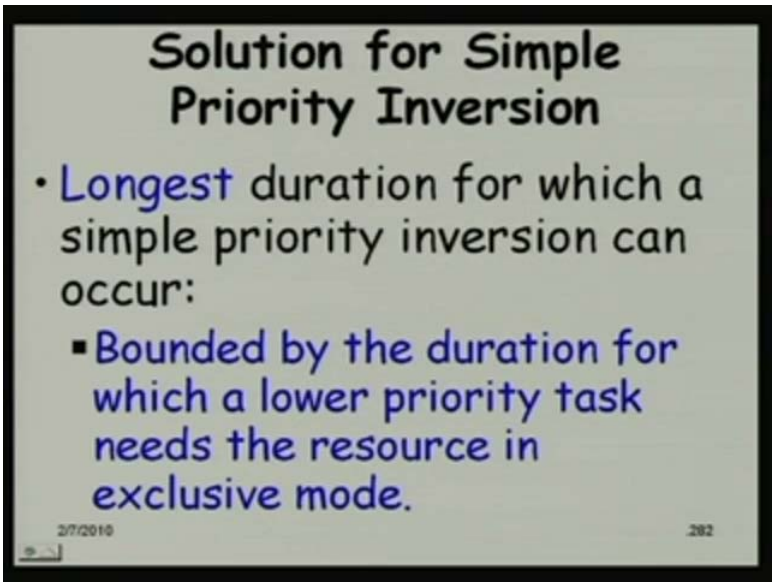
(Refer Slide Time: 14:37)



Now, after spending lots of hours there, they came to the following conclusion: that the VxWorks uses as a mutex object, which we will call as the technique to handle priority inversions, and it accepts a boolean parameter, indicating whether priority inheritance… we will we will discuss priority inheritance, whether priority inheritance would be enabled or not, and after spending many hours, it was clear to the JPL engineers, that if the priority inheritance is turned on, then this kind of situation would not occur.

And, the initialization parameters, that is including whether the inheritance is set on or not, priority inheritance is set on or not, these are stored as global variables, and this header part of the C program, where the variable is set on, was uploaded to the space craft, and then, the system started working.

(Refer Slide Time: 15:49)

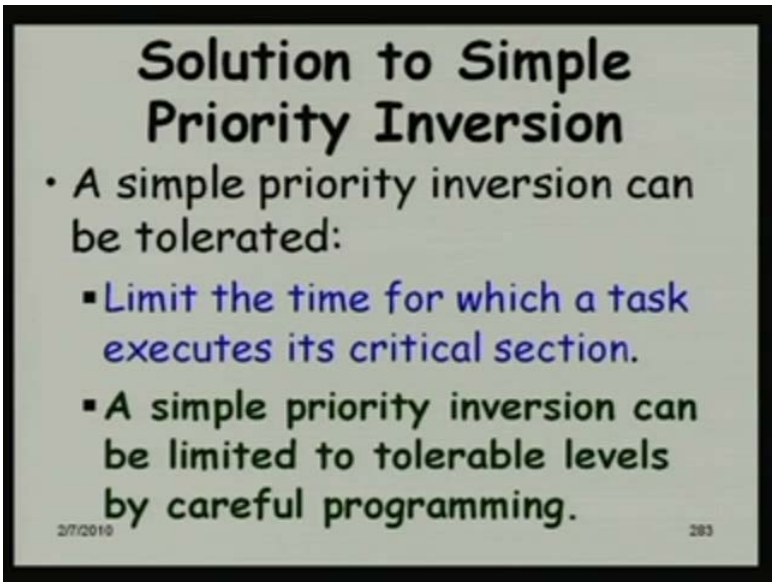


So, this was a celebrated example, where the system failed due to unbounded priority inversions occurring, it was debugged, found that yes, because the mutex variable was not initialized and priority inversions - unbounded priority inversions - are occurring; therefore, the system was resetting each time, and once they uploaded the header with the priority inheritance turned on, and from that point onwards, the vehicle started working, and for another six months or so, it worked correctly, and sent many pictures, until it finally, was abandoned.

So, let us see, when a simple priority inversion occurs, that is a high priority task is waiting, what can be done? Now, the longest duration for which a simple priority inversion can occur, when a high priority task is waiting for a low priority task, it is basically, how long the low priority task will use the resource in the exclusive mode; is it not? Simple thing.

The maximum duration for which the low priority task will need the critical resource, is the duration for which a simple priority inversion will occur. We are not talking of unbounded inversions.

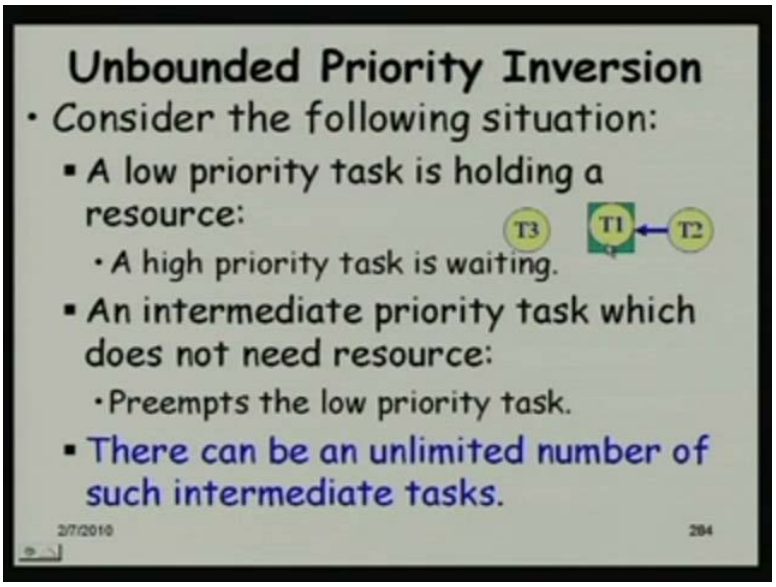
(Refer Slide Time: 17:16)



Now, if simple priority inversions were to be handled, it is not very difficult. Somehow, reduce the maximum duration for which a task will need a critical resource. How can you do that?

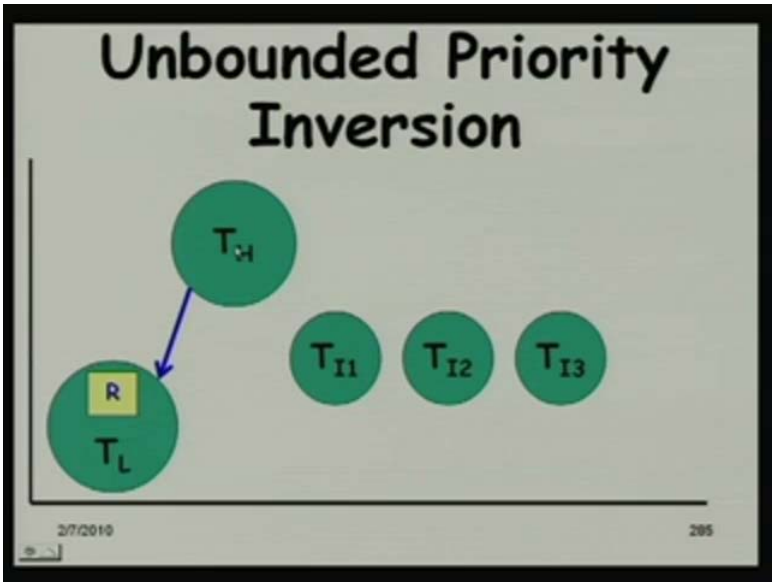
You limit the time for which a task executes in critical section, may be by careful programming or may be by splitting its access to a critical resource across multiple instance right, do part of it, release it, and then, acquire it, try to do something for low priority task; many solutions can be proposed. So, simple priority inversion, if it was the real problem, it would not be really that difficult to handle it, but as I was saying that the tricky problem is the unbounded priority inversion case. Now let us see, how unbounded priority inversion can be handled.

(Refer Slide Time: 18:10)



So let us see the situation, a low priority task is holding a resource, a high priority task is waiting; so, low priority task is holding, high priority task is waiting, and intermediate priority task has come in, preempted T 1 started executing and T 2 is waiting all through.

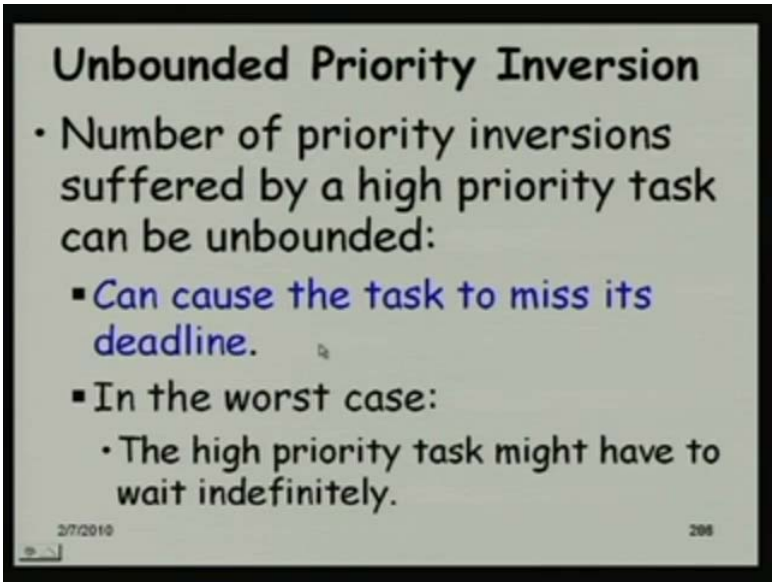
(Refer Slide Time: 18:37)



And there are many instances of those intermediate priority tasks, which are keeping on coming in, and preventing the high priority task from acquiring the resource, because low priority task is not able to complete its resource uses.

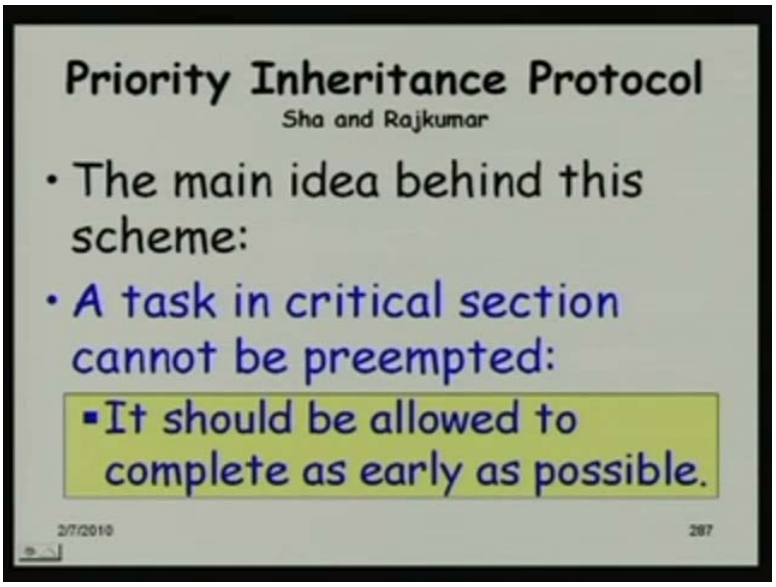
I hope that is clear. Is it not? So, since this is an important point, and later also, somebody would ask you - what is unbounded priority inversion? Just as a final example, let us see this situation that a low priority task has acquired a resource. High priority task is waiting, and intermediate priority tasks, one after other, are rising and requesting CPU usages, and meanwhile, low priority task is not able to complete; high priority task is waiting all through.

(Refer Slide Time: 19:35)



So, the high priority task that is waiting for all these unbounded priority inversions is likely to miss its deadline.

(Refer Slide Time: 19:48)



A simple solution was proposed in the form of a priority inheritance protocol by Sha and Rajkumar. The main idea is that, a task in its critical section cannot be preempted from resource uses.

We can rule that (( )) we can rule that completely to the initial situation.

But how can you, see the question, I mean, the suggestion is giving there is that we can somehow get back to the situation where the task started using the resource, but how do you do that? Because, you know, it has used the task, and we need to make a copy of that, and then, we need to…

(( )) check points also.

But see, it is not a check point; you have just take a complete backup of the resource; restore the resource. It is computationally expensive right and storage wise expensive.

Actually, it can miss the deadline.

It takes time.

Instead of reading that much long time restoring to a just previous point is can (( )).

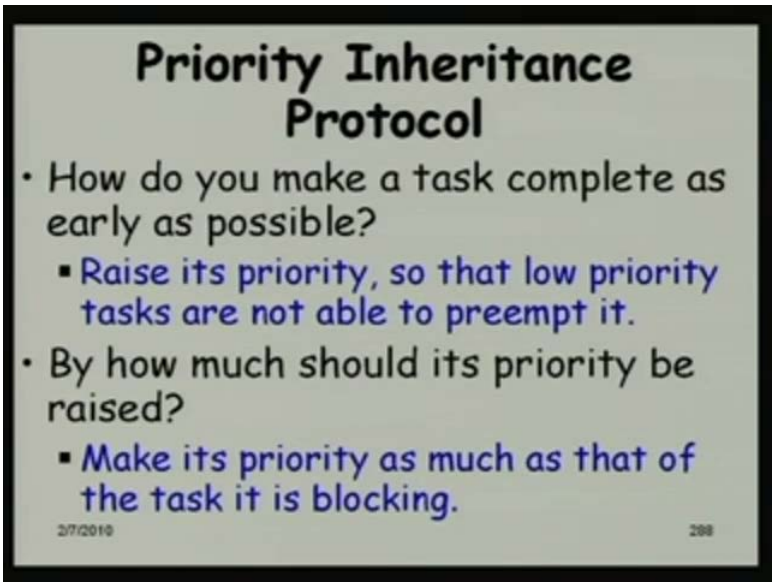
No, let us just understand the situation actually. The situation is that, the task needs a resource for few milliseconds right, may be microseconds. Now, if we have to keep a trace of what it has been doing or let us say a check point where the resource is backed up and then restored, then it is not that it has just used as resource, it might have performed other computations based on that resource uses right, and all those are going to become inconsistent. So, it is not just one critical resource you need to backup, may be all other changes that it has done to other resources need to backup; it is a tricky problem, considering the time constraint of few millisecond.

Overheads also using.

Overhead, yes, time overhead also, the task is going to miss its deadline, if the high priority task is waiting for the resources to be restored, when the constraints are few milliseconds, microseconds, deadlines can be missed.

So, let us see what the suggestion or what is the solution they give? They said that see, a task, of course, cannot be preempted, low priority task once it has acquired a resource, but only thing that can be done is that allow it to complete as early as possible, because the simple priority inversion is not a problem; the problem is that if unbounded priority inversions occur. So, a task in the critical section cannot be preempted, that is for sure. What can allow what can… we can do is allow the task to complete execution without incurring the preemptions by intermediate priority tasks.

(Refer Slide Time: 22:52)



So, how do you make a task that is holding the resource to complete as quickly as possible?

So, the idea is that, we can raise its priority. So that, the low priority task is not able to preempt it right. So, that is the basic priority inheritance principle. The idea is that, see the low priority task, because the priority was low, higher priority task was able to preempt it; those are the intermediate priority tasks. Now, what we can do is raise its priority, but raise by how much?

To that extent to that extent which is (( )) resource (( )) process… the one less than that priority one less than the (( )) highest resource that equal to the…

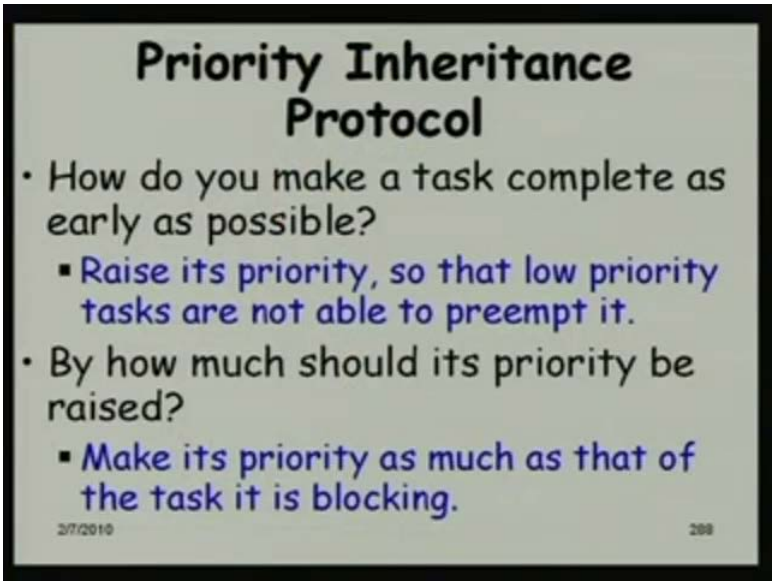
Equal to what?

(( )) equal to the highest priority task waiting for the resource.

Yes. yes that is exactly… See, what they are saying is that the task is should be raised to a priority, which is as much as the priority of the task that is waiting for the resource, because if you raise it to anywhere lower than that, then we will have intermediate priority tasks which will preempt it.

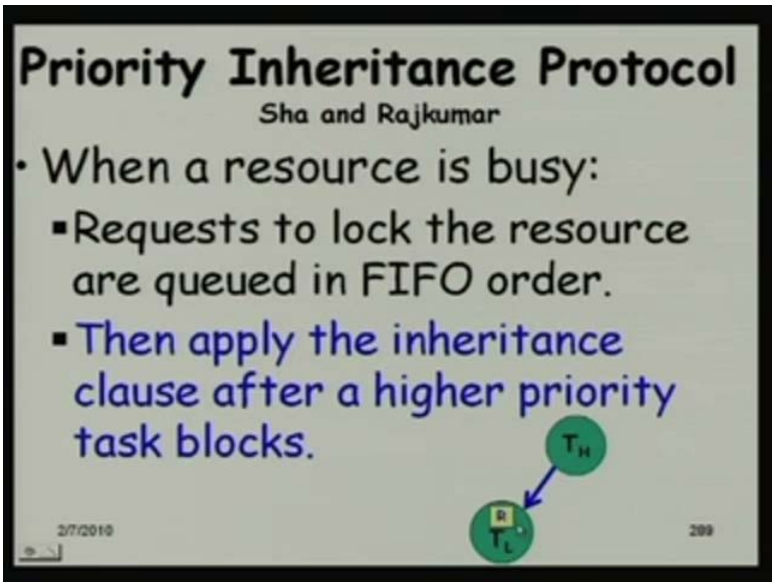
So, we should not have intermediate priority tasks coming in and preempting it, and that is basically equal to the highest priority that is waiting for the resource, and in that case, there will be no intermediate priority task. Yes, exactly, that is the solution that was proposed by them.

(Refer Slide Time: 24:50)



So, the priority should be raised to as much as that of the task that is blocking it, that is blocked right; the task that is holding the resource, should be, the priority of that should be raised by as much as the task that is waiting for the resource, the highest priority task.

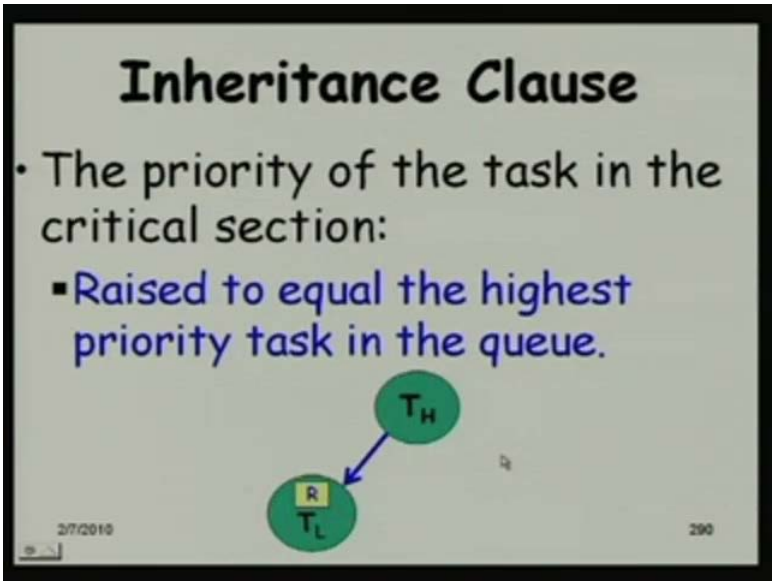
(Refer Slide Time: 25:11)



Now, when a resource is busy, the request to lock the resource are queued in FIFO order; see multiple tasks might come in, wait for the resource, right we just soon it just one task has waiting, but it is likely that multiple tasks might be waiting for the resource.

Can you visualize the situation? See, may be a low priority task T L was using the resource R, and meanwhile, some other task T i requested the resource right. T i started executing, and then requested, and it blocked; and then, T L was executing again, and then T H a higher priority task started executing, because that time it did not need resource, and it continued up till the point it started, requesting for the resource R, then that time it blocked, and there will be two tasks waiting for the resource R. Right? So, we will have to find the highest priority task that is waiting for the resource, and then, we have to raise the priority of the task holding the resource by that much.

(Refer Slide Time: 26:37)



And as soon as the task releases the resource, it should get back its old priority.

(( )) we are assigning to the maximum priority task.

Yes.

Then why we are using a queue? Why not a (( )) like where the maximum (( ))

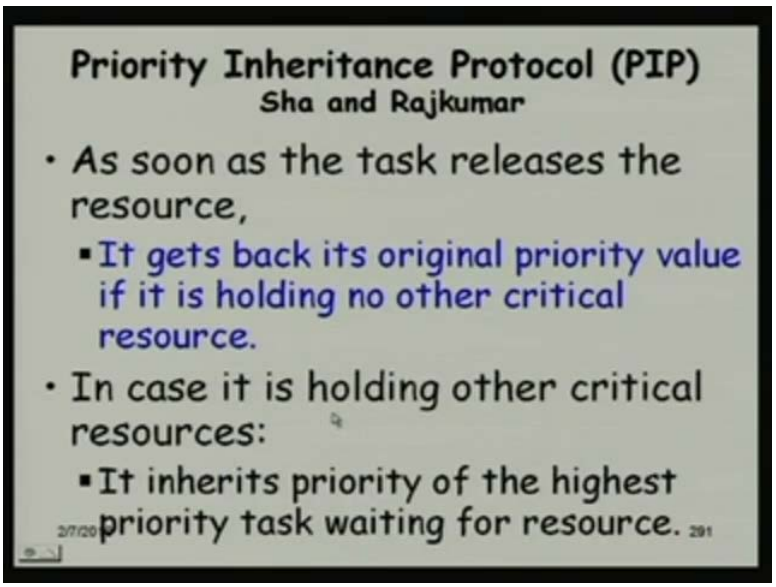
No, see actually, the ways… We will discuss that point later. See the question is that, why exactly we were saying a FIFO queue? See, many operating systems for simplicity, they let task wait in a queue, and then, use a round robin scheduling among them, some of the operating systems. Some operating system is just a queue, we will we will see it; we will discuss those issues with specific reference to commercial operating systems.

Some are just FIFO queue, some are FIFO queue with round robin scheduling, and number of tasks is not large, just imagine; just remember that the number of tasks that may be waiting for a resource may be just two or three maximum, and heap or something does not make sense there, because anyway, it just looking up three, does not take time; right a heap becomes meaningful, a priority queue, when the number of tasks waiting is large enough. So, let us proceed.

(( )) situation like this Mars pathfinder and (( )) that may be (( )) that may be more complex and more that may be …

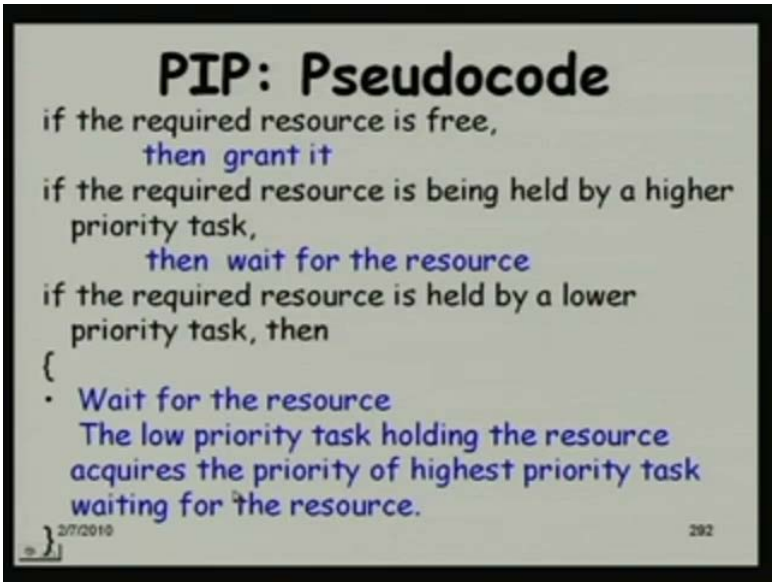
See a more complex system, as you are saying has many tasks right, but many tasks waiting for a resource for a small interval of time is bounded; it can be 3, 5 that is all. You cannot have many tasks in a millisecond arising and waiting for the resource. Right? So, a queue is a solution that is used in many operating systems.

(Refer Slide Time: 28:42)



So let us, we will discuss those things later. Now, let us understand the basic ideas here. So that, once the high priority task, sorry, once the low priority task has completed its resource uses, then it gets back its old priority; of course, this is natural to expect, but only thing is that, if it is holding some other resource, then it should get the, inherit the priority of the highest task waiting for that resource, it has released one resource, but it might be holding other resources. So, that we have to consider.

(Refer Slide Time: 29:28)



Sir (( )) holding two different resource (( ))

Yes.

Then, in that case, the highest priority task which is waiting on.

Of course.

(( ))

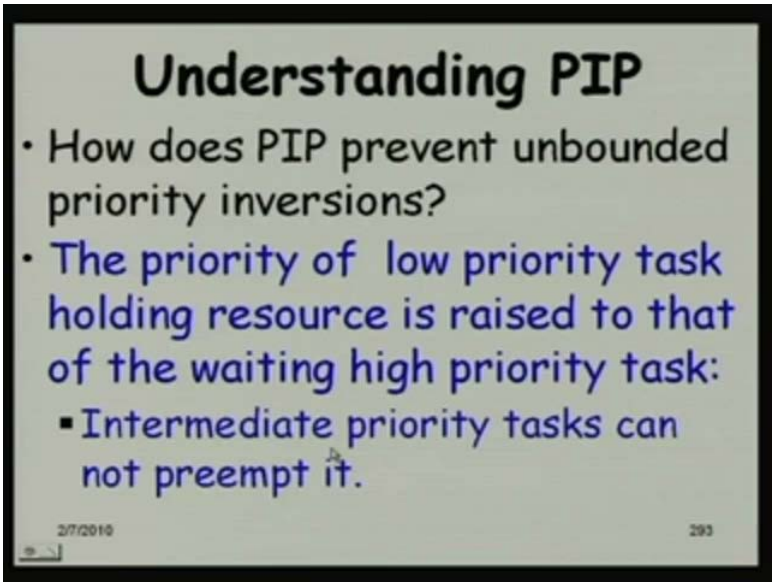
Exactly, exactly, of course. So, it should be the highest priority task among all its resources, whose priority should determine the priority of a task.

Now, let us look at the priority inheritance protocol, the basic pseudo code; simple one whatever we discussed just written in pseudo code form; whenever, there is a resource is available and the request is there, grant it, you do not check the priority whether it is a

low priority high priority nothing. As soon as the resource is, as long as the resource is available, its granted; if the required resource is held by a higher priority task, then wait for the resource, nothing needs to be done, because this not a priority inversion situation. Is it not? Because already higher priority task is using the resource and naturally a low priority task will wait for it.

But if the resource is held by a lower priority task, then we will have to apply the inheritance clause here. So, not only the high priority tasks wait for the resource, but the low priority task has to inherit the highest priority task in the queue.

(Refer Slide Time: 30:54)

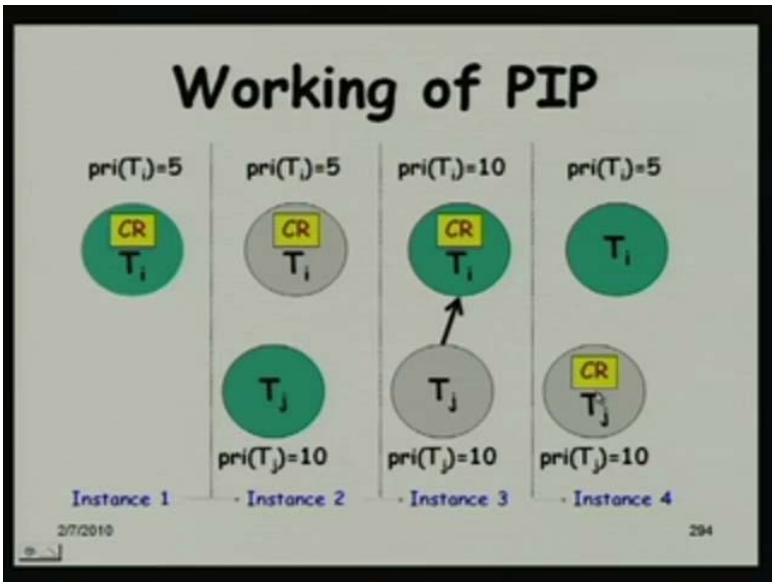


But let us understand, how does priority inheritance protocol prevent unbounded inversions?

Sir, the task cannot preempt now.

Exactly, because the priority of the task is raised to that of the highest waiting task, there is no intermediate priority tasks, which can preempt the task from CPU usage.

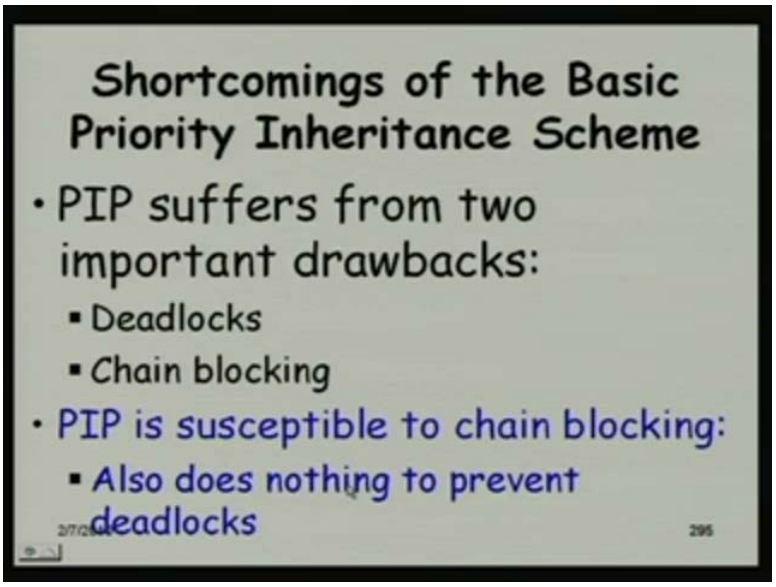
(Refer Slide Time: 31:21)



So, this just a example of the working of the inheritance - priority inheritance - protocol, a task T i with priority 5, was using the resource CR, and in the next instance, a task with priority 10 stared using the CPU, and then this was preempted, because T j did not require the resource, it started executing, but after sometime, it needed the resource, and it started waiting for the resource to be released by Ti, and T i got execution on the CPU, it started executing on the CPU.

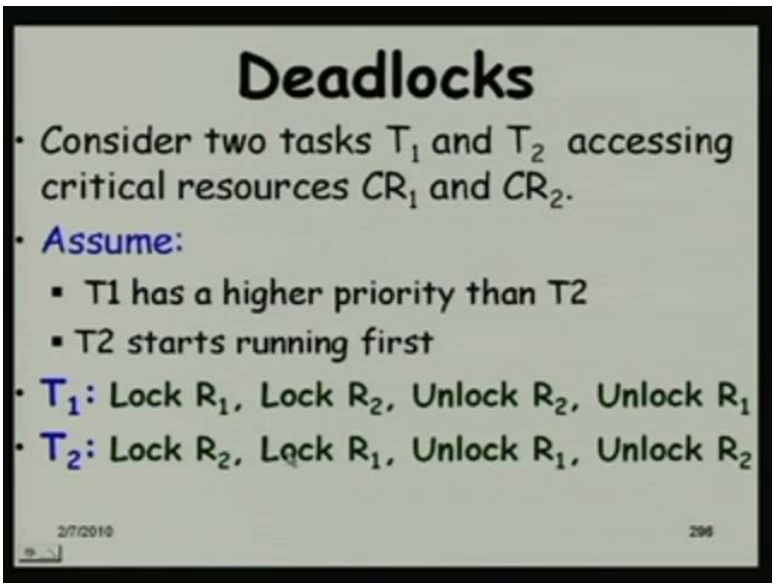
And, due to the inheritance protocol, the priority of T j, sorry T i is raised to 10, because T j is 10. So, the inheritance clause by that priority is raised, and after it completed the resource usage, its priority was changed to 5, and then, T j has got hold of the resource and it will start executing now. Is that ok?

(Refer Slide Time: 32:43)



The protocol is very simple, basic priority inheritance scheme, but there are problems with this protocol, why? This is not used as it is in the operating systems. Two main problems are deadlocks and chain blocking; it is let us see, what is chain blocking, and we will see, how deadlocks can occur here.

(Refer Slide Time: 33:08)

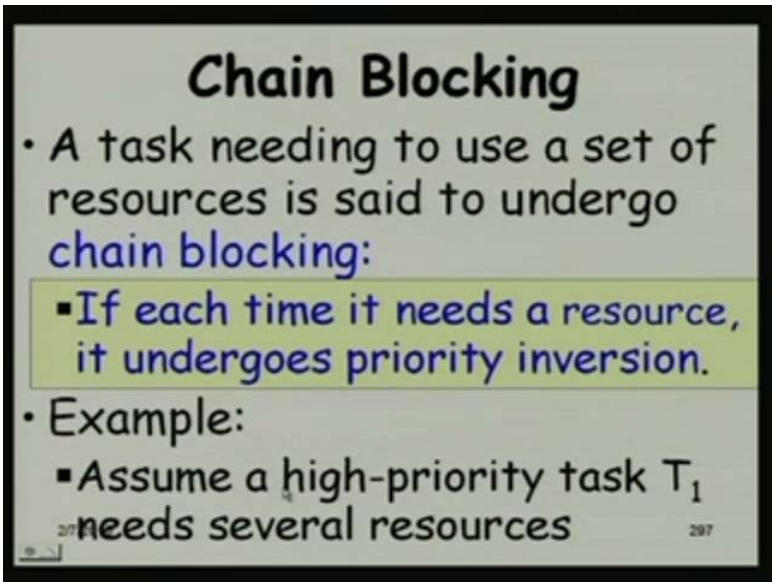


Now, first let us look at deadlock. Let us assume that, we have two tasks T 1 and T 2 that are accessing the resource CR 1 and CR 2. And let us assume that T 1 has higher priority than T 2, and T 2 starts running first.

So, T 2 is the lower priority, T 1 is the higher priority, T 2 locks, let us say, resource R 2, it started executing, locked R 2 and that time T 1 started executing, preempted T 2 from CPU users, locked R 1, and then it, after some computation, it try to lock R 2, but R 2 is the lock being held by T 2 and it will block for T 2. Right?

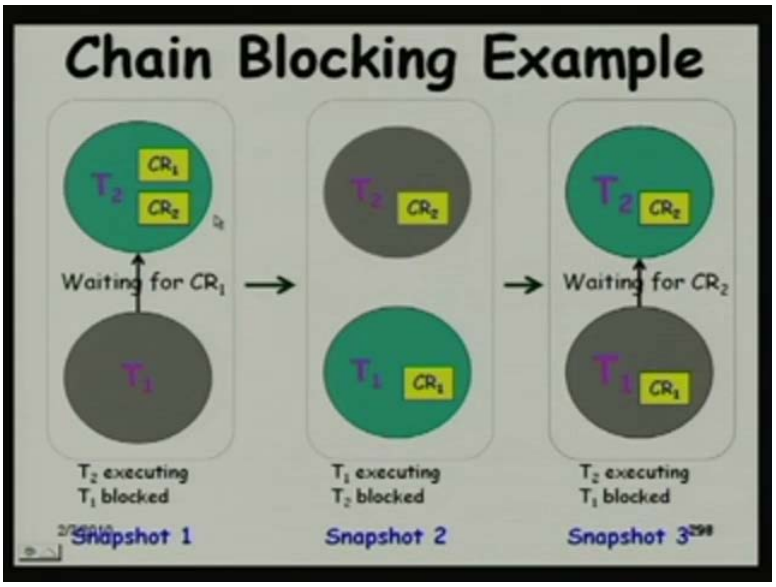
And, T 2's priority will be raised to that of T 1 through the inheritance clause, but T 2 has they have already got into deadlock right, even if priority is raised, already got into a deadlock.

(Refer Slide Time: 34:32)



So, using the simple priority inheritance scheme, deadlocks can occur. The other problem is chain blocking. The definition of chain blocking is that whenever a task needs a set of resources, if each time it needs one resource, it has to wait and undergo priority inversion, then it for set of 5 resources, it has to separately wait five times to get the resource, and we will say that it has undergone chain blocking.

(Refer Slide Time: 35:15)



Assume that a task T 1 needs several resources. So, T 1 needs let us say, CR1 and CR 2 simple example, two resources. So, after some computation, so it blocks for T 1 sorry T 2, and then, T 2 release CR 1 after executing for sometime, and T 1 could execute with CR 1, but after sometime, it needed CR 2, and then, it started blocking again right. This is example of a chain blocking.

We can also have a situation where multiple tasks are holding different resource and it is waiting for T 2 for getting CR 1, it is waiting for T 3 to get CR 3 and so on. So, chain blocking is the situation where a task to get the set of resources it needs, each time needs to block for one resource.

But, but in the previous slide however, chain blocking a problem.

I mean you are saying that, why is chain blocking a problem, is it?

Because T 2 is waiting now. So, we have the task waiting in the process (( ))

No, it is not a deadlock situation; it is the delay that is causing the problem. See, each time it it waited for some microseconds to get CR 1, then it will wait again to get CR 2, it will like wait again, just see the problem a multiple equation of the delay for each individual resource, undesirable situation; this can cause problem right; chain blocking

can cause problem. Deadlock is definitely unacceptable, but chain blocking also can cause problem, if the tasks have tight deadlines.

Sir, this principle just converts the unbounded priority inversion to a normal priority inversion.

Exactly exactly.

(( ))

See see, that is a interesting observation. See, we have not eliminated a simple priority inversion; a simple priority inversion can never be eliminated, but what we have said is that a simple priority inversion can be tolerated by careful programming, by splitting the access to a critical resource of a low priority task into small chunks. So, simple priority inversion, we will see all these protocols, which are refinement of the basic inheritance scheme and which are used in the commercial operating systems. They cannot eliminate a simple priority inversion, but what they are targeted is to eliminate unbounded priority inversions.

So, I hope whatever we discussed so far is clear right. So, let us just try to answer a few simple questions.

Sir,

Yes.

Sir (( )) when does the a a process has blocked for some resource.

Yes.

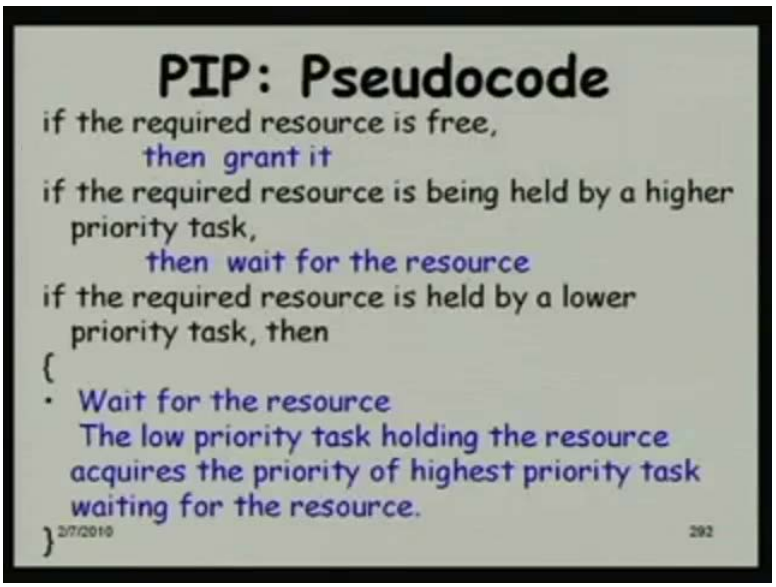
And another higher priority process comes and it also blocks for the same resource.

Yes.

So, so, the priority inheritance takes place during the suspended state or when one of the lower priority task is executing; at what time does the priority inheritance takes place?

So, let us, the question is that when does the priority inheritance takes place? So, let us see the basic pseudo code we had seen, let me just get back there. So, that this question we can answer.

(Refer Slide Time: 38:59)



See, this is the pseudo code that we saw here. Whenever there is a request for a resource, this pseudo code is applied, any task is requesting for a resource; if it is resource is free, no problem, grant it; if the resource is held by a higher priority task, no problem, just wait for the … keep on waiting for the resource; if it is held by a lower priority task, then it waits for the resource, first thing and also the priority inheritance clause is applied.

(( )). So, in in this scenario the lower priority task is running on the processor.

Yes.

So, what if the lower priority task is itself not running and the higher priority task is waiting for resource, which is held by the lower priority task, which is not running on the CPU. So, in that case, with the lower priority task inherit the (( )) the priority of the higher priority task.

Of course; see this clause, does not say that which task is running right. It saying that, if the resource is held by the lower priority task, then apply this, does not say that whether the lower priority task is running or not, nothing, as soon as there is.

(( ))

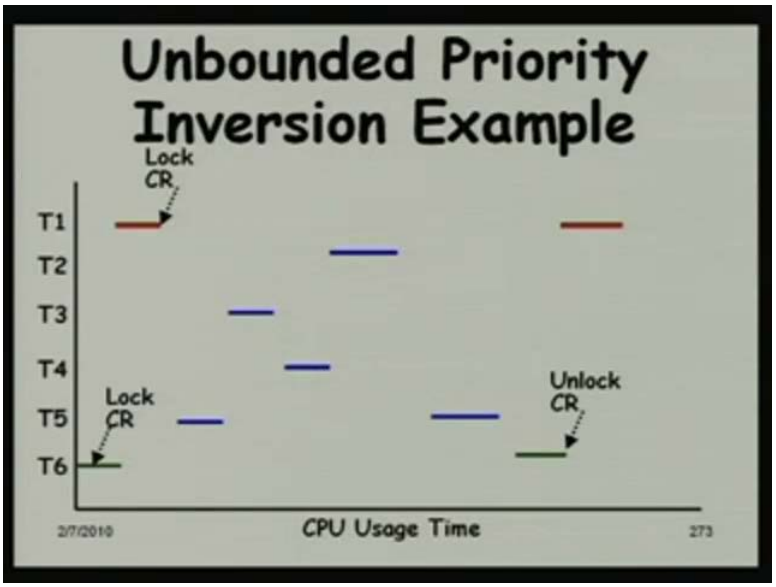
Sorry.

If this pseudo code is getting executed just current process ((is)) running.

Need not be; like he says the current process may not be running, say, the lowest priority task holding the resource may not be running; it is blocked by another or may be the higher priority task was waiting, sorry it was running and then, it has requested the resource. So, the lower priority task… like the example that we had shown here just see that…

Where is that example?

(Refer Time Slide: 40:56)



This is the example; see here. See here, the lower priority task was running for some time, and then, it was preempted by the higher priority task, and while the higher priority task was running, it requested the resource. So, very likely the lower priority task will not be running when the resource is requested.

(( ))

Yes.

At that time no no process will be running right at that ((moment))

Ok.

The scheduler will be executing (( ))

No, the one that was running before it was requested. So, may be that was his question, but the thing is that, whenever request, is resource is requested, lower priority task will definitely not be running, because the request is being by some other task, is it not? Lower priority task will not definitely be running.

Yes please.

Sir, you are saying that (( )).

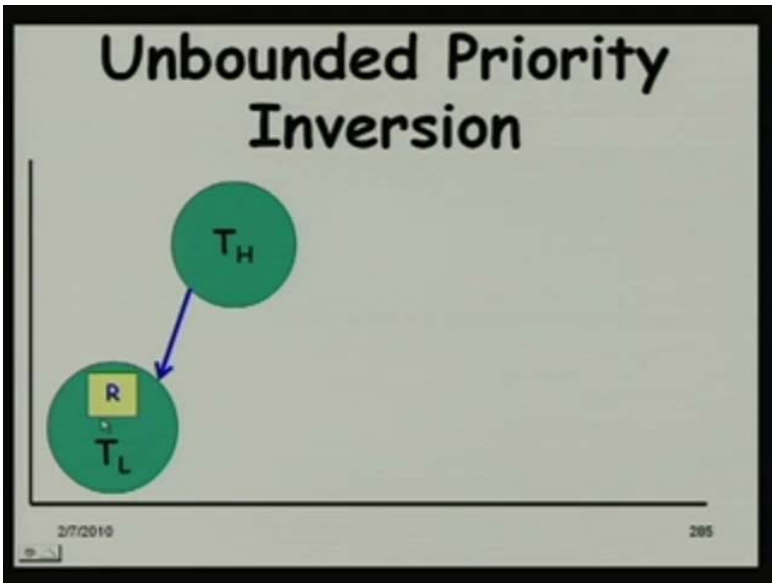
Yes.

Then there might be a a list of many ((tasks)) which are (( )) holding on separate resources.

Yes.

So, on that case a task will have multiple priorities or just one single priority (( ))

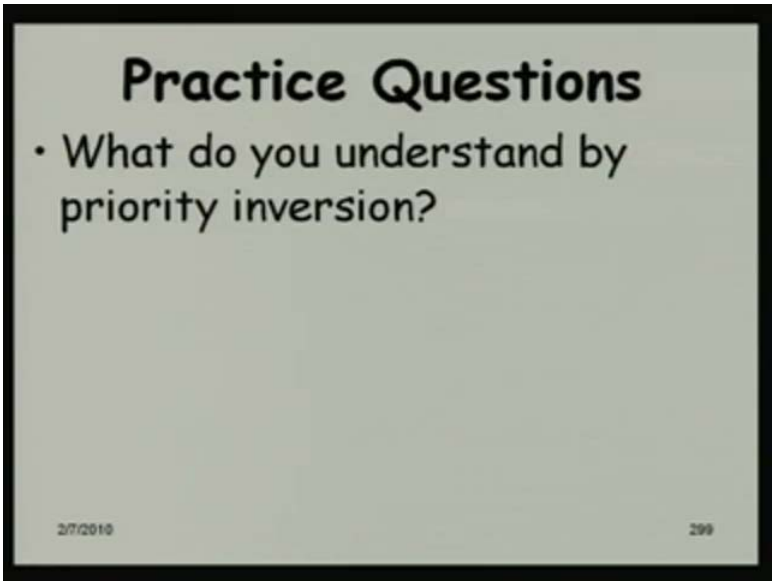
(Refer Slide Time: 42:18)



You mean, one task having multiple priority.

No, no no; see a task will get the highest priority of all the tasks that are waiting for various resources, that is being held by it. So, the priority inheritance protocol is simple, but has problem that is why it is not used as it is. Now, let us look at some very simple questions based on the priority - simple priority - inheritance protocol. So, let me get back to the point where just left it; this we had seen yeah.

(Refer Slide Time: 43:09)

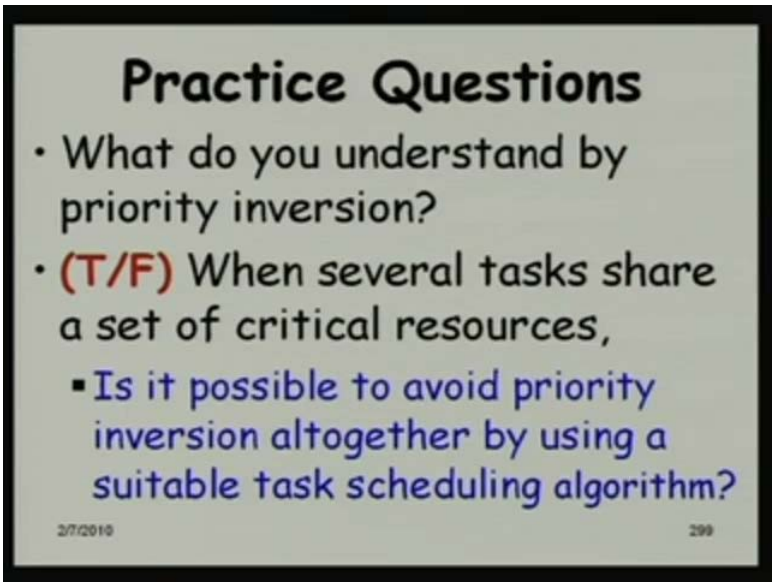


So, priority inversion, if somebody asks you - what is priority inversion? - what do you say?

(( )) situation where the low higher priority task waits for the lower priority task in case of a resource (( )) lower priority task.

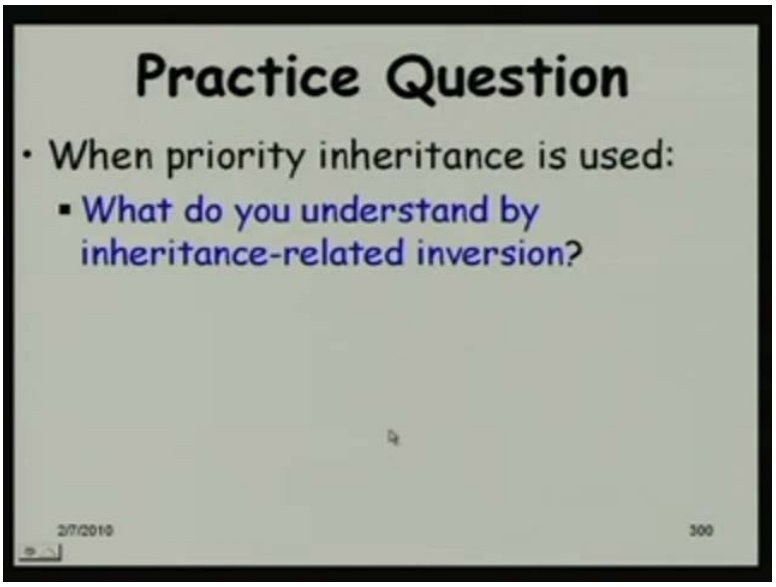
So, lower priority task keeps on executing and a higher priority task is waiting.

(Refer Slide Time: 43:30)



(( )) So, that is a priority inversion. InversionNow, just tell me, whether this sentence is true or false? See, when several tasks share a set of critical resources, it is possible to avoid priority inversions altogether using a suitable scheme - like he was asking, you know, he was he pointed out. So, no, because that he has already pointed out, that a simple priority inversion is very difficult to prevent.

(Refer Slide Time: 44:01)



Now, when a priority inheritance is used, what is inheritance related inversion?

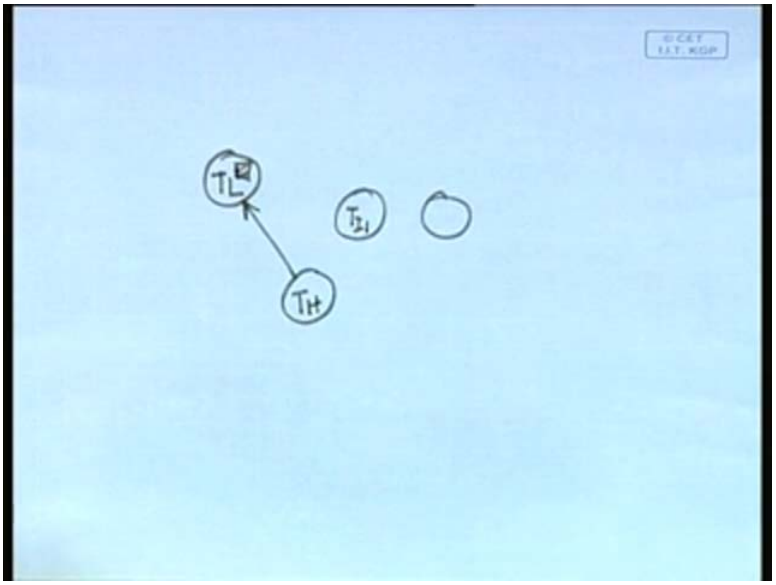
It is may be, when we higher the…. When we increase the priority of a lower priority task, some intermediate tasks between the higher and lower are not be into (( )) processer.

Exactly.

(( )). So, it is like they are, its its like a priority inversion, because typically the priority of that (( )).

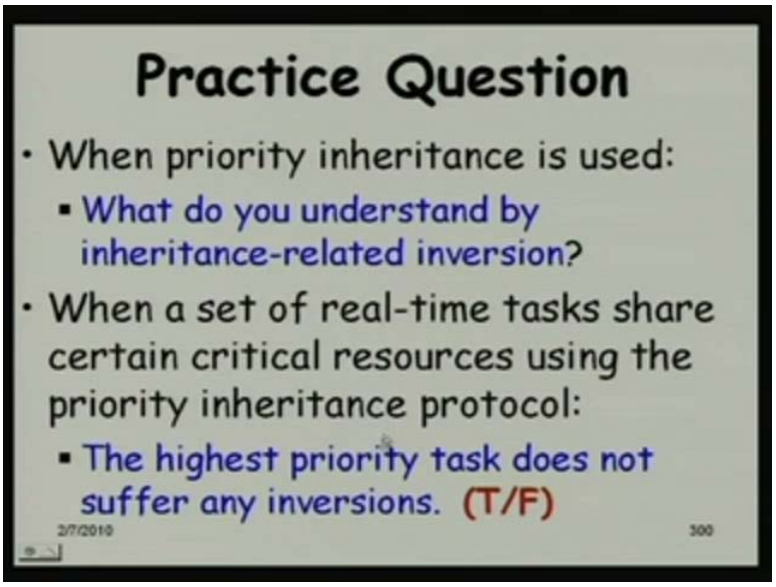
Yes yes. So, what he said is correct.

(Refer Slide Time: 44:45)



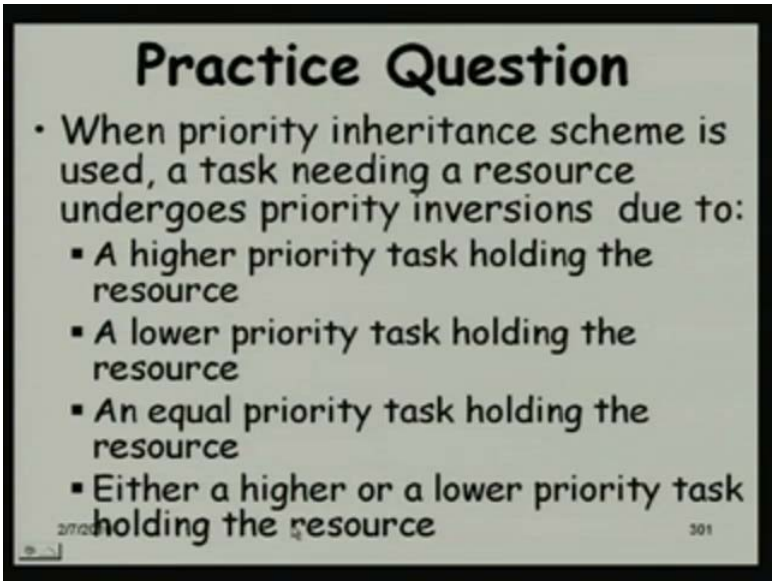
See, he says that see, we had a low priority task T L, and a high priority task is waiting for some resource held by it, and as a result the T L inherits the priority of the high priority task. But, there may be other tasks - intermediate tasks - which do not need this resource, but still they cannot proceed with their computation, they are undergoing inversion, they should have completed, because they do not need this resource. So, that is an inheritance related inversion.

(Refer Slide Time: 45:19)



Now, let us look at the next question - when a set of real-time tasks share critical resources using the priority inheritance protocol, the highest priority task does not suffer any inversions; false right. What about lower priority task? When a set of tasks share a certain critical resources using priority inheritance protocol, the lowest priority task does not suffer any inversion. Will it be correct? yeah Lower priority task, by its by definition itself, it cannot suffer any inversion.

(Refer Slide Time: 46:03)



Now, when a priority inheritance scheme is used, a task needing a resource undergoes a priority inversion, due to a higher priority task holding the resource, due to a lower priority task holding the resource, due to an equal priority task holding the resource, either a higher or a lower priority task holding the resource.

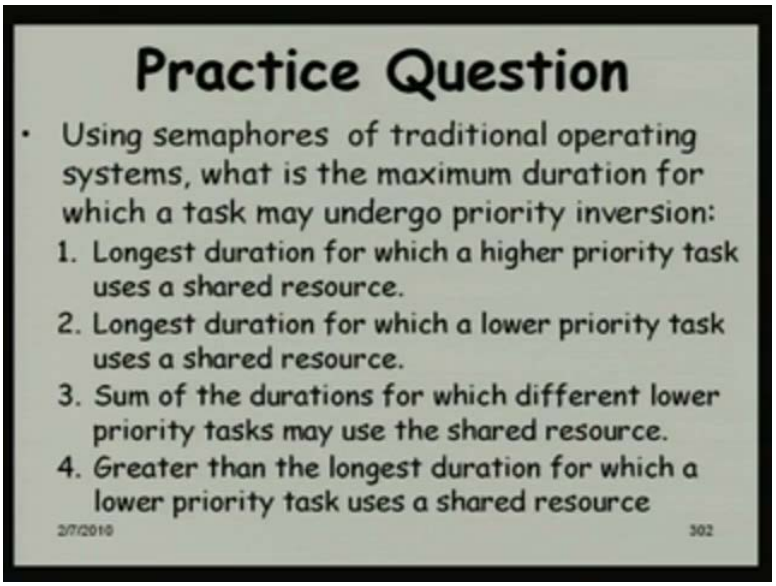
Lower priority task.

What about you? What is your answer? Which one is the correct option here? Priority inheritance scheme is used and a task undergoes a inversion in this. When a higher priority task is holding the resource, a lower priority task is already holding the resource, equal priority task is holding the resource and either a higher or a lower priority task is holding the resource; yeah exactly.

When a higher priority task is holding the resource, inversion cannot occur; and a equal priority task holding the resource also, inversion does not occur, because it is at least the same priority; and this also, either a higher or lower is not true, because lower priority task, a higher priority task cannot cause inversion. So, it is only this one - second choice.

So now let us see, what are the improvements that were proposed to the basic inheritance scheme, to make it more usable, because when deadlocks and chain blocking is there, I think, there is one more question.

(Refer Slide Time: 47:36)



Using semaphores of traditional operating system, what is the maximum duration for which a task may undergo priority inversion? So, that is, we are trying to use the semaphore of the traditional operating systems and you are trying to find out the maximum duration for which a task will undergo inversion. Is it equal to the longest duration for which a higher priority task uses the resource; the longest duration, for which a lower priority task uses the resource; sum of the durations for which different lower priority tasks may use the shared resource or greater than the longest duration for which a lower priority task uses the shared resource.

(( ))

So, what about …

Because as soon as the lower priority releases the resource.

Yeah.

Then what is the case of (( )) again complete (( ))

So, they are thinking the second option is correct, but what about you here?

(( )) the third one (( ))

Sum of durations for which different lower priority task may use the shared resource. Is it?

(( ))

The time for which, they use the resource. Anybody has any other answers?

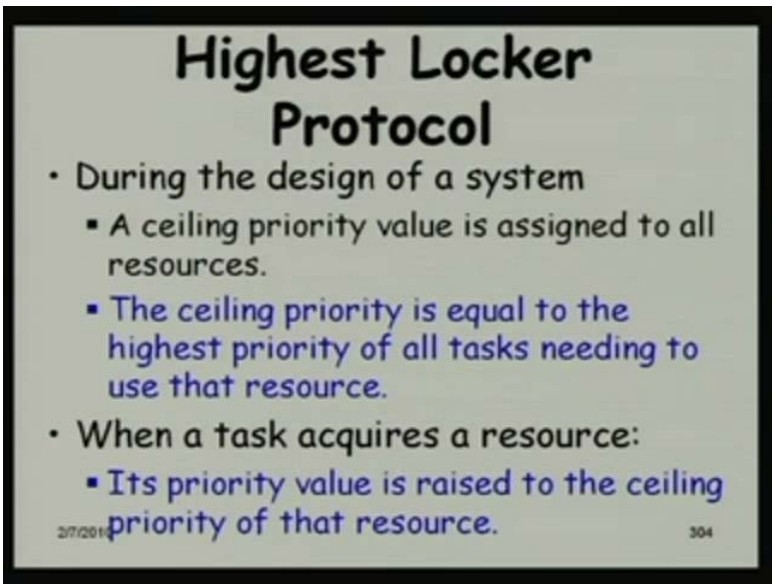
(( ))

The answer is actually 4; the answer is that it is greater than the longest duration for which the lower priority task uses a shared resource, because here even when they are not using the resource, they are they are just holding it, still there may be intermediate priority task, which are not requiring the resource they have preempted it right. So, only thing we can say is that, it is longer than the maximum duration for which a task holds

the resource; that is all, we can say. We cannot say that it is sum of the durations for different tasks need the resource.

So, now, let us see some of the improvements that were proposed to overcome the problems of the basic priority inheritance scheme.

(Refer Slide Time: 49:58)

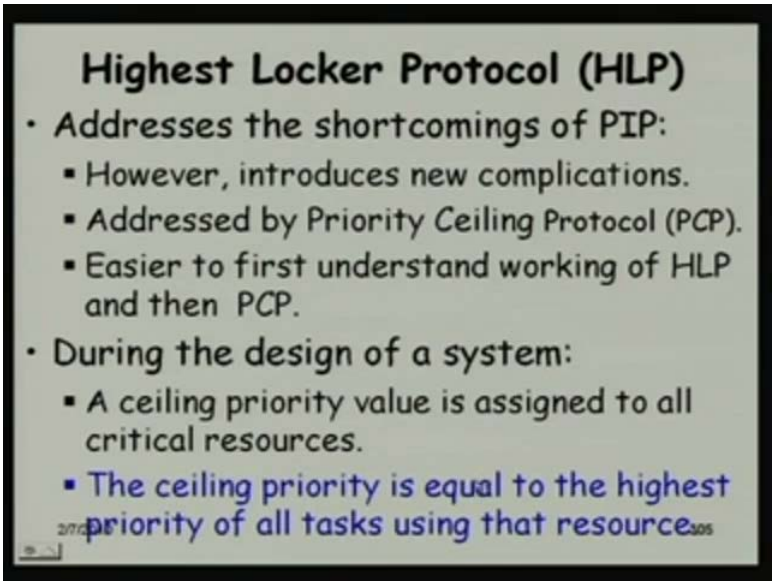


So, we first look at the highest locker protocol; this is again a solution, which is a straight extension of the inheritance protocol, but itself has problem. It solves some of the problems, I mean, the main problems that the basic inheritance scheme it overcomes those, but it creates new problems.

Now, we will see the priority ceiling protocol after this, we will see that, that is the one which is actually used - the priority ceiling protocol, but to be able to understand the priority ceiling protocol we need to discuss about the highest locker protocol, because unless we look at the highest locker protocol, ceiling protocol may not make too much sense. So, first let us look at the highest locker protocol. In the highest locker protocol the idea is that a ceiling priority is assigned to every resource; look at all the resources, find out who are the tasks using it, and then assign a ceiling priority to the resource, and the ceiling priority is computed as equal to the highest priority of all tasks needing that resource.

If we find that three tasks are needing that resource, we will set the ceiling priority for that resource as the highest of all the three priority task priorities. We will take some example and also explain. Now, once we have computed the ceiling priority, the rule is simple here, whenever a task acquires a resource - any task - the rule is straightforward here, whenever a task acquires a resource, raise its priority to that of the ceiling priority; simple rule here. So, whenever a task acquires a resource, its priority is raised to the ceiling priority of the resource

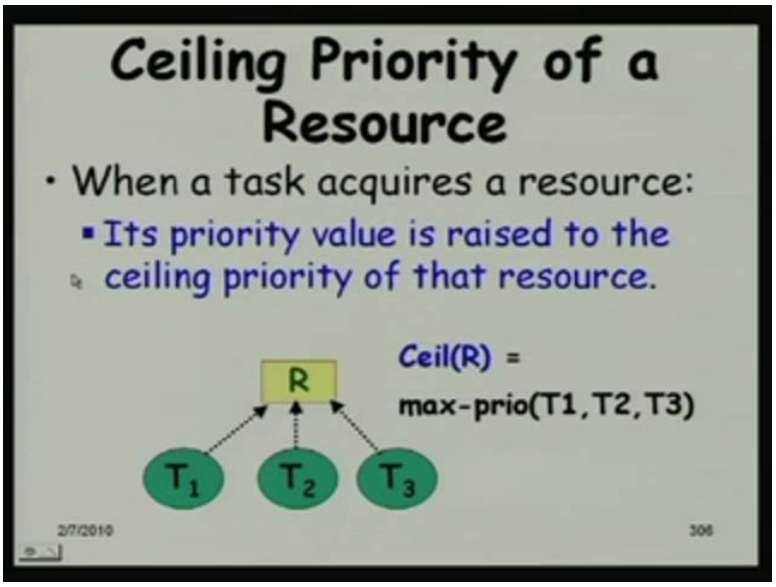
(Refer Slide Time: 51:49)



It addresses all the shortcomings of the basic inheritance scheme, the priority inheritance scheme, but we will see that introduces new complications, which will be addressed by the ceiling protocol. But if we understand the highest locker protocol, then understanding the priority ceiling protocol will be easy, because priority ceiling has many clauses and straightaway understanding it might appear bit unnatural and difficult, we should understand highest locker protocol, find out its problems and then understanding why something is done in priority ceiling protocol will be easy.

So, here during the design of the system, a ceiling priority is assigned to all critical resources, that we had seen and the ceiling priority is equal to the highest priority of all tasks using that resource.

(Refer Slide Time: 52:45)



So, if R is the resource, for every resource R, we will find out which are the tasks using the, which might use the resource, that assumed to be known, that which task will use which resource, which data structure, which device, etcetera be used by one task, and based on that we know the priorities of these tasks, and we will set the ceiling priority of the resource R is the maximum priority of all the three tasks.

So, let us stop here, we will continue from this point in the next class. So, any questions etcetera, make it please keep it ready, you will you can ask during the next class.

Fine, thank you.