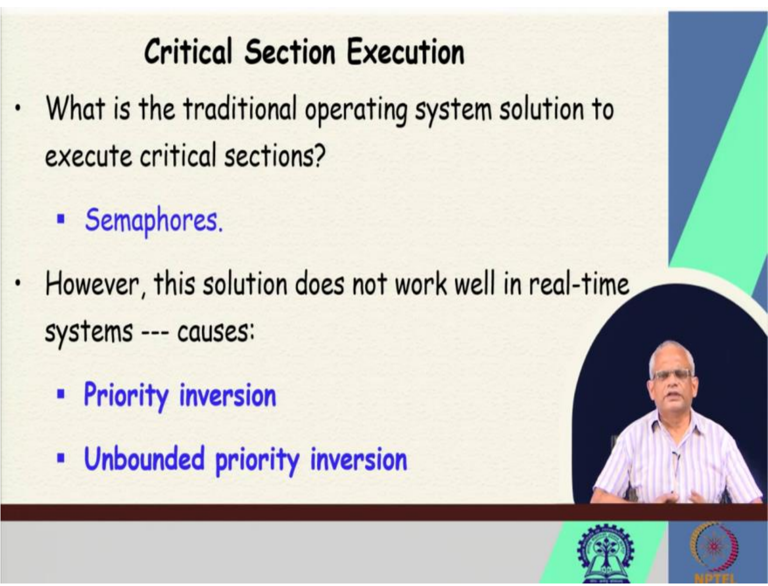
Real Time System
Professor Rajib Mall
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture: 26
Resource Sharing Among Real-Time Tasks

Welcome to this lecture. In the last few lectures, we had looked at uniprocessor scheduling with the simplified assumption that tasks do not share any resources, but in any realistic application tasks do share resources. And the traditional operating system solution of semaphores is not suitable for resource sharing among real time tasks. And we need a special mechanism to let tasks share resources in a real time system. And we were just having a bit of introduction last lecture towards the end of the last lecture. Now, let us proceed from there.

(Refer Slide Time: 01:07)



A critical section is a part of the application code where some non preemptable shared resources accessed. And once a task enters its critical section, it cannot be preempted. Because if we preempt it, it would leave the resource in inconsistent state and finally, the result will be a failure of the system. So, let us look at the critical section execution.

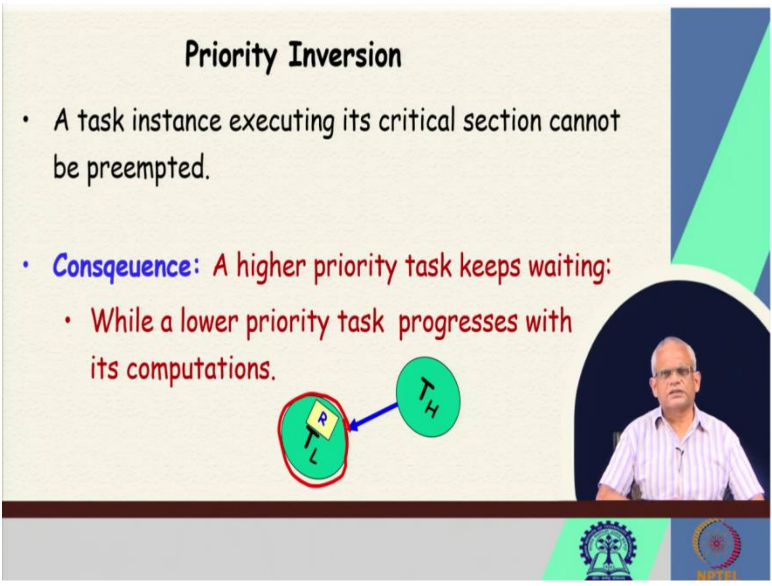
In the traditional operating system, preemption is prevented of the resources in the critical sections by using semaphores. Semaphores is a well-known solution in traditional operating systems, but if we use this in real time systems to share resources among real time, time bounded

tasks, then there is going to be failure of the system. Now, what causes this problem, let us investigate.

There are two major problems one is simple priority inversion which can be as you will see can be circumvented the effect of a priority inversion can be observed by careful programming. But unbounded priority inversion cannot be avoided with any amount of careful programming. It needs a special mechanism to be implemented in the operating system, the operating system needs to support this to prevent unbounded priority inversion.

So, let us first understand what is priority inversion and what is unbounded priority inversion and how these can be prevented, what are the different mechanisms, their advantages and disadvantages and which is the best mechanism for prevention upon unbounded priority inversion and how can it be implemented in the operating system, these are the issues that had to be discussed now.

(Refer Slide Time: 03:53)

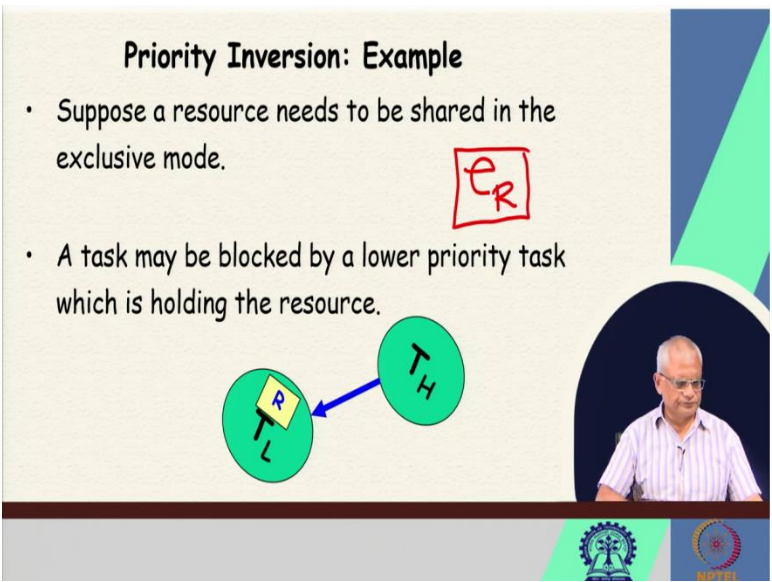


A simple priority inversion is it occurs when a task instance that is executing in its critical section cannot preempt it from using the resource. The task in its executing in its critical section cannot be preempted and as a result the high priority task which might be needing the resource would have to keep on waiting until the low priority task which is using the resource completes its usage of the resource.

So, this is the situation here that low priority task which we have denoted as T_L is using a non-preemptable resource R. It has acquired the resource R and using it. But a high priority task it became ready and the rate monotonic scheduler naturally started executing the high priority task. But after some time the high priority task needed the resource R, the non-preemptable resource R, but T_L cannot be preempted from its usage of resource R.

And as a result, T_H blocks and T_L proceeds with its execution and this is the situation of a priority inversion where a high priority task T_H is waiting for the resource and the low priority task T_L is proceeding with its computation and if T_L holds the resource for long enough time T_H can miss its deadline. A priority inversion, a simple priority inversion can cause a high priority task to miss its deadline.

(Refer Slide Time: 6:21)

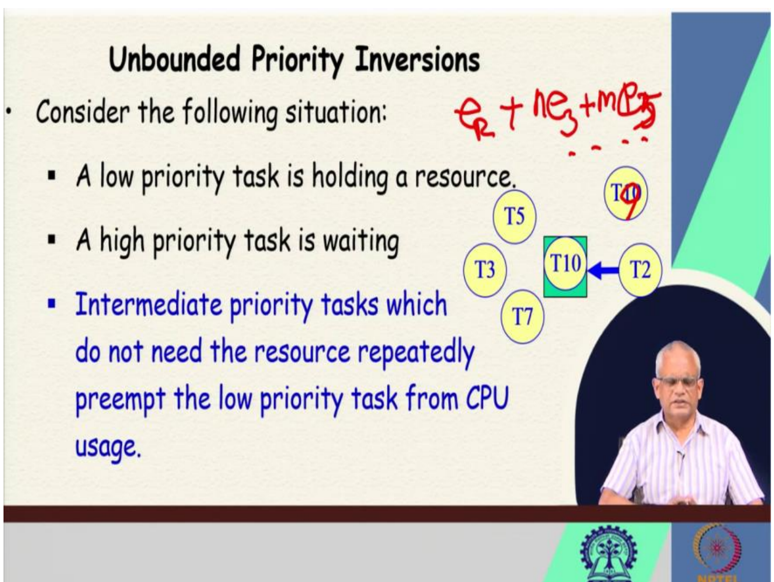


So, an example T_L acquire the resource R first, executing resource, executing using resource R and that time T_H became ready and after some time needed the resource R but needs to block because T_L's usage of resource R must complete. So, what is the maximum duration for which T_H can block? The maximum duration for which T_H can block is the maximum duration for which T_L needs the resource R.

Let us denote e_r is the TL's execution time for the resource R, e_r is the time for which T_L needs to execute using resource R. So, the maximum blocking the T_H can undergo on account of a simple priority inversion is bounded by e_r, it can be less than e_r because T_H may get enabled.

Sometime after T_L has used, already used let us say half of e_r execution time is over. So, the waiting time for T_H will be half e_r. But at most T_H can block for e_r time that is due to a simple priority inversion.

(Refer Slide Time: 08:07)



But now let us look at the situation of an unbounded priority inversion. In an unbounded priority inversion let us first understand the situation in which unbounded priority inversion occurs. Let us assume T10 is a very low priority task and it is holding a resource, already acquired resource when the other tasks we are not needing it and it is executing and a very high priority task T2 which became ready and started to execute needed the same resource that T10 is holding and therefore, T2 blocks for T10.

But during this time when T2 is blocking for T10 other tasks which have higher priority than T10 but lower priority than T2, which are intermediate priority tasks, they become ready and the preempt T10 from usage of the resource. So, all these tasks T7, T3, T5, T10, T9, etc., they are higher priority than T10, but lower priority than T2.

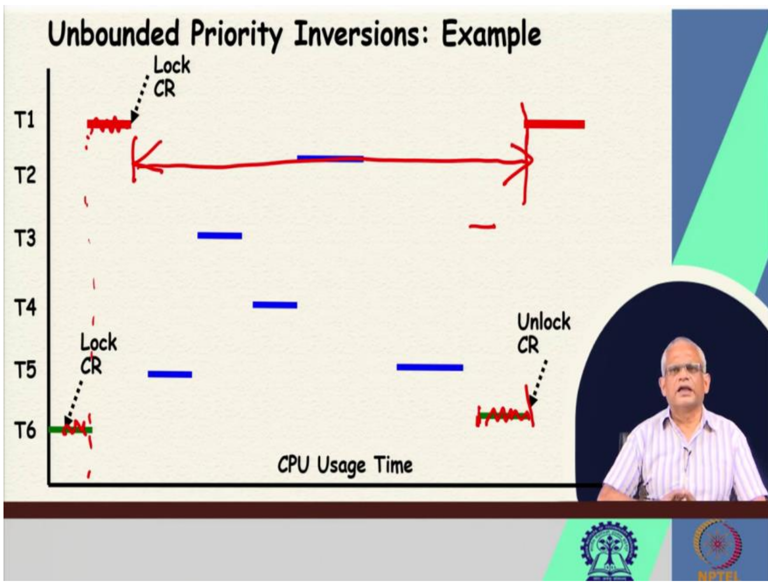
Now, they start executing and T10 cannot complete its usage of the critical resource because it is not getting the CPU to execute. So, T2 keeps on waiting. So, T2 undergoes priority inversion not only due to T10 but also due to other tasks T7, T3, T5, T10, and by the time T9 completes execution may be another instance of T3 comes up and so on.

So, T2 theoretically can keep on waiting indefinitely because they become ready again and again and T10 cannot really complete its usage of the resource. What is the maximum blocking time here? T2 will block due to the usage of the critical resource e_r by T10 + the execution times of all other tasks and they may occur many times.

So, n times execution time of task T3, m times execution time of e5 and so on. And n and m can be large numbers. So, n, m, etc., can be large numbers. So, T2 need to wait for long duration to get the resource and T2 is definitely going to miss its deadline, if there is an unbounded priority inversion situation.

Compared to a simple priority inversion where if the low priority task holds the resource for a long time, then only the task, high priority task can miss its deadline, but normally tasks do not hold the resource for a long time. So, a task may not miss its deadline due to a simple priority inversion. But if there is an unbounded priority inversion situation, then high priority task is very likely to miss its deadline. So, it is a very bad problem, unbounded priority inversion is a very bad problem and unless you do something then the system is going to fail.

(Refer Slide Time: 12:43)



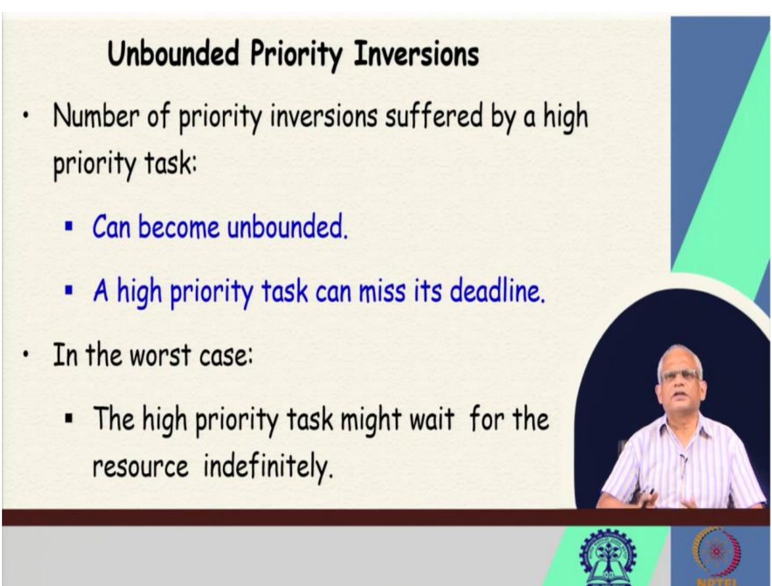
This is the same example where a low priority task T6 locks a resource a shared resource shared non-preemptable resource written CR critical resource. And as it locked here and started using after some time, the high priority task T1 preempted T6 and started executing but after some time it wanted to lock, acquire the resource CR. But CR is already locked by T6.

So, T1 blocks for the resource. But there are other intermediate priority tasks which were waiting because the high priority task was using the CPU. The other tasks were waiting, but now as T1 release the resource the other tasks, they were allocated the CPU by the scheduler, because these are the higher priority tasks and many instances of them kept on coming.

And finally, T6 got the CPU and it executed for some time and released the resource CR and as soon as it release the resource CR the task T1 started to execute and the blocking time, the priority inversion time here is this and it can vary from instance to instance, depending on how many intermediate tasks are there at that time.

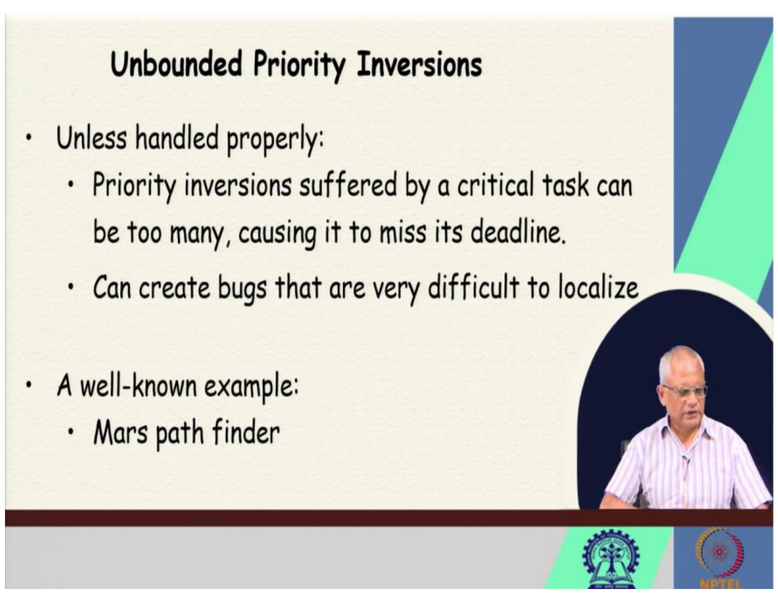
So, it is a long interval, but we cannot put a bound on it because depends on how many intermediate priority tasks get ready and start executing and that is why we call it as an unbounded priority inversion, we cannot put a bound on the blocking time of the high priority task.

(Refer Slide Time: 14:58)



We cannot put a bound on the number of inversions, priority inversion supported suffered by the high priority task and therefore, can become unbounded, the high priority task can easily miss its deadline and in the worst case, the high priority task might wait for the resource indefinitely and in the meanwhile, other tasks are executing.

(Refer Slide Time: 15:40)



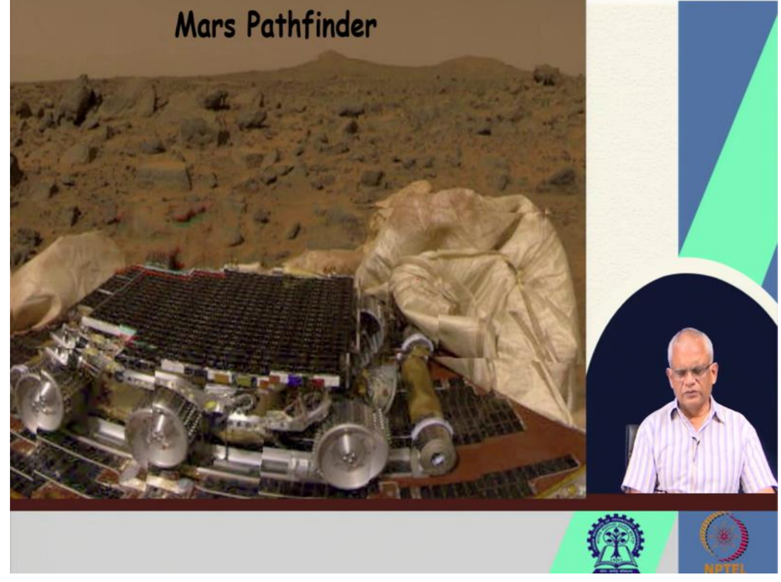
So, very important problem must not only understand the problem, but also how to handle this problem to be able to successfully design and develop a real time system we must understand unbounded priority inversion and how can we successfully handle this problem. And if we are not aware of the unbounded priority inversion problem, we will see that the system has failed.

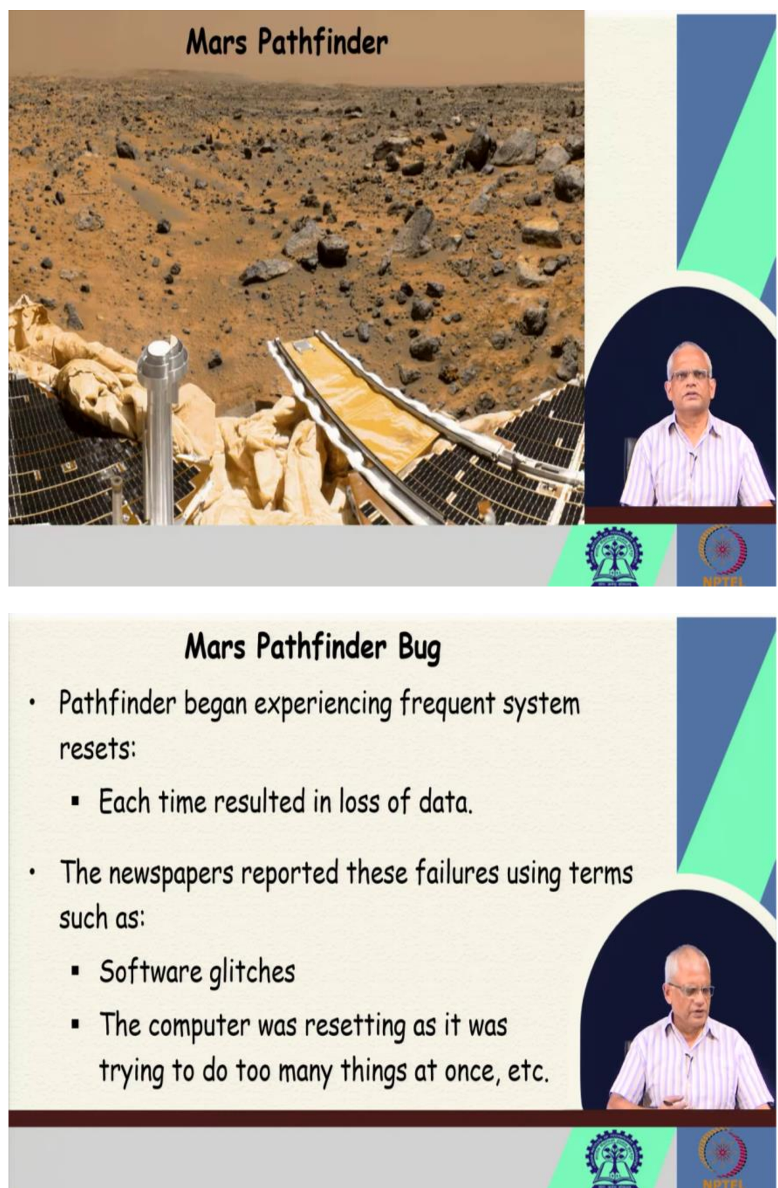
And if we do not look for this specific problem, if we just look at the code, it becomes very difficult to localize the fault that why the system is failing, everything is okay, it becomes very difficult. A well known example of the unbounded priority inversion is the Mars path finder. Let us look at this well known example, the Mars Path finder problem.

(Refer Slide Time: 16:49)

Mars Pathfinder

- Landed on the Mars surface on July 4th, 1997.
  - Bounced onto the Martian surface surrounded by airbags.
  - Deployed the Sojourner rover.
  - Gathered and transmitted voluminous data back to Earth.
  - Included the panoramic pictures released by NASA and now available on the Web.





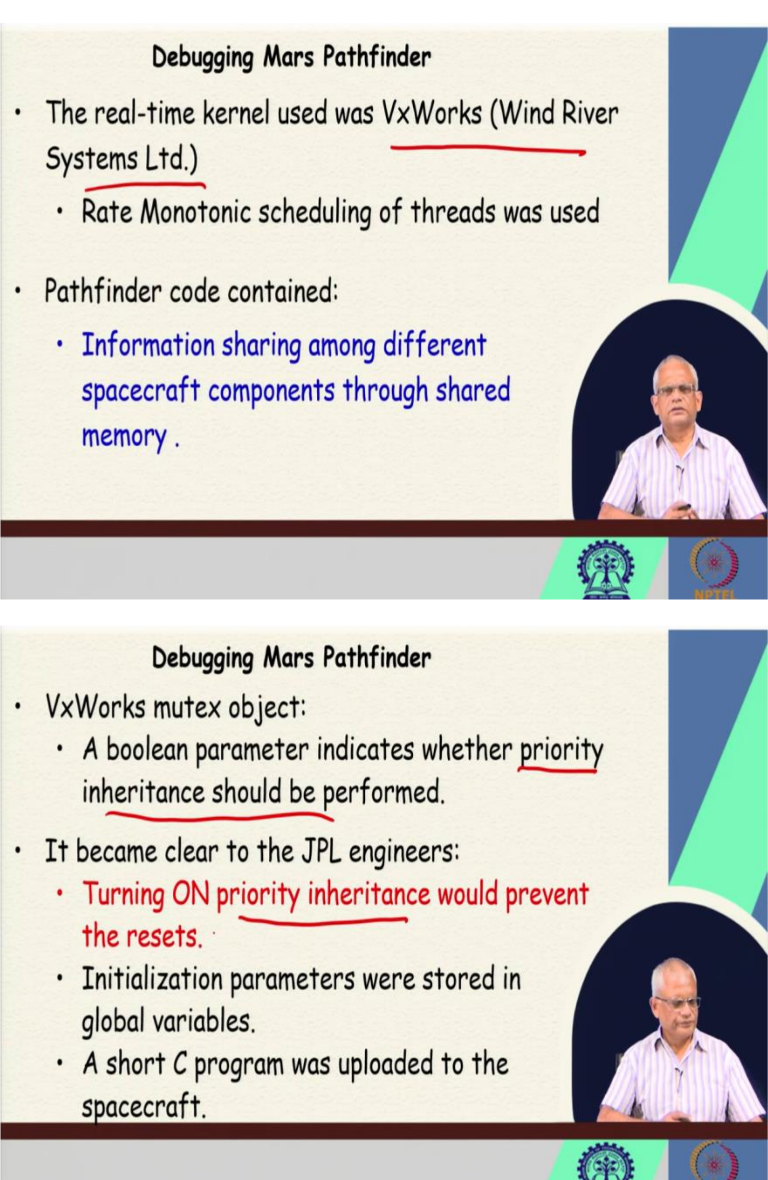
This is the Mars Pathfinder landed on the Mars surface on July 4th, 1997. It bounced onto the Mars surface with airbags and the deployment was perfect. The rover Sojourner, it started transmitting a lot of data back to the earth and there were many pictures. The pictures are released by the NASA and available on the web.

You can also find it just put your one or two pictures here. This is the kind of pictures that it started to send of the Mars surface. And this airbag with which it was dropped and successfully deployed started moving and sending the photographs. This is another photograph which was sent. But then there were problems; the pathfinder began experiencing frequent system resets.

And as it was trying to send the picture, it just failed, loss of data and this were flashed across the newspapers said that the pathfinder is having software glitches.

And some report said that the computer, the pathfinder was getting reset as it was trying to do too many things at once and it was not the computational power it was getting adjusted and therefore it was failing. There were many reports like this, but then the pathfinder is a very expensive mission. And the team members they got into mission mode, started working day and night to find what is causing the problem.

(Refer Slide Time: 19:27)



They were trying to debug the mars pathfinder, trying to find out what was the problem. And to give a brief background is that there was a real time operating system in the pathfinder and the name was VxWorks from Wind River Systems. And rate monotonic scheduling of the threads for different tasks in the mars pathfinder was used and the different tasks shared resources through memory.

The shared memory across different threads was the mechanism by which they pass results among each other. And there was a object, a Boolean parameter, which indicated whether priority inheritance should be enabled or disabled. We will see shortly what we mean by priority inheritance. And due to some reason, during the system initialization time, the priority inheritance was reset.

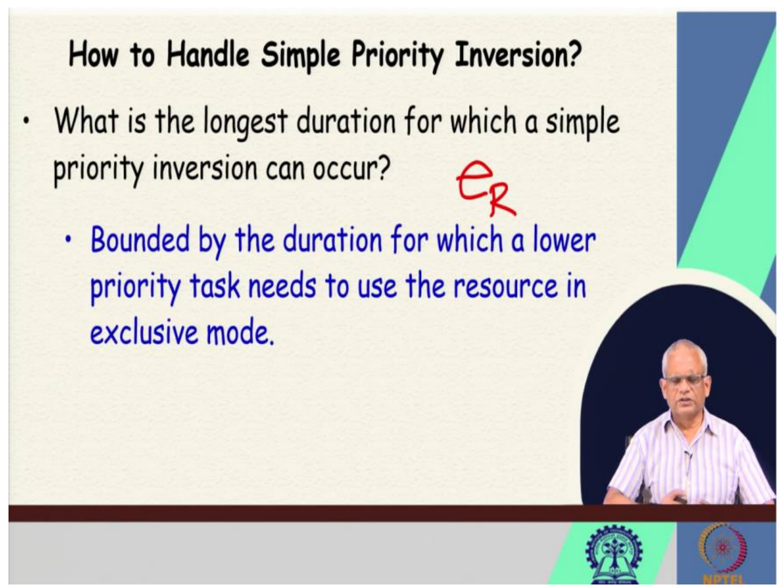
And they were simulating this in the laboratory, the situation in which it was failing and after hours of work it became clear for the Jet Propulsion Lab engineers that somehow the priority inheritance was off. If they could turn on the priority inheritance, then the resets can be prevented, they could simulate the exact same situation with priority inheritance of the initialization parameters were stored in the global variables.

And somehow, before the system was commissioned, the initialization parameter was set wrongly and as they realized it, they just uploaded a short C program to initialize the priority inheritance to on and then the pathfinder started working correctly. So, the priority inheritance mechanism is the way the real time tasks can share resources without undergoing unbounded priority inversion.

Since, the priority inheritance mechanism was off, the different tasks in the pathfinder were undergoing lot of delays due to the unbounded priority inversion and they were missing the deadline and the system was resetting. That was the exception handling mechanism, that when a deadline is missed, the system resets and that is what was happening repeatedly.

As the deadline was missed for a task due to unbounded priority inversion the exception handling mechanism used to reset the system and once the priority inheritance mechanism was turned on, then the tasks could share resources without undergoing unbounded priority inversion. So, a simple thing like a mechanism priority inheritance unless we implement it there can be failures and costly mistakes.

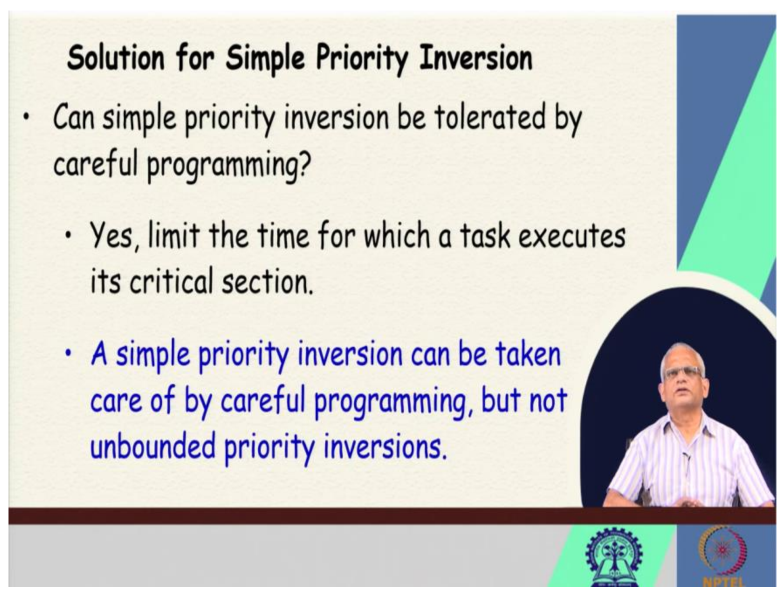
(Refer Slide Time: 23:41)



Now before we look at unbounded priority inversion, let us see how simple priority inversion can be handled. A simple priority inversion, you said that a low priority task is holding the resource and a high priority task waits for it and as soon as the low priority task releases the resource the high priority task acquires the resource.

But what is the longest duration for which the high priority task will block for the low priority task holding the resource? We had seen that this is given by e_r, this is the time for which the low priority task needs the resource. So, the time for which the high priority task undergoes simple priority inversion is given by the time for which the low priority task needs the non-preemptable resource that is given by e_r.

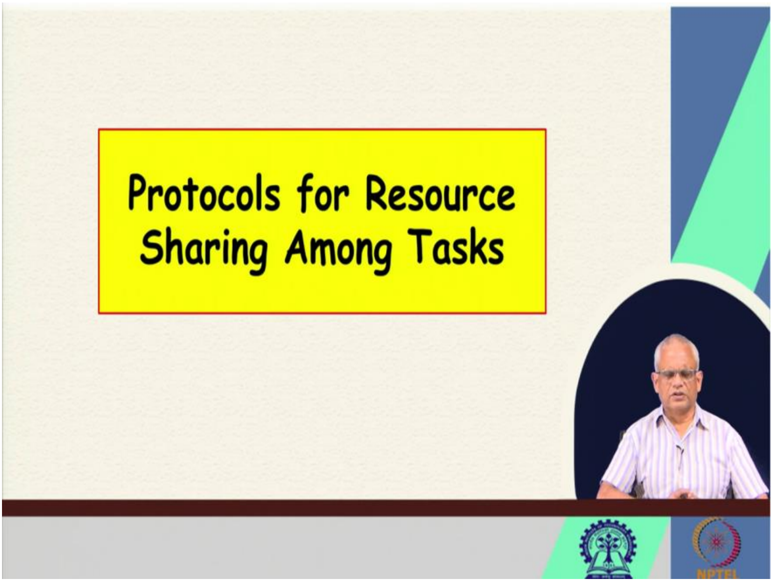
(Refer Slide Time: 24:49)



How can simple priority inversion be handled satisfactorily? Because we had seen that if e_r is long enough it can cause tasks to miss their deadlines. The high priority tasks can miss their deadline if e_r is long enough. So, the idea here to handle simple priority inversion is to minimize the time for which a low priority task executes in its critical section. And how can we do that?

How can we minimize the time for which a low priority task needs its critical resource? We can do careful programming. One is that we break the access to multiple small instances so that the maximum blocking time is e_r / 10 and not e_r. So, if we can reduce the time for which a low priority task at most needs a resource, critical resource, at any instant to low values, the time for which the high priority task undergoes priority inversion can be minimized.

(Refer Slide Time: 26:24)



Now, simple priority inversion is not that big a problem if we have careful programmers who while programming remember that the critical resource usage by a low priority task at any particular time can cause the high priority tasks to wait, block priority inversion can occur and they can miss the deadline.

And therefore a careful programmer while writing the code for a low priority task needs to minimize the access time of the critical resource maybe break it up into small access times. So, that at any time the access time is small. But then, what about unbounded priority inversion? Even with the most careful programming, unbounded priority inversion can cause task to miss their deadlines.

We need special protocols for avoiding unbounded priority inversion. We are at the end of this lecture. And in the next lecture, we will discuss how are what are the mechanisms that exist which can prevent unbounded priority inversions. We will stop now. Thank you.