Real-Time Systems

Prof. Dr. Rajib Mall

Department of Computer Science and Engineering

Indian Institute of Technology, Kharagpur

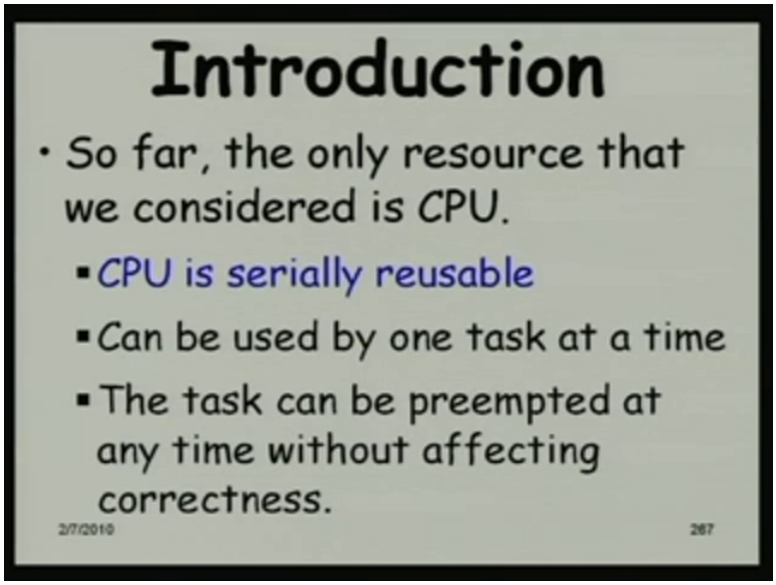
Module No. # 01

Lecture No. # 13

Resource Sharing among Real-Time Tasks

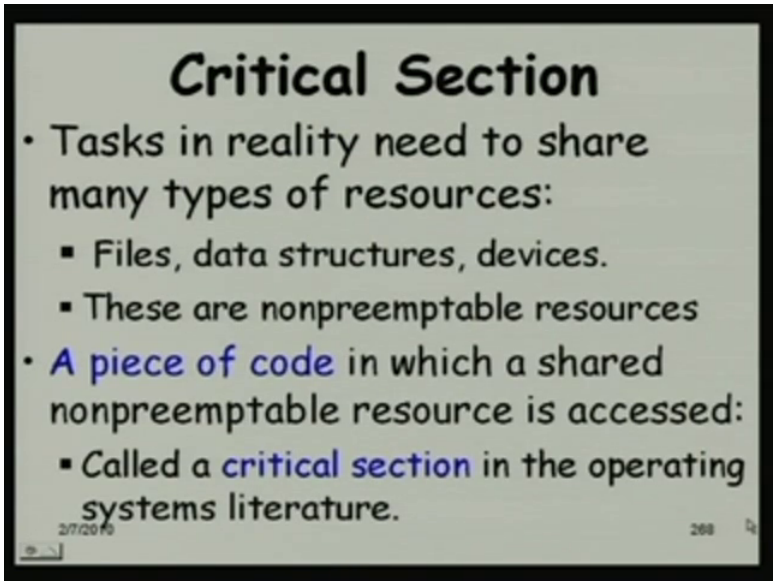
Good morning, let us get started. Today, we will discuss about resource sharing among real-time tasks because we have considered the situation where tasks share no resource, but that was unrealistic. In practical situation tasks will be sharing some resources. We will just discuss about the issues that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

(Refer Slide Time: 01:01)



We considered only CPU as the shared resource among tasks. CPU is a serially reusable resource. It is used by only one task at a time, like the other resources memory, devices, etc.

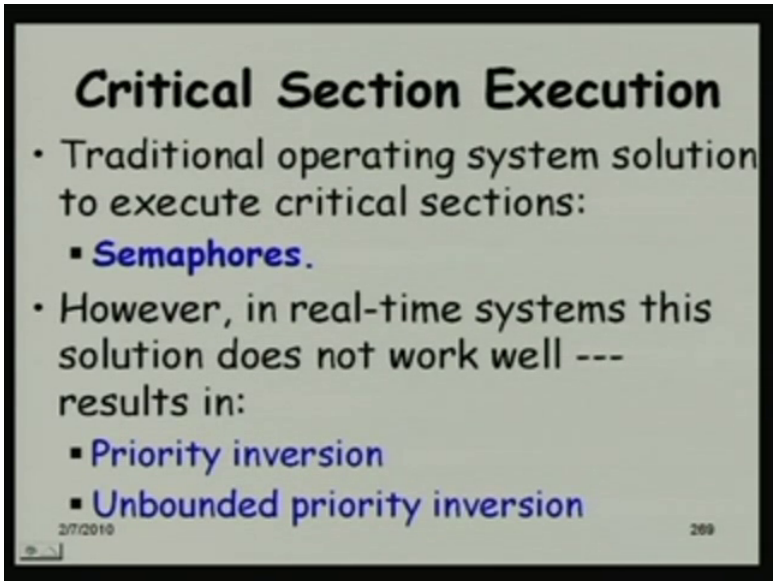
(Refer Slide Time: 01:42)



The tasks need to share many other types of resources other than the CPU. For example, files, data structures, devices etc. Many of these resources are not used in a preemptable mode.

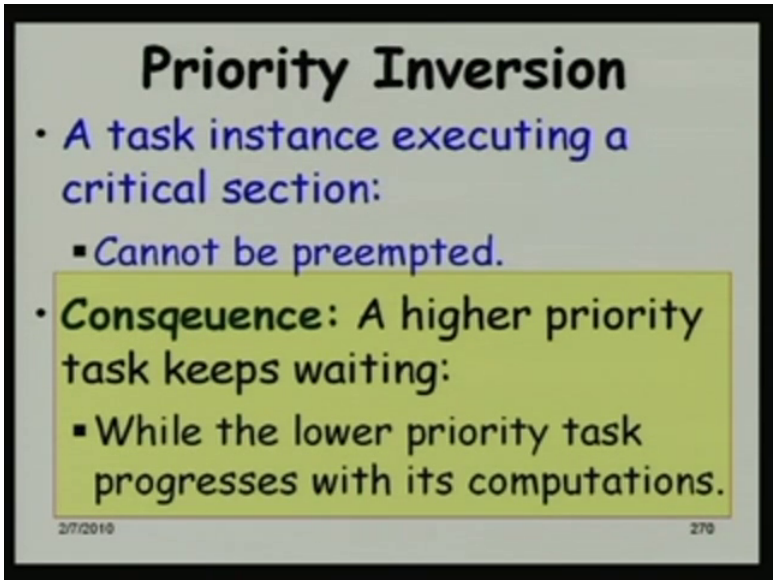
The part of the code in which a non preemptable resource is accessed is called critical section in the operating system literature. It is the part of the code where once the control is entered the execution starts and it should complete without releasing the resource. Sometimes we loosely talk of critical 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 semaphores are not the solution and it will create problems. The problems are known as priority inversion and unbounded priority inversion.

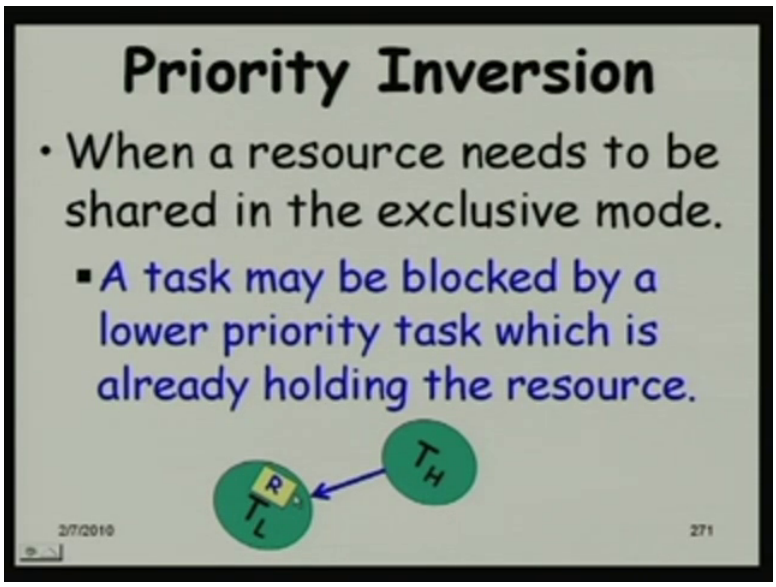
(Refer Slide Time: 03:36)



In priority inversion, when a task is executing in its critical section, it should not be preempted from resource uses because it will make the resource inconsistent. These things we know from the first course on operating system.

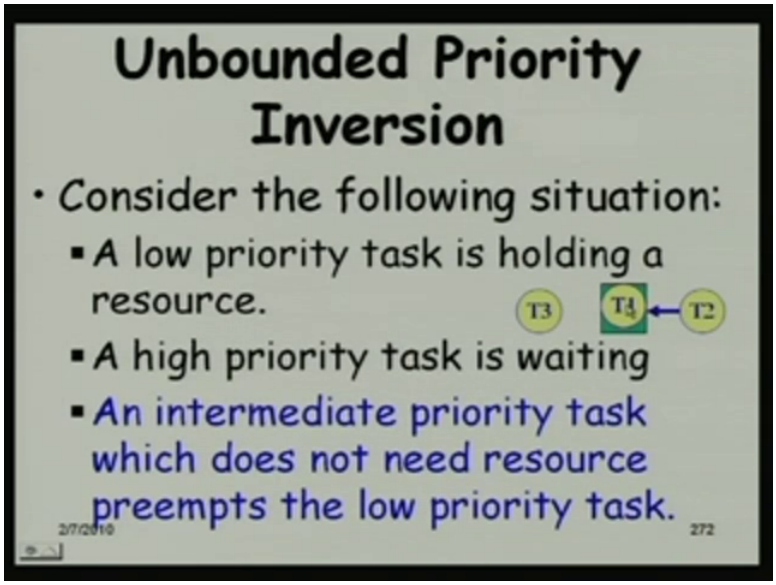
The consequence in a real-time application is that if a task cannot be preempted (a lower priority task), then higher priority task will keep on waiting. It will create a situation when the real lower priority task is making progress with its computation, and higher priority task is waiting. It is against the basic assumption that we had in the scheduling algorithms.

(Refer Slide Time: 04:59)



When a task is executing and accessed the resource in the exclusive mode, then it can block a higher priority task. For example, 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. Until the lower priority task completes its execution, the resource will not be available to higher priority task.

(Refer Slide Time: 05:51)

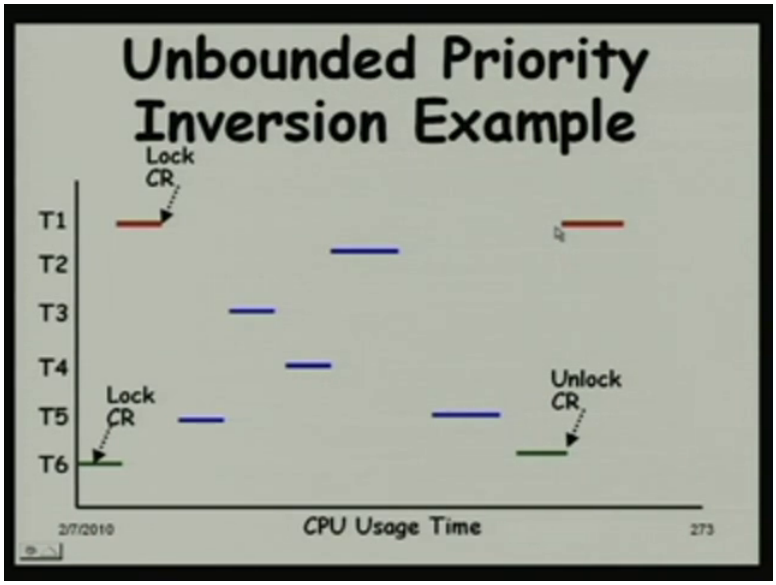


A simple priority inversion can be handled. But the one that is very difficult to handle is unbounded priority inversion.

Consider the situation, that there is a resource and a task was executing. After sometime, the task acquired the resource, and meanwhile, a higher priority task was released. The higher priority task is waiting for the resource. This is the simple priority inversion situation. But if some tasks, which do not need the resource, that are of higher priority than T1, but lower priority than T2, also get ready and start executing, because they do not need the resource, they will preempt; they will not wait for T1.

We have a task T3, which does not need the resource and it preempts T1. T2 will wait for T3 to complete, and then T1 to complete but they may not be one intermediate priority task, there might be multiple of them, and as a result, T2 keeps waiting for all these intermediate priority tasks to complete, so that T1 will finally execute, and release the resource, and then T2 will proceed; but that may be too late.

(Refer Time Slide: 07:45)

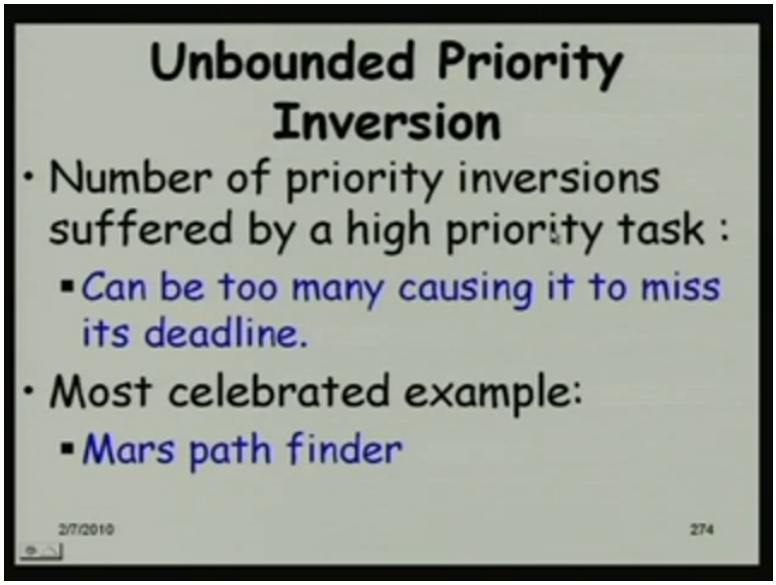


This is an example of unbounded priority inversion. We have six tasks and T1 has the highest priority and T6 has lowest priority.

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

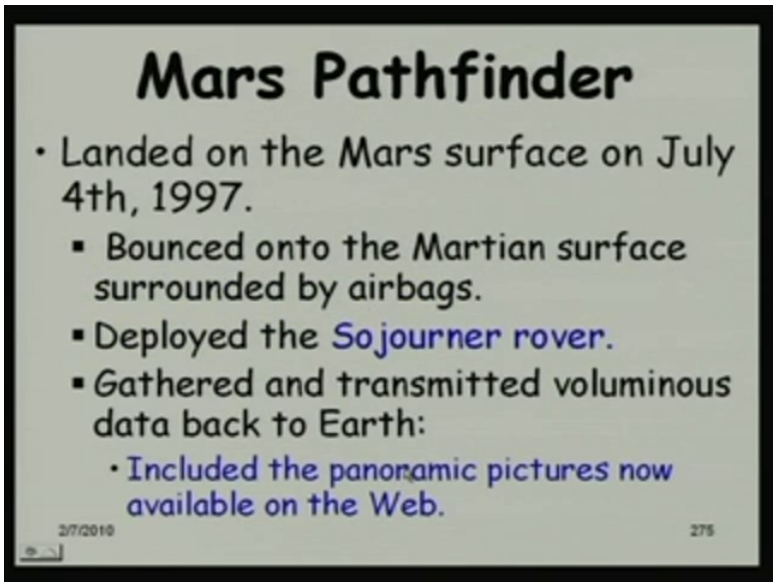
T6 will execute, but in meanwhile other tasks have become ready and we are just waiting. Therefore, T6 does not get the CPU but T5 get it and executes for some time. Meanwhile T6 is waiting is executing, then, T4 is executing, T2 executes for some time, T5 again gets back the CPU, starts executing, and after all this, T6 gets to use the CPU and completes resource usage, unlocks the resource, and then only, the task T1 can start executing.

(Refer Slide Time: 09:56)



The number of priority inversion suffered by a high priority task can be too many and is likely to miss deadline, whenever unbounded priority situations occur. There are many situations where systems have malfunctioned, because of the priority inversion problem. The most celebrated example is the Mars Pathfinder.

(Refer Slide Time: 10:32)



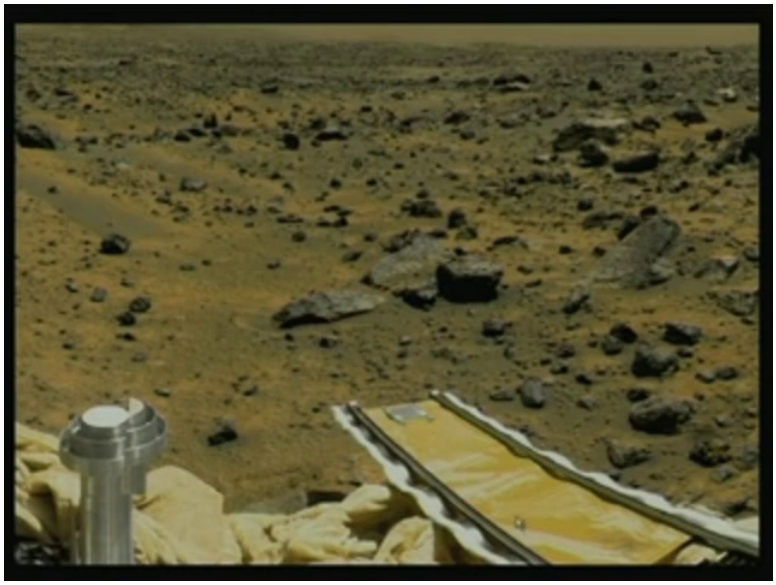
The pathfinder was sent to the Mars, and it landed on 4th July 1997, and it bounced onto the Mars surface by airbags, so that, it will not get damaged. It had a vehicle which will roam around the Mars surface called as the Sojourner.

The Sojourner rover is the one, which was released to roam around the moon surface. The Sojourner roamed around and gathered data and starts transmitting the voluminous data to the Earth.

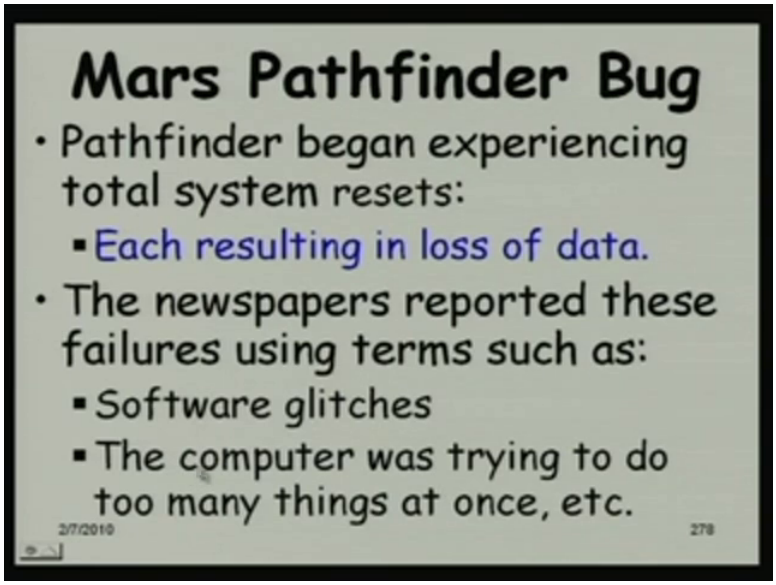
(Refer Slide Time: 11:35)

This is one of the pictures. It contains the rover vehicle, the Sojourner etc

(Refer Slide Time: 11:55)

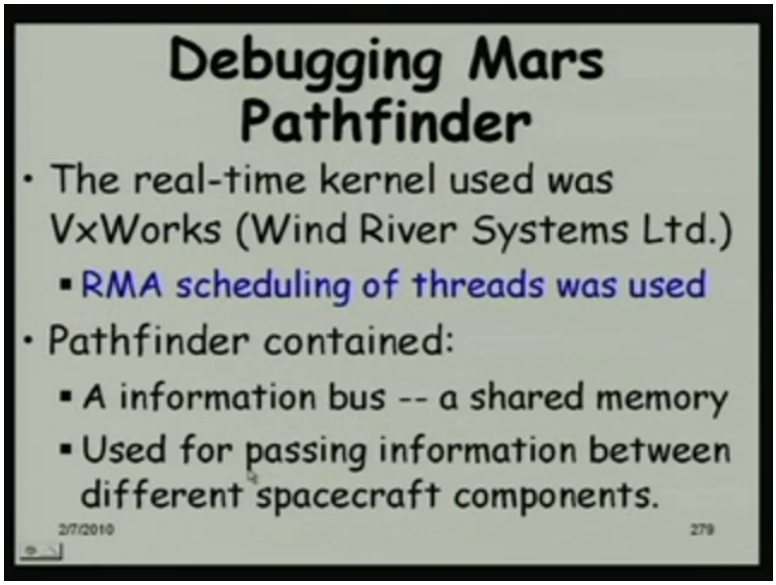


(Refer Slide Time: 12:06)



But soon, it started experiencing total system resets. Each time the system reset, huge amount of data was lost and it appeared in all newspapers. They wrote like Mars Pathfinder is getting into problem, and; some said software glitches have occurred in the Mars Pathfinder, it is not taking commands, and some papers, they said that the computer was trying to do too many things at once and it was crashing etc.

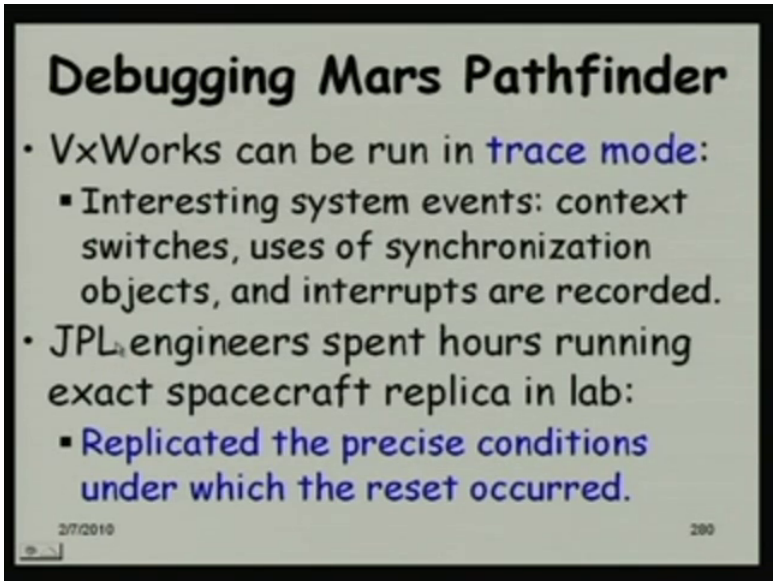
(Refer Slide Time: 12:48)



The pathfinder was using a real-time kernel - the VxWorks - the vendor is Wind River Systems, and RMA scheduling of the threads was used. The pathfinder contained an

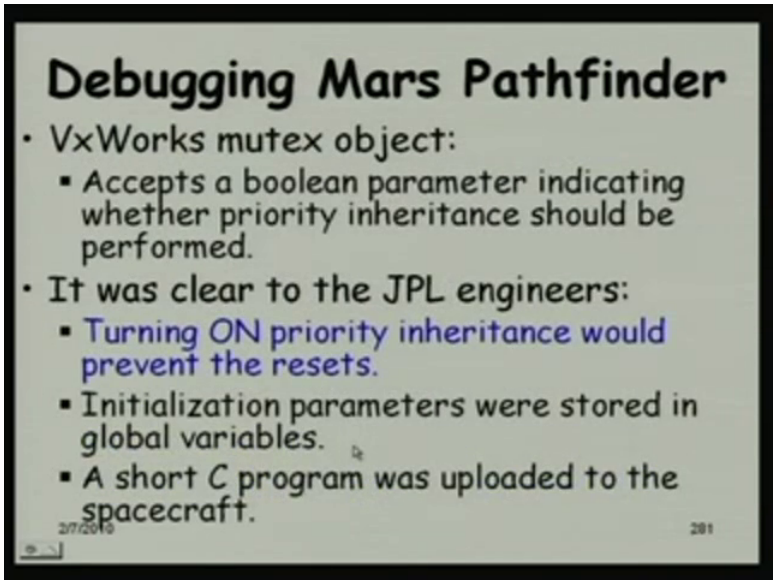
information bus, a communication channel, which was a shared memory to communicate among various tasks that execute on the Pathfinder and the rover.

(Refer Slide Time: 13:34)



Once it started resetting itself, the VxWorks provides a facility to run it in the trace mode. 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. The Jet Propulsion Laboratory engineers 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.

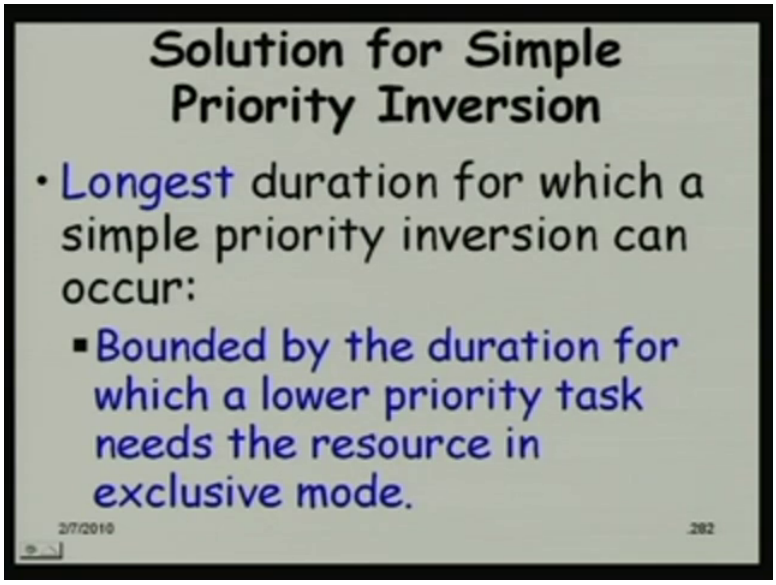
(Refer Slide Time: 14:37)



After spending lots of hours, 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 priority inheritance would be enabled or not. 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.

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.

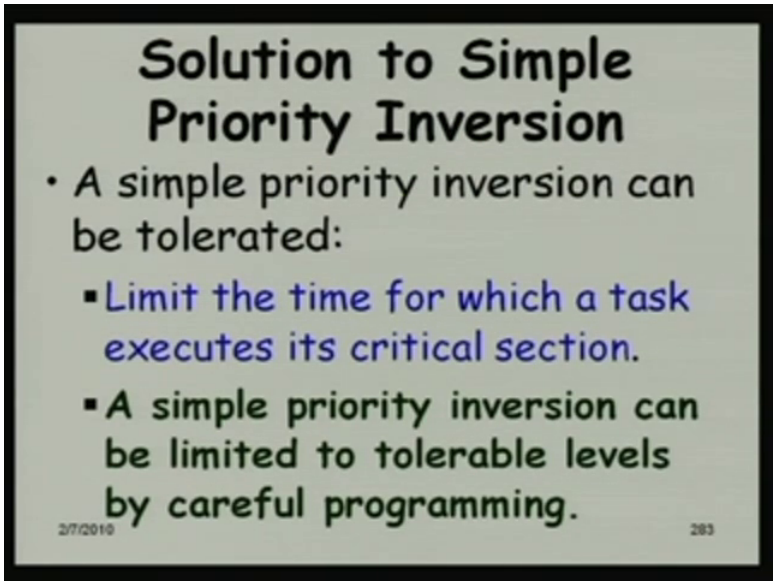
(Refer Slide Time: 15:49)



When a simple priority inversion occurs and the high priority task is waiting then what can be done? 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.

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.

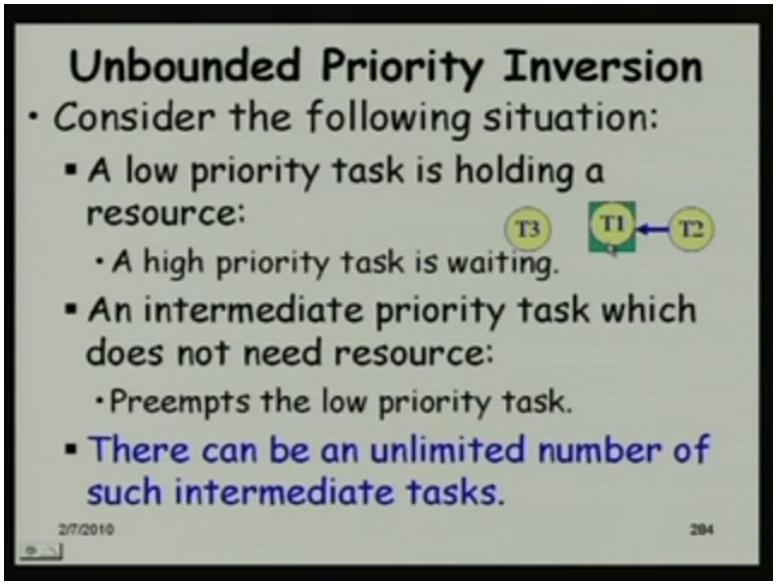
(Refer Slide Time: 17:16)



If simple priority inversions were to be handled then we have to reduce the maximum duration for which a task will need a critical resource.

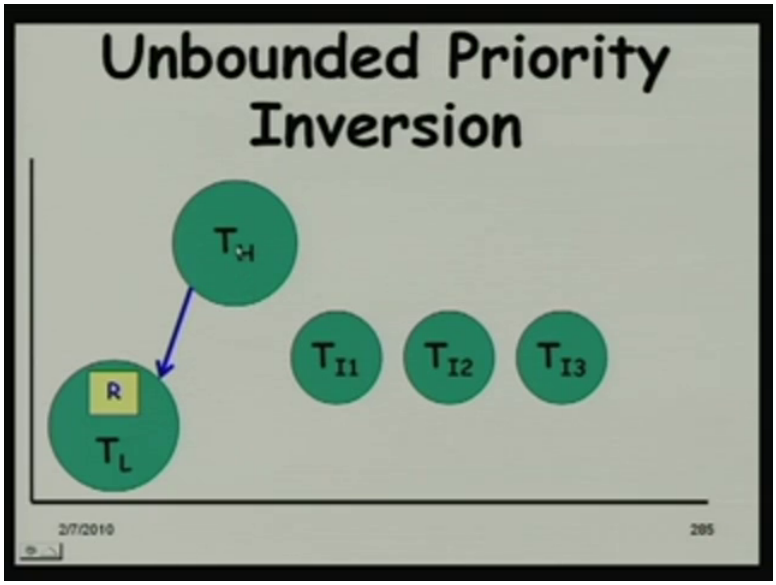
How to do that? Limit the time for which a task executes in critical section, by careful programming or by splitting its access to a critical resource across multiple instances, do part of it, release it, and then, acquire it. How unbounded priority inversion can be handled?

(Refer Slide Time: 18:10)



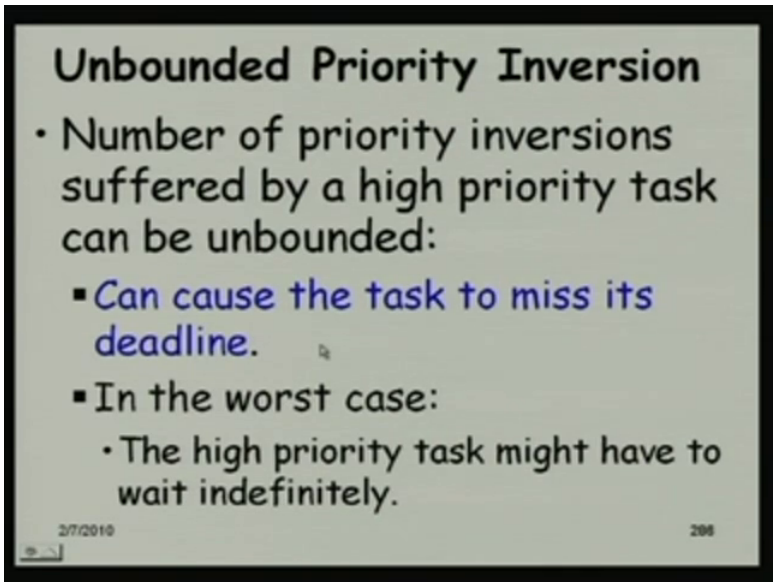
A low priority task is holding a resource and a high priority task is waiting and an intermediate priority task has come in, preempted T1, started executing and T2 is waiting all through.

(Refer Slide Time: 18:37)



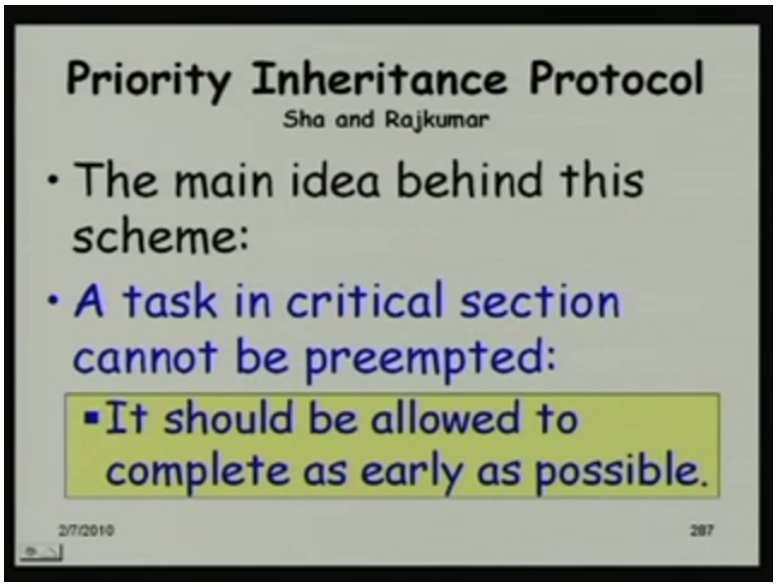
And there are many instances of those intermediate priority tasks, which are keeping on coming and preventing the high priority task from acquiring the resource, because low priority task is not able to complete its resource uses.

(Refer Slide Time: 19:35)



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

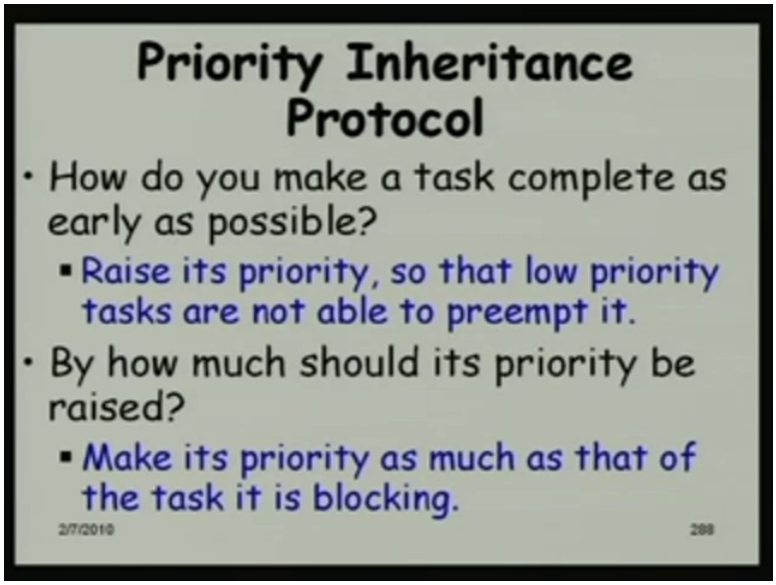
(Refer Slide Time: 19:48)



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.

Sha and Rajkumar said that a low priority task once it has acquired a resource cannot be preempted only thing that can be done is that to 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.

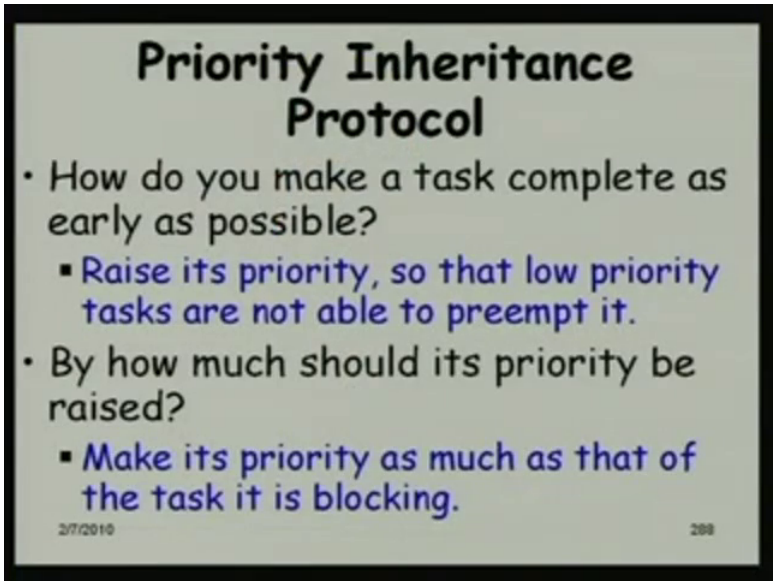
(Refer Slide Time: 22:52)



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

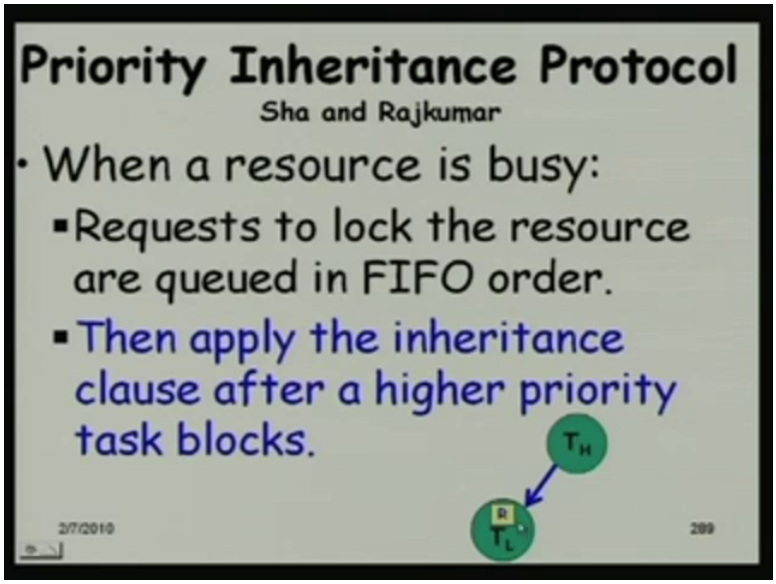
We can raise its priority so that the low priority task is not able to preempt it. The priority of the task should be raised to a priority, which is as much as the priority of the task that is waiting for the resource because if we raise it to anywhere lower than that, then we will have intermediate priority tasks which will preempt it.

(Refer Slide Time: 24:50)



The priority should be raised to as much as that of the task that is blocking it. The priority of the task that is holding the resource should be raised by as much as the task that is waiting for the resource (the highest priority task).

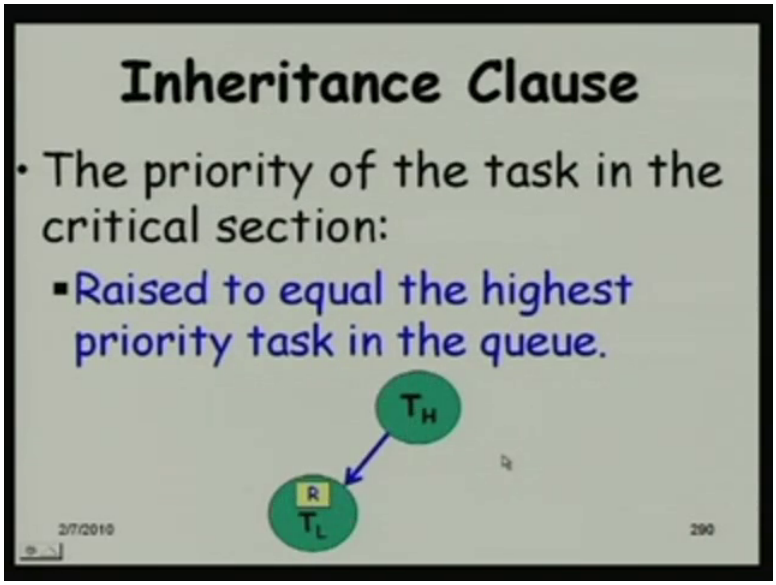
(Refer Slide Time: 25:11)



When a resource is busy, then the requests to lock the resource are queued in FIFO order. Multiple tasks might come in and wait for the resource, right we just shown it just one task has waiting, but it is likely that multiple tasks might be waiting for the resource.

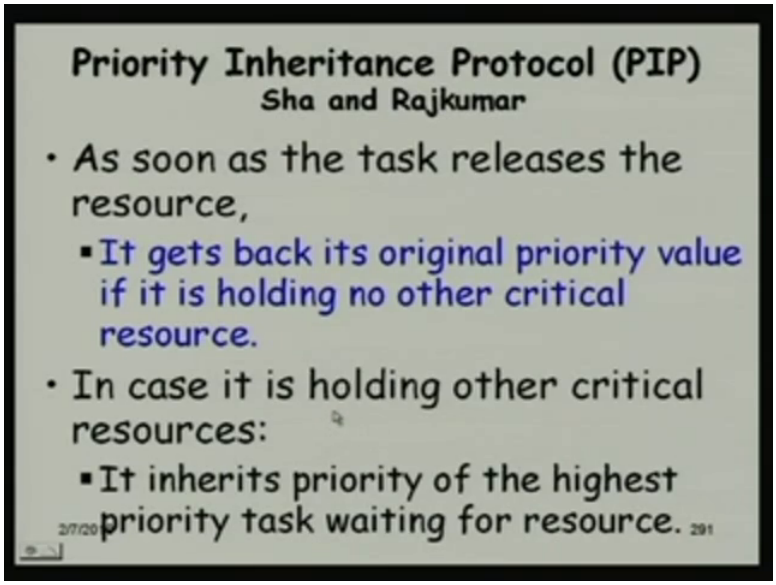
May be a low priority task T_L was using the resource R, and meanwhile, some other task T_i requested the resource. T_i started executing, and then requested, and it blocked. Then, T_L was executing again, and T_H a higher priority task started executing, because that time it did not need resource, and 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. 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. As soon as the task releases the resource, it should get back its old priority.

(Refer Slide Time: 26:37)



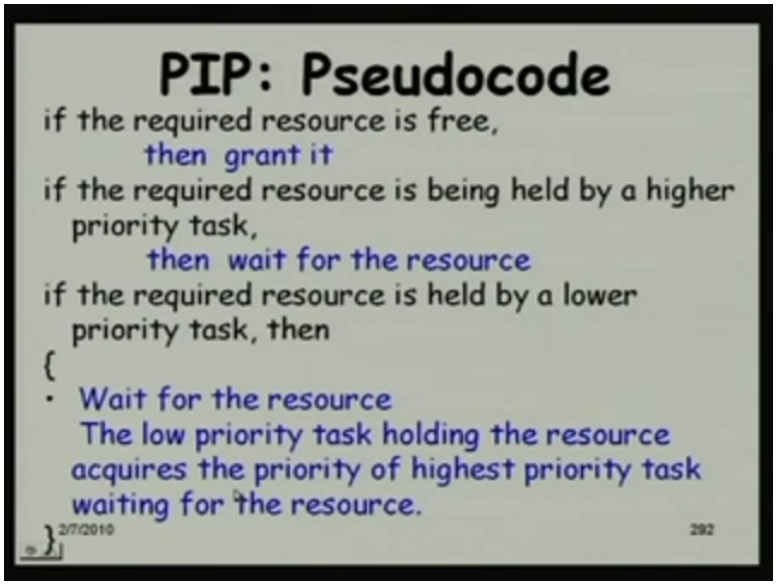
Many operating systems for simplicity let task wait in a queue, and then, use a round robin scheduling among them.

(Refer Slide Time: 28:42)



Once the low priority task has completed its resource uses, then it gets back its old priority. Also, if it is holding some other resource, then it releases that.

(Refer Slide Time: 29:28)

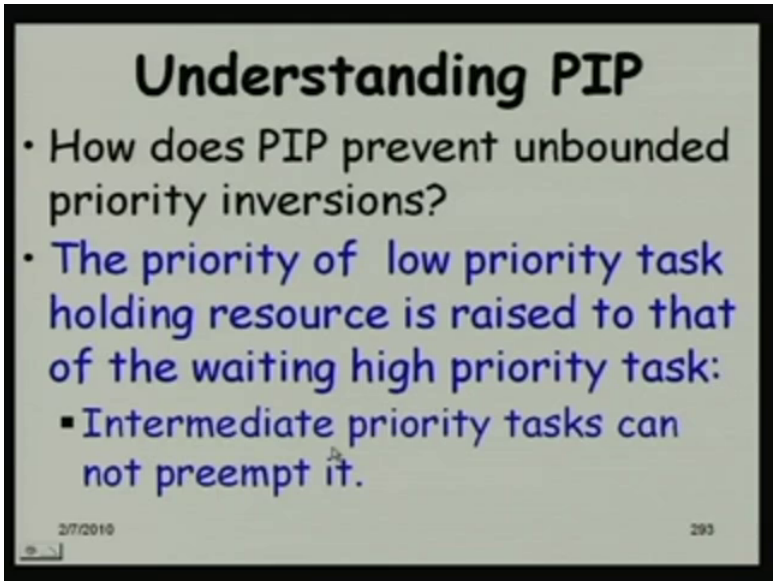


The basic pseudo code of priority inversion protocol is given. Whenever, there is a resource is available and the request is there, grant it, you do not check the priority whether it is a low priority or high priority. As long as the resource is available, it is granted. If the required resource is held by a higher priority task, then it should wait for the resource and nothing needs to be done, because this not a priority inversion situation.

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. 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.

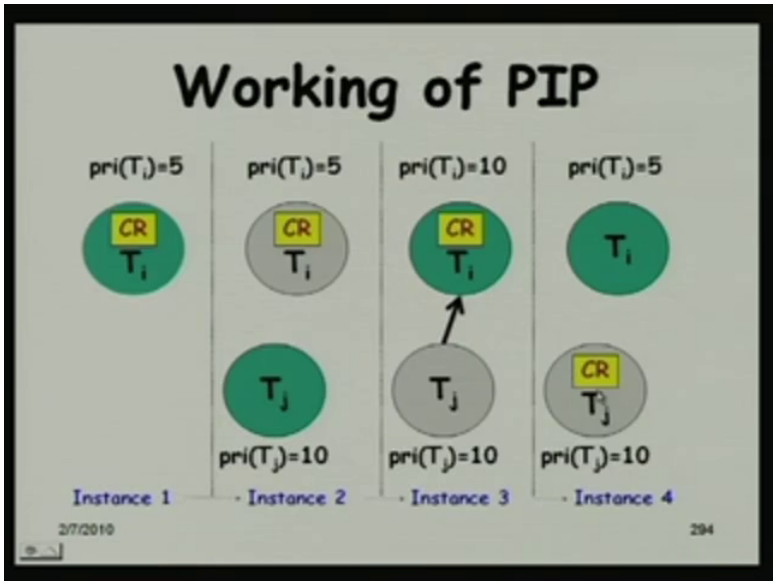
(Refer Slide Time: 30:54)



How does priority inheritance protocol prevent unbounded inversions?

The task cannot preempt now 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.

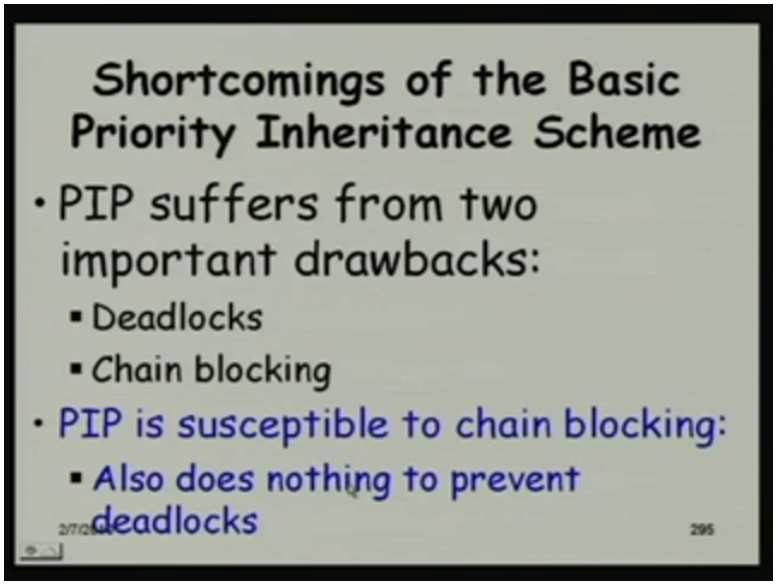
(Refer Slide Time: 31:21)



This is an example of the working of 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. Then this was preempted, because T_j did not require the resource and it started executing. But after sometime, it needed the resource, and it started waiting for the resource to be released by T_i, and T_i got execution on the CPU.

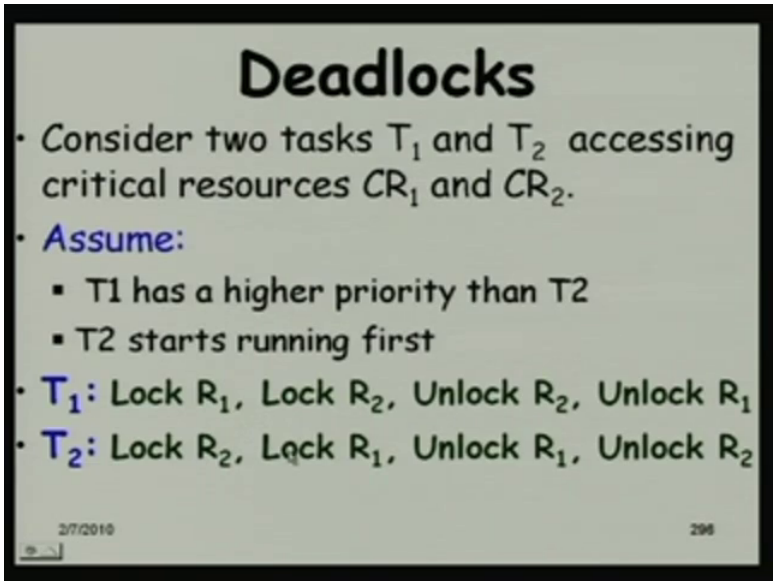
Due to the inheritance protocol, the priority of T_i is raised to 10, because T_j is 10. The inheritance clause by that priority is raised, and after it completed the resource usage, its priority was changed to 5, and then, T_j has got hold of the resource and it will start executing now.

(Refer Slide Time: 32:43)



The protocol is very simple basic priority inheritance scheme, but there are problems with this protocol. Two main problems are deadlocks and chain blocking.

(Refer Slide Time: 33:08)

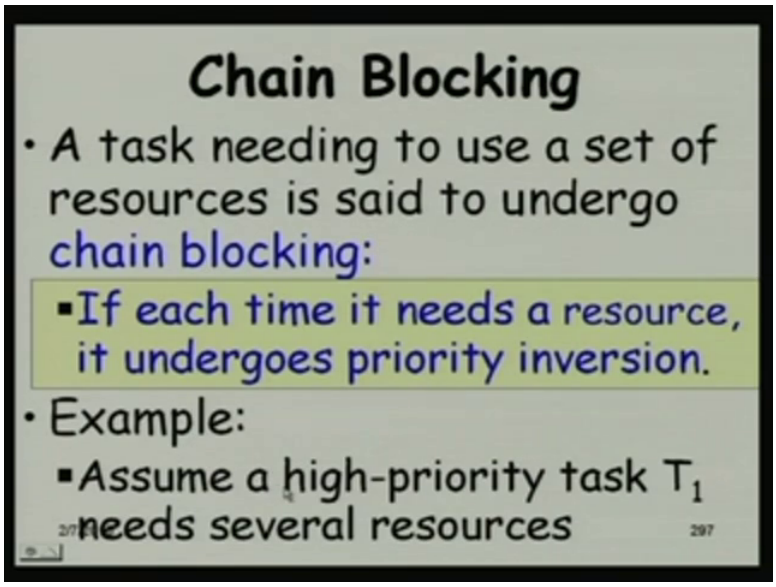


Assume that, we have two tasks T_1 and T_2 and they are accessing the resource CR_1 and CR_2. Also assume that T1 has higher priority than T_2, and T_2 starts running first.

T_2 is the lower priority and T_1 is the higher priority, T_2 locks resource R_2, and started executing. At that time T_1 started executing, preempted T_2 from CPU users, and locked

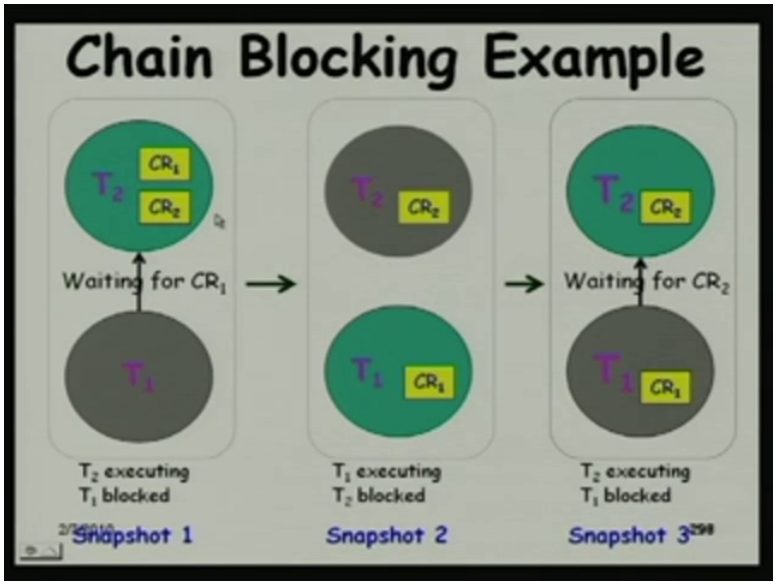
R_1. After some computation, it tries to lock R_2, but R_2 is the lock being held by T_2 and it will block for T_2. And, T_2's priority will be raised to that of T_1 through the inheritance clause, but T_2 has they have already got into deadlock even if priority is raised.

(Refer Slide Time: 34:32)



Deadlock occurs using the simple priority inheritance scheme. The other problem is chain blocking. The definition of chain blocking is whenever a task needs a set of resources, and 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 say, CR_1 and CR_2, two resources. After some computation, it blocks for T_2, and then, T_2 release CR_1 after executing for some time, and T_1 could execute with CR_1, but after sometime, it needed CR_2, and then, it started blocking again. 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.

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

This principle just converts the unbounded priority inversion to a normal priority inversion.

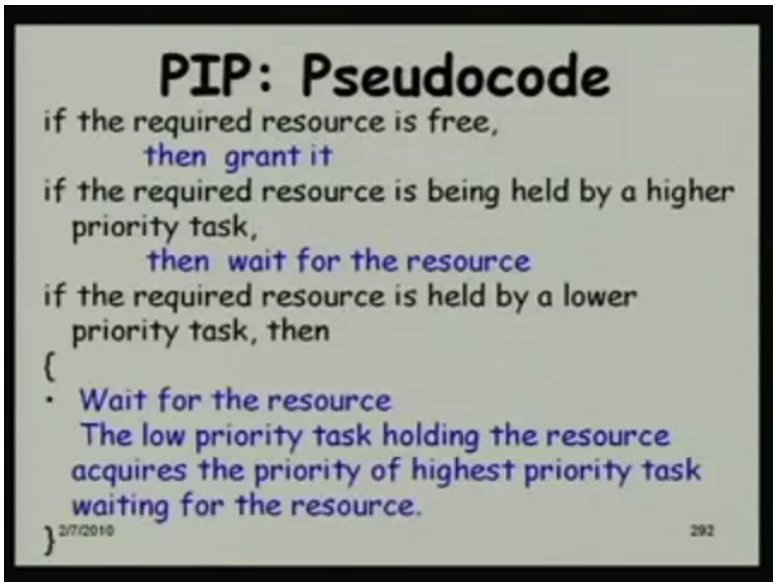
We have not eliminated a simple priority inversion; a simple priority inversion can never be eliminated, but what we have said is that a simple priority inversion can be tolerated by careful programming, by splitting the access to a critical resource of a low priority

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.

When a process does have blocked for some resource and another higher priority process comes and it also blocks for the same resource. 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?

The question is that when the priority inheritance takes place? The basic pseudo code is given in the slide.

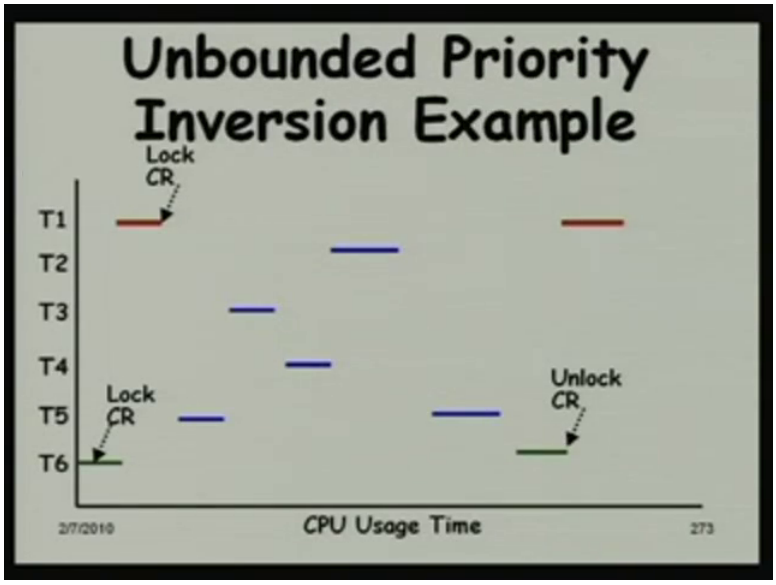
(Refer Slide Time: 38:59)



Whenever there is a request for a resource, this pseudo code is applied. Any task is requesting for a resource and if it is resource is free then grant it. If the resource is held by a higher priority task, then wait. And keep on waiting for the resource.

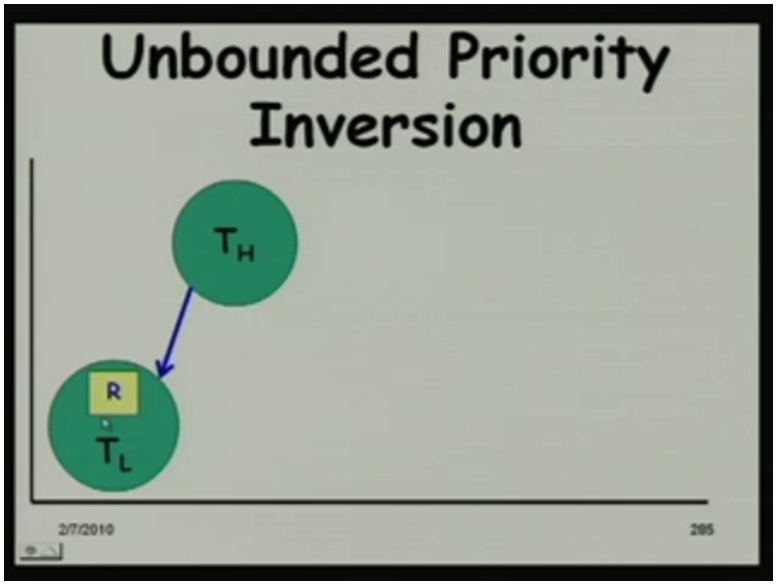
If the lower priority task is itself not running and the higher priority task is waiting for resource, which is held by the lower priority task, which is not running on the CPU. So, in that case, with the lower priority task inherit the priority of the higher priority task.

(Refer Time Slide: 40:56)



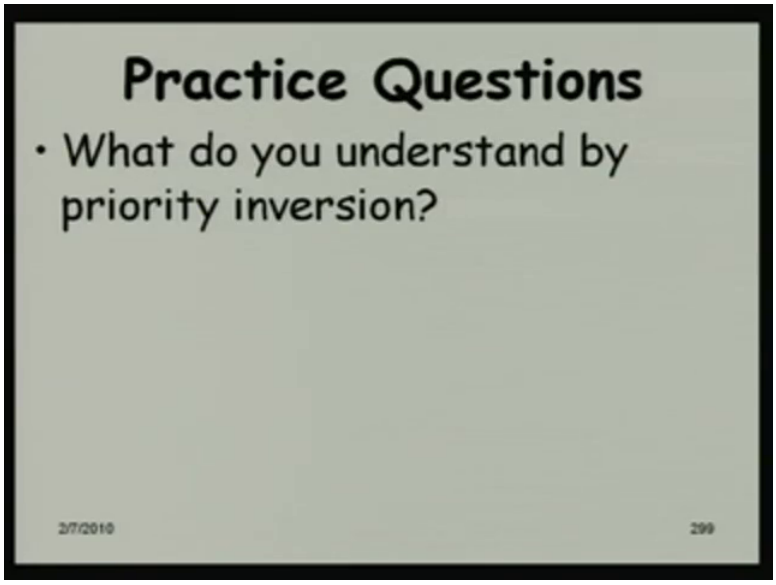
This is the example. The lower priority task was running for some time, and then, it was preempted by the higher priority task. While the higher priority task was running, it requested the resource. So, very likely the lower priority task will not be running when the resource is requested.

(Refer Slide Time: 42:18)



A task will get the highest priority of all the tasks that are waiting for various resources, which is being held by it.

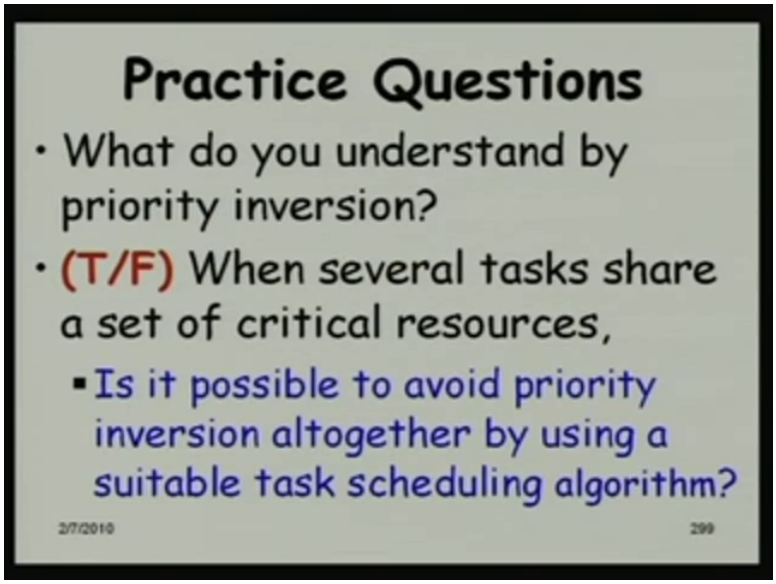
(Refer Slide Time: 43:09)



Question: What is priority inversion?

Actually situation where the higher priority task waits for the lower priority task in case of a resource need lower priority task. Lower priority task keeps on executing and a higher priority task is waiting.

(Refer Slide Time: 43:30)

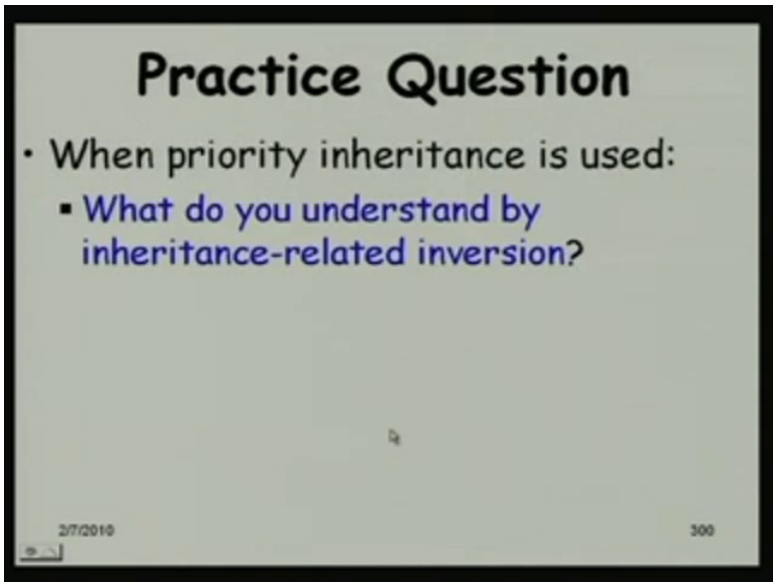


Whether this sentence is true or false?

Sentence: When several tasks share a set of critical resources, it is possible to avoid priority inversions altogether using a suitable scheme.

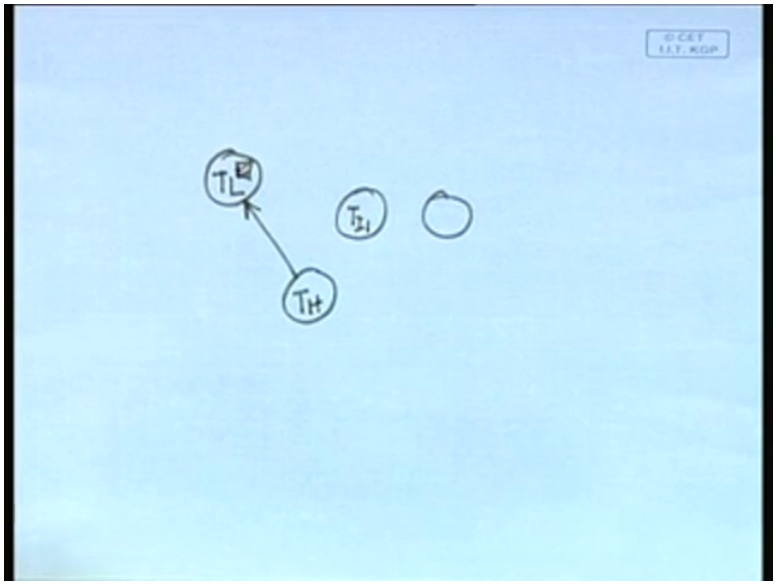
Answer: No, because a simple priority inversion is very difficult to prevent.

(Refer Slide Time: 44:01)



When a priority inheritance is used, what is inheritance- related inversion?

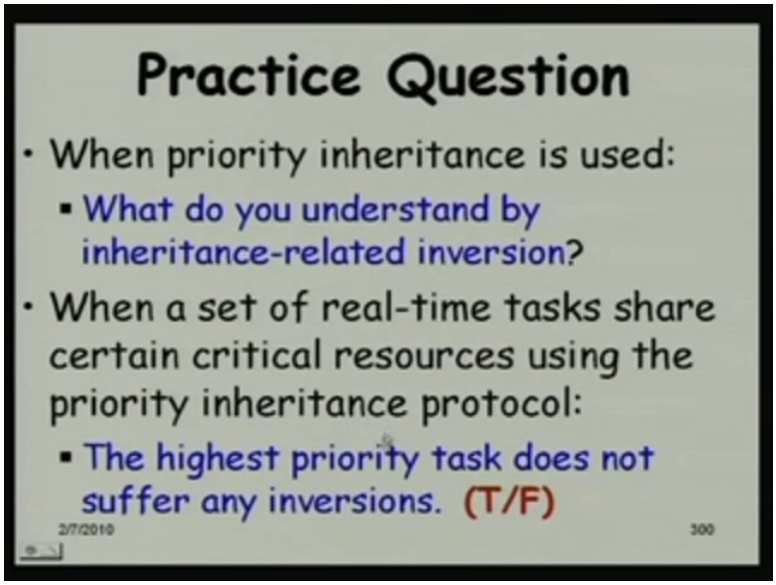
(Refer Slide Time: 44:45)



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.

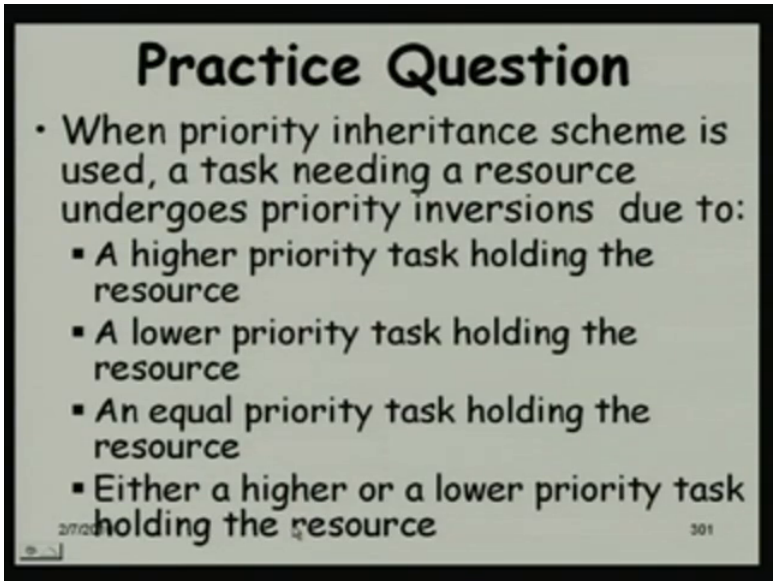
(Refer Slide Time: 45:19)



Statement: When a set of real-time tasks share critical resources using the priority inheritance protocol, then the highest priority task does not suffer any inversions.

Solution: True, Lower priority task, by its by definition itself, it cannot suffer any inversion.

(Refer Slide Time: 46:03)

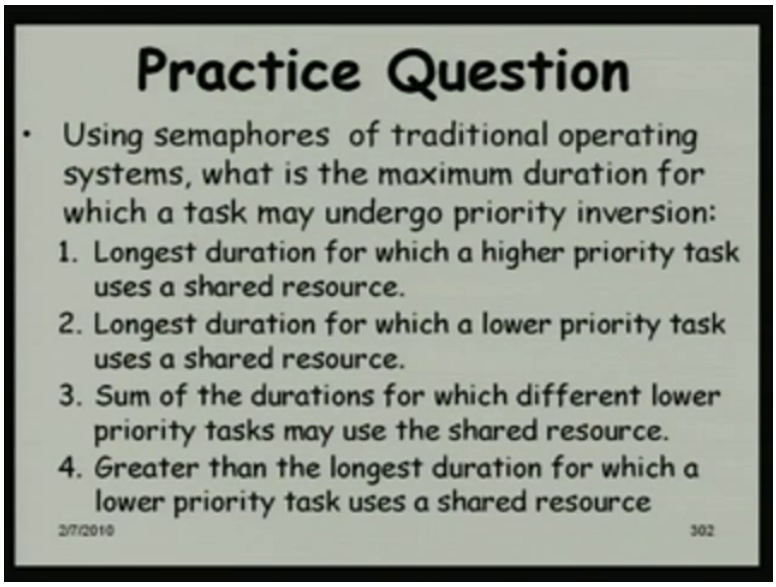


When a priority inheritance scheme is used, a task needing a resource undergoes a priority inversion due to:

1) Higher priority task holding the resource, 2) A lower priority task holding the resource, 3) An equal priority task holding the resource, 4) Either a higher or a lower priority task holding the resource.

Solution: A lower priority task holding the resource, because 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.

(Refer Slide Time: 47:36)



Using semaphores of traditional operating system, what is the maximum duration for which a task may undergo priority inversion?

i) The longest duration for which a higher priority task uses the resource;

ii) The longest duration, for which a lower priority task uses the resource; s

iii) Sum of the durations for which different lower priority tasks may use the shared resource

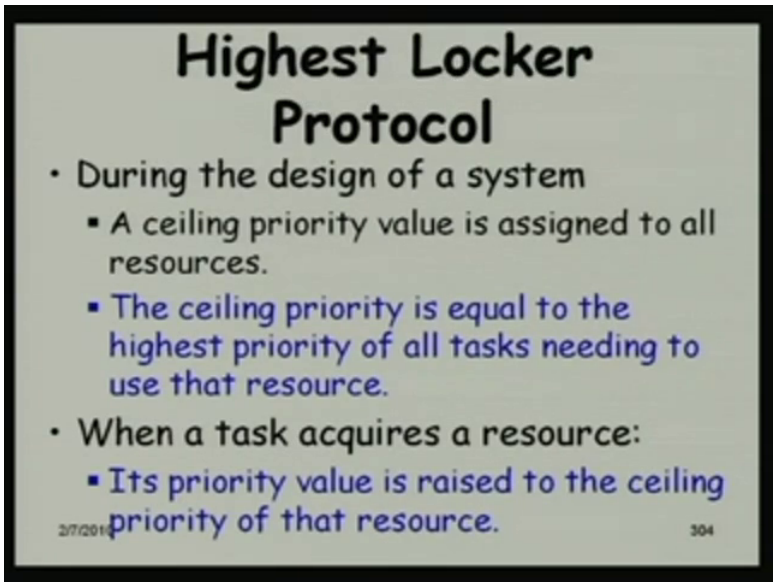
iv) Greater than the longest duration for which a lower priority task uses the shared resource.

Answer: Greater than the longest duration for which a lower priority task uses the shared resource.

Because even when they are not using the resource only holding the resource, there may be intermediate priority task, which are not requiring the resource and they have preempted it. It is longer than the maximum duration for which a task holds the resource. We cannot say that it is sum of the durations for different tasks need the resource.

We discuss some of the improvements that were proposed to overcome the problems of the basic priority inheritance scheme.

(Refer Slide Time: 49:58)

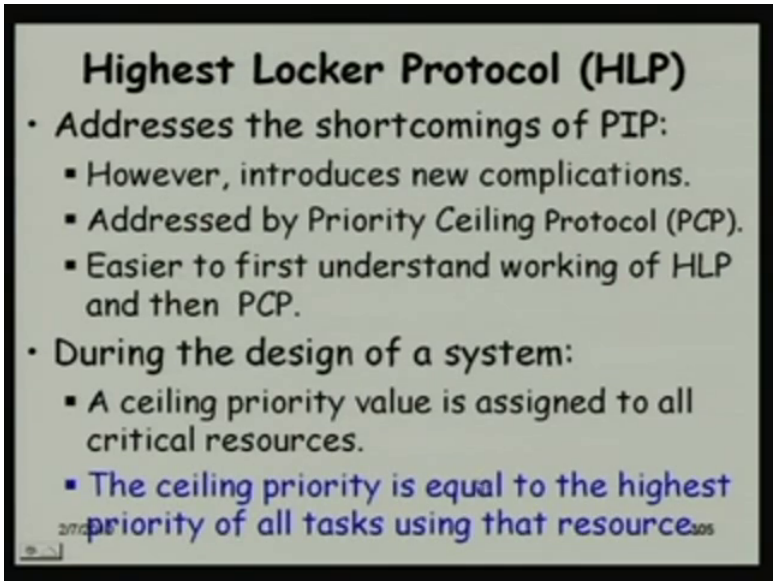


Highest locker protocol is a straight extension of the inheritance protocol, but itself has problem. It solves the main problems of the basic inheritance scheme, but it creates new problems.

In the highest locker protocol a ceiling priority is assigned to every resource and the ceiling priority is computed as equal to the highest priority of all tasks needing that resource.

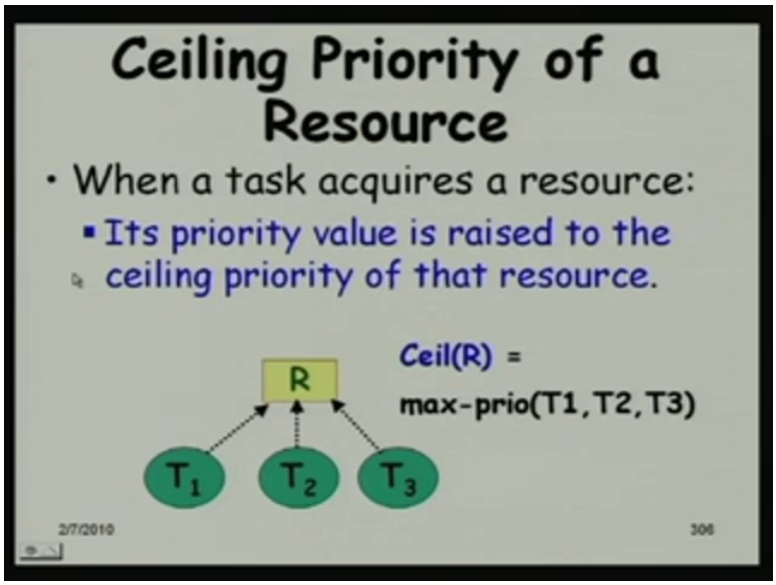
Once we have computed the ceiling priority, whenever a task acquires a resource raise its priority to that of the ceiling priority.

(Refer Slide Time: 51:49)



It addresses all the shortcomings of the basic inheritance scheme but it introduces new complications, which will be addressed by the ceiling protocol. 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)



For every resource R, we will find out which are the tasks using 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.

Let us stop here, we will continue from this point in the next class.