Real Time Operating System

Prof. Rajib Mall

Department of Computer Science and Engineering

Indian Institute of Technology, Kharagpur

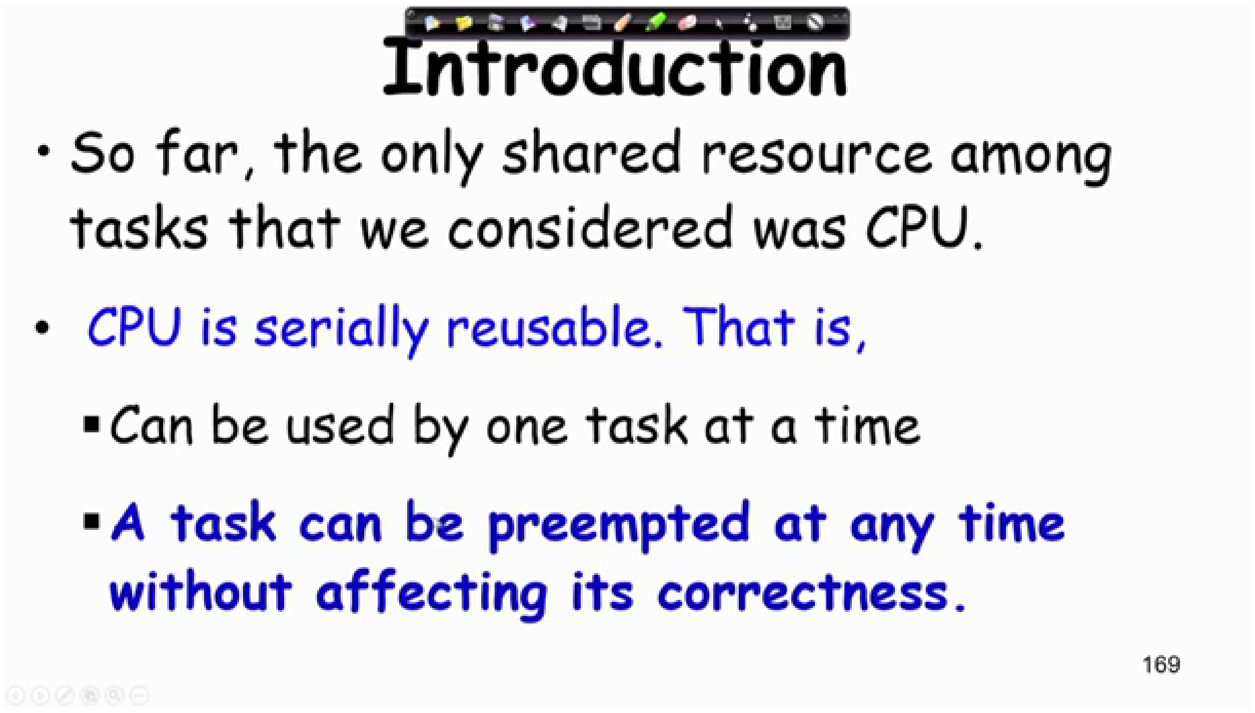
Lecture – 13

Resource Sharing Among Real – Time Tasks

Welcome to this lecture. If you remember in the previous lectures we had the look that some basic aspects of real time operating systems and then we looked at some simple schedulers, the clock driven schedulers, and then we looked at the event driven schedulers and we found that the rate monotonic scheduler is the one which is simple and it can meet the demands of the practical applications and that is the one which is actually used widely even all commercial real time operating systems do support rate monotonic scheduling.

But in our discussion under rate monotonic schedulers we had assumed that tasks are independent, in the sense that they process on their own local data and there is no communication whatever required among these tasks. But then in almost every real application the tasks need to share resources and today this lecture we will see that resource sharing introduces complexity. We need to change our analysis that task scheduling and analysis and we need to have new algorithms to be able to share resources. Let us get started with it.

(Refer Slide Time: 02:12)



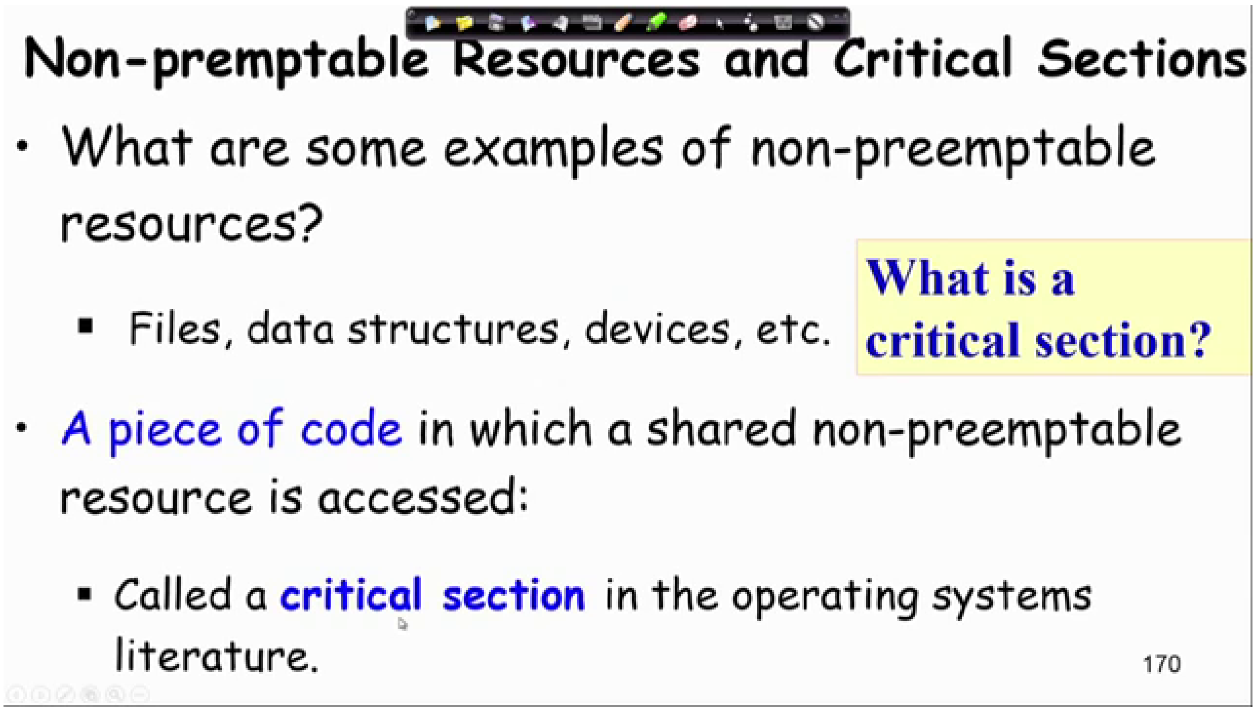
Introduction

- So far, the only shared resource among tasks that we considered was CPU.
- CPU is serially reusable. That is,
  - Can be used by one task at a time
  - **A task can be preempted at any time without affecting its correctness.**

169

We were worried which task should run at one time and that was our simple scheduling problem. If we consider CPU as a resource we can say that CPU is a serially reusable resource, what we mean by a serially reusable resource is that at any time one tasks can run on the CPU, but at the same time we must remember that the simplifying thing that is there is that a task which is running at the CPU can be preempted at any time without affecting the correctness of the result of the task or in other words even though CPU can be used by only one task at a time. But then when we have a higher priority task we can always preempt a lower priority task and later run the task from that point where it was preempted and still our results will be correct. But then now we are going to look at resources which are not serially reusable.

(Refer Slide Time: 03:43)


Non-premptable Resources and Critical Sections

- What are some examples of non-preemptable resources?
  - Files, data structures, devices, etc.
- A piece of code in which a shared non-preemptable resource is accessed:
  - Called a critical section in the operating systems literature.

What is a critical section?

170

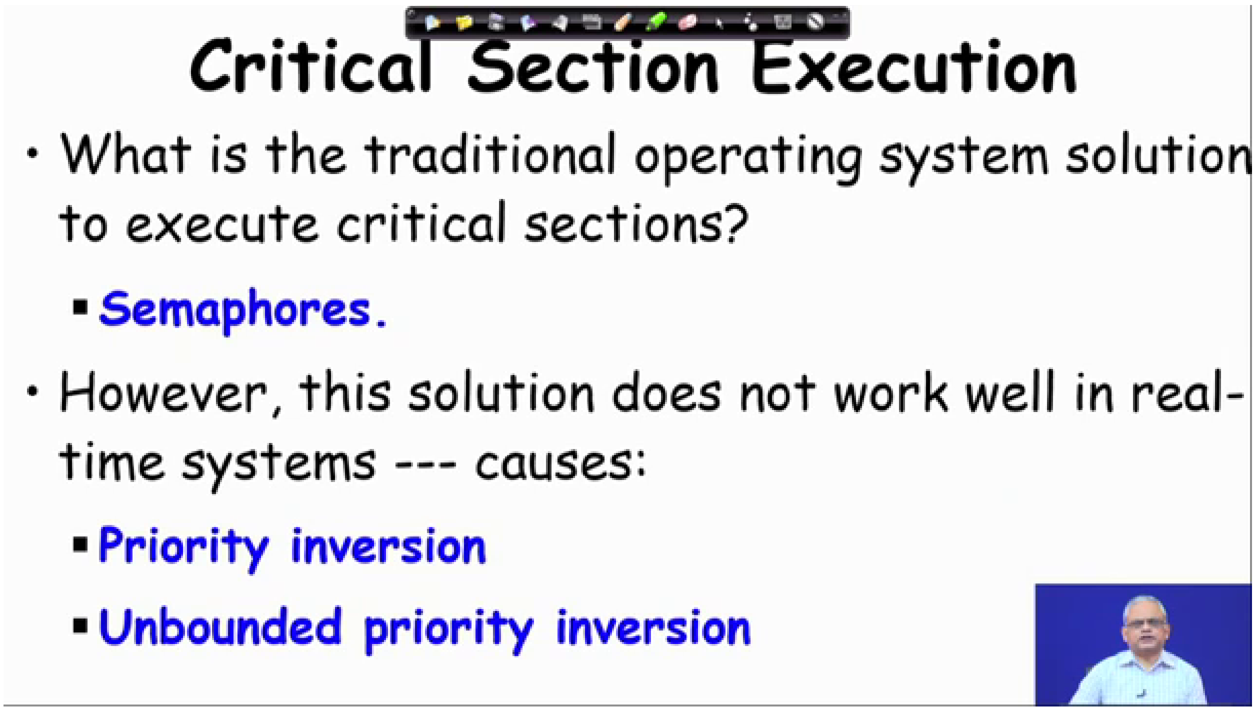
These are the non-preemptable resources and also we look at the critical sections and non-preemptable resource is one where once a task is using the resource we cannot preempt it any time that we need to another task to run need to wait until the task completes use of the resource.

Now, let me ask you this question that can you give some examples of non-preempt able resources, normally discussed in a first level course. So, many of you would be possibly able to answer this question that the examples of non-preempt able resource are many. For example, a file a data structure that is shared among the tasks devices and so on which once a tasks starts using it must use until a point where the results are consistent and then only it can be preempted.

So, files data structures devices etcetera are examples of non-preempt able resources where one task once it starts using these resources it must complete or come to a logical end of use of these resources before it can be preempted otherwise the results will be inconsistent. Just look at the example of a array of data and then one task has changed only some part or maybe it has just read some part and then before it could write it we preempted it then another task which started using the resource overwrote the data and then the task that had read it has got the old data. So, it becomes inconsistent it should have read the data and then return back the result then it would have left it the consistent state.

Now, the next question that would like to ask is that what are critical sections. We know what is a non-preemptable resource and what are some examples of non-preemptable resource. But then what is a critical section? Again it is based on what is discussed in a first level operating system. If you look back to your books the first level operating system course you will find a critical section is a piece of code it is a part of the program in which some shared non-preemptable resource is accessed. So, a critical section is a section of the code in which some non-preemptable resource is accessed that is read and written and this section of code is called as a critical section in all most every operating system book and literature.

(Refer Slide Time: 07:28)



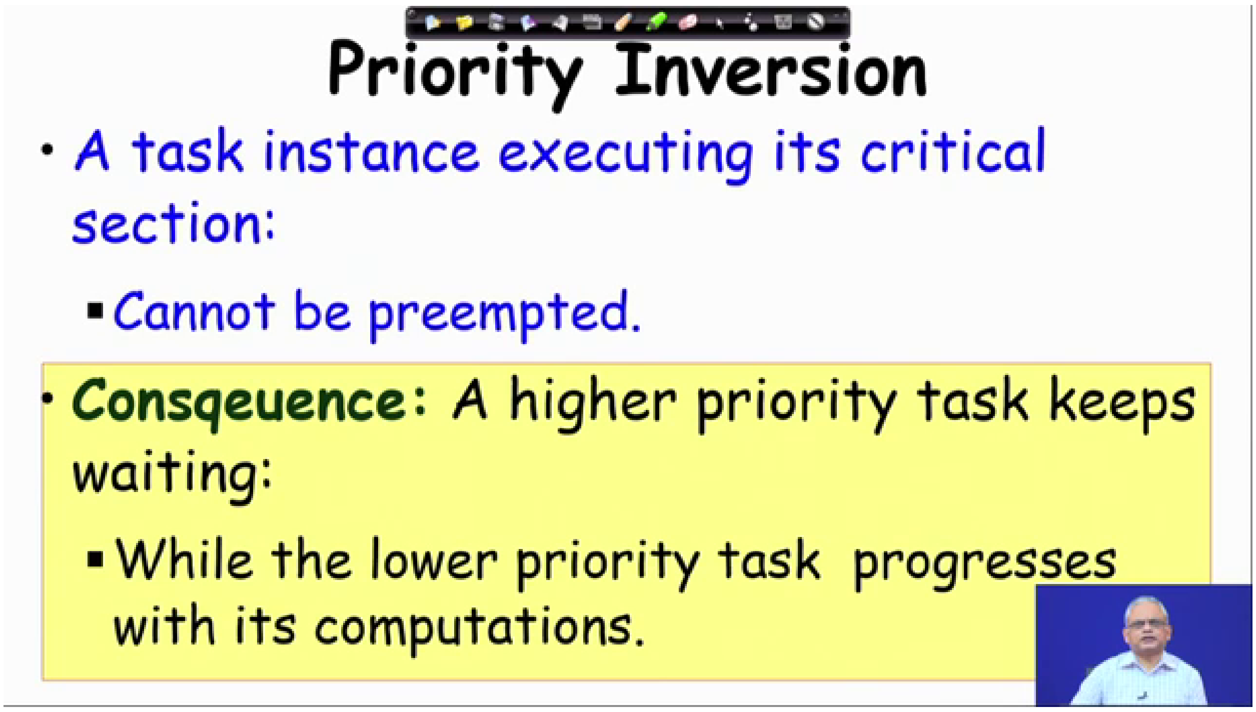
Critical Section Execution

- What is the traditional operating system solution to execute critical sections?
  - Semaphores.
- However, this solution does not work well in real-time systems --- causes:
  - Priority inversion
  - Unbounded priority inversion

But then we know that when a task is executing its critical section it should not be preempted because if it is inside its critical section. So, executing its critical section and it gets preempted then the result will become inconsistent wrong results. So, what are the traditional operating system solution to execute critical section? Again if you remember it is semaphoresk you do not remember you might have to look back at your operating system book. But in a real time application where the task share resources we cannot really use the semaphore the semaphore causes severe problems in a real time application. It causes priority inversion and unbounded priority inversion two very bad problems and unbounded priority inversion can cause a task to miss its deadline and cause the failure of the application.

So, these are two important problems that arise if we use a traditional operating system solution for tasks executing their critical section, let us look at these two problems priority inversion and unbounded priority inversion. First let us look at priority inversion.

(Refer Slide Time: 09:24)



Priority Inversion

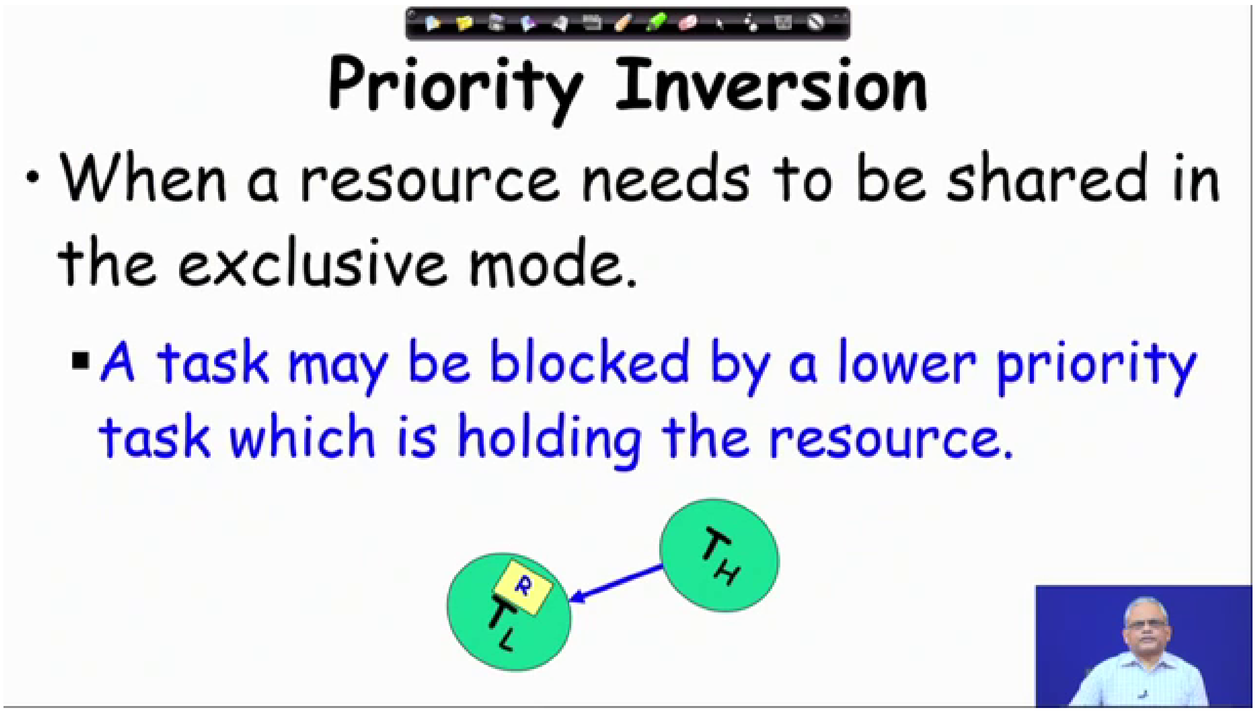
- A task instance executing its critical section:
  - Cannot be preempted.
- Consqeuence: A higher priority task keeps waiting:
  - While the lower priority task progresses with its computations.

To understand what is priority inversion let us assume that a task instance that is a job is executing its critical section and we know that when it is executing its critical section then cannot be preempted and for this purpose let us say we use a solution like semaphore. So, the task invokes the semaphore and then.

Starts using the critical section, but then since the task cannot be preempted when it is executing its critical section a higher priority task that has become ready in the meanwhile would have to keep on waiting because the task executing its critical section cannot be preempted. And it may so happen that the task which is executing a critical section is a very low priority task may be a logging result logging or something and now we have a very high priority task critical task which has become ready, but then it waits before the task that is executing release the semaphore. So, this situation is called as priority inversion and let me just summarize it again because it is a important concept.

The term priority inversion implies that when a low priority task is executing its critical section and a high priority task that is ready keeps on waiting until the low priority task can be preempted. So, the high priority task is ready and needs the resource actually. If it is just ready if it does not need resource it can be executed, but then the high priority task needs the resource and the low priority task cannot be preempted from using its resources and therefore, the high priority task keeps on waiting for the low priority task to release its resource.

(Refer Slide Time: 12:02)

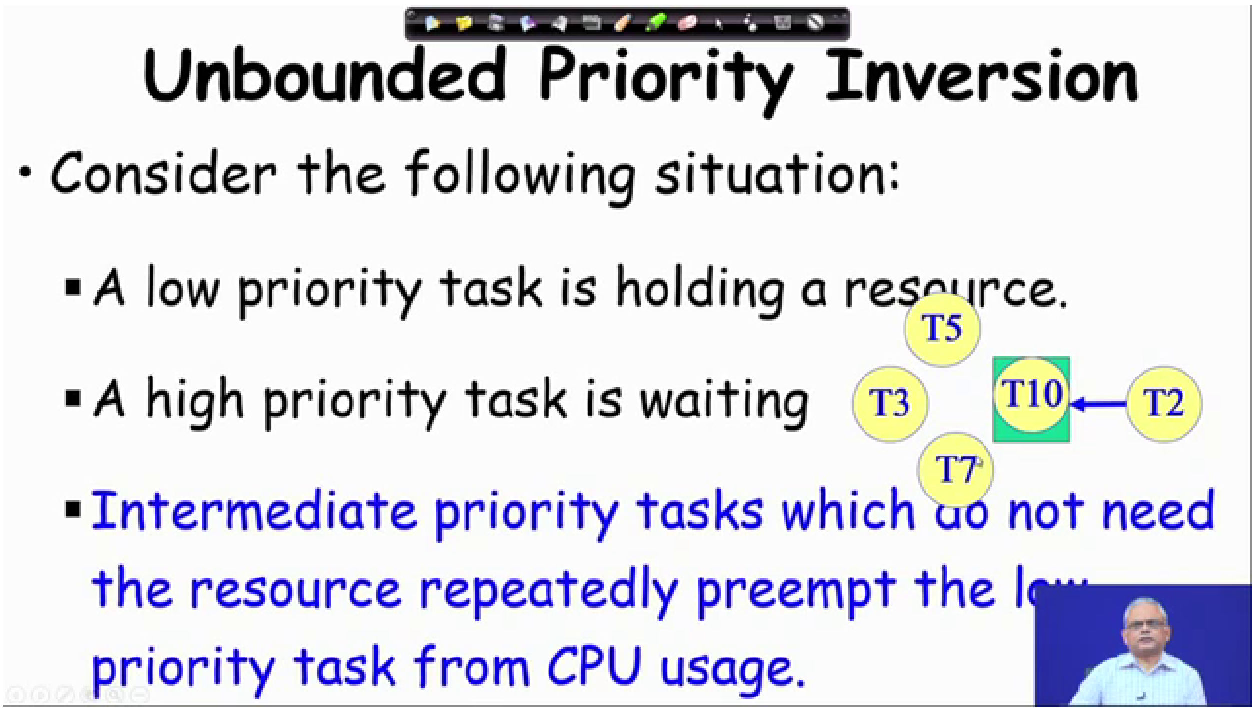


Priority Inversion

- When a resource needs to be shared in the exclusive mode.
  - A task may be blocked by a lower priority task which is holding the resource.

Just give an example of a priority inversion let us assume that the low priority task T L shown here is holding a resource R and doing some computation an R and this R is actually a shared non-preemptable resource. Now, a high priority task needs that resource, but until the low priority task actually completes using the resource and releases the semaphore the high priority task cannot access the resource. So, the high priority task keeps on waiting and if the low priority task holds the resource for a long time the high priority task is of course, going to miss its deadline, but as we will see now that priority inversion by itself is not too bad.

(Refer Slide Time: 12:56)



Unbounded Priority Inversion

- Consider the following situation:
  - A low priority task is holding a resource.
  - A high priority task is waiting
  - Intermediate priority tasks which do not need the resource repeatedly preempt the low priority task from CPU usage.

We can solve the priority inversion problem with careful design, but what really is a problem which is hard to solve is unbounded priority inversion. Let us try to first understand unbounded priority inversion and then we will see why it is difficult to solve and how can the simple priority inversion problem be solved.

To understand unbounded priority inversion let us consider the following situation. We have a low priority task that is holding a resource. So, its executing its critical section now in the meanwhile a high priority task is waiting for that resource. And because the low priority task holding the resource cannot be preempted before it completes its uses of the resource the high priority task is waiting, these are simple priority inversion. But then the unbounded priority inversion arises when there are many intermediate priority tasks which are not needing the resource and there for they can execute and use the CPU the low priority task which was holding the resource cannot make progress with its computation because the intermediate priority task are not needing that same resource that there for do not block they just keep on executing on the CPU. The low priority task cannot completes its execution just keeps on holding the resource and waits for the CPU, but in the mean while the high priority task is waiting.

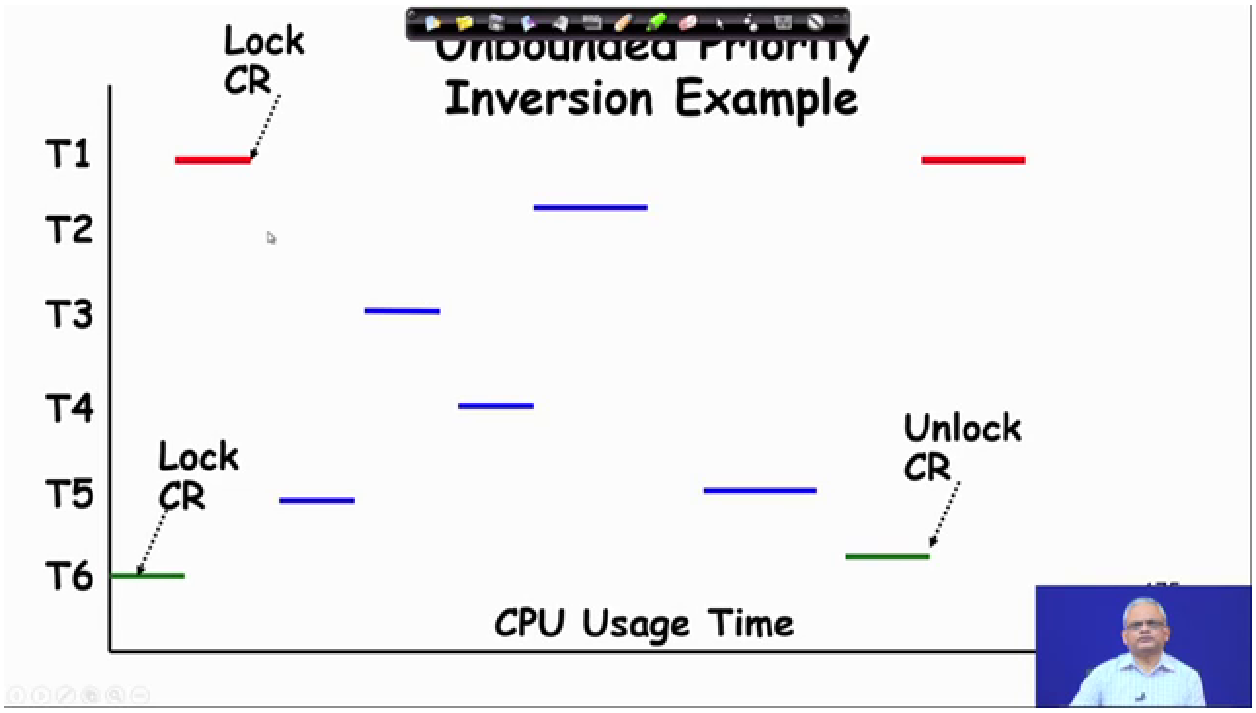
So, this is really a bad situation. The low priority task is holing a resource fine and the high priority task is waiting that is a simple priority inversion, but an unbounded priority inversion situation occurs where the low priority task has been preempted form using the

CPU and therefore, the low priority task is not able to make progress with its resource uses and the intermediate priority task keep on executing on the CPU.

Just to explain the unbounded priority inversion let us look at the following example we have a resource shown as a green box here and we a task T10 which is executing say low priority task. But then after some time into the execution it needs the resource this is a non-preemptable shared resource and since the resource is available T10 starts using the resource. But after it has started using the resource a very high priority task started executing and it needed the resource, but T10 cannot be preempted. So, T2 a very high priority task waits for T10 to complete executing the non-preemptable resource so that T2 can get it, but unfortunately other tasks like T3 T5 etcetera these are having priority more than T10 and they become ready and start using the CPU and since these are higher priority driven T10, T10 is preempted and it cannot complete its execution and mean while T2 keeps on waiting.

Here a simple priority inversion occurs when T2 is simply waiting for T 10, but then the unbounded priority inversion occurs where tasks which are of low priority than T2 they start executing because they are higher priority than T10 and T2 undergoes many priority inversion one due to T10 mutually then letter due to T5, T3, T7 etcetera.

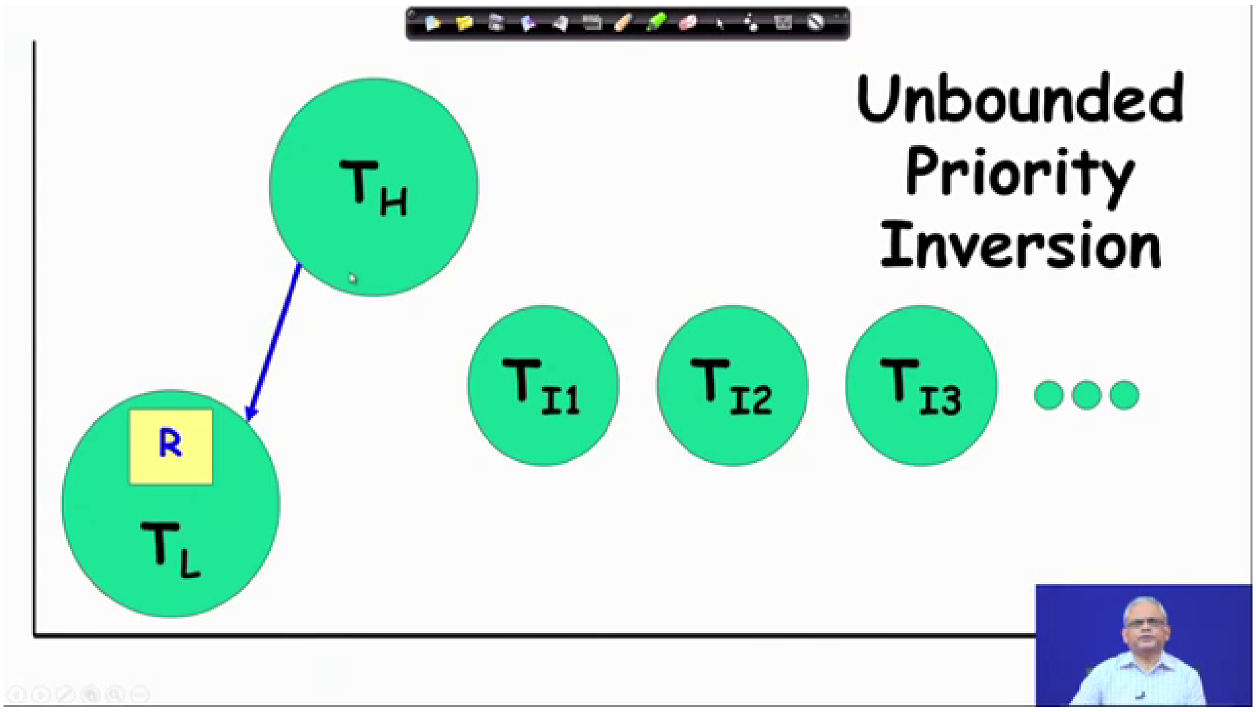
(Refer Slide Time: 18:03)



To explain the concept we have another schematic example here. On the x axis you have we have plotted the task that I have using CPU and for how long they are using the CPU on the y axis we have plotted the tasks. T1 is the highest priority and T6 is the lowest priority and T2, T3, T4, T5 etcetera these are intermediate priority task. To start with T6 started executing over time and after sometime in the execution it needed a non-preempt able resource we will call it as a critical resource CR and since the critical resource was not being used by any other task it could block the resource and started using the resource. But then after sometime it started after sometime T1 became ready, it started executing preempted T6 because T1 is the higher priority started using the CPU started executing, but after sometime into the execution it needed the critical resource CR. But then it will not get CR until it is released by T6. So, the task T1 waits.

But in the mean while T2 T3 T4 etcetera these are tasks which do not need the resource they become ready. So, first T5 became ready and then as soon as the T1 block for the critical resource T5 starts executing, but after sometime T4 T3 becomes ready and it starts executing, and after sometime T4 becomes ready it starts executing and T2 becomes ready starts executing and again the next instance of T5 becomes ready starts executing and so on. And all the while T6 is kept waiting it cannot really complete its uses of the critical resource. But after a long time after many priority inversion each of these task is causing a priority inversion to the task T1 and after many priority inversions task T6 gets the resource starts executing and after sometime releases the resource that is unlock CR and only at that point the T1 task can start executing. So, the task T1 has undergone multiple priority inversion and in the worst case T6 may never get CPU and T1 may get unbounded priority inversion where we do not know how many inversions it will undergo.

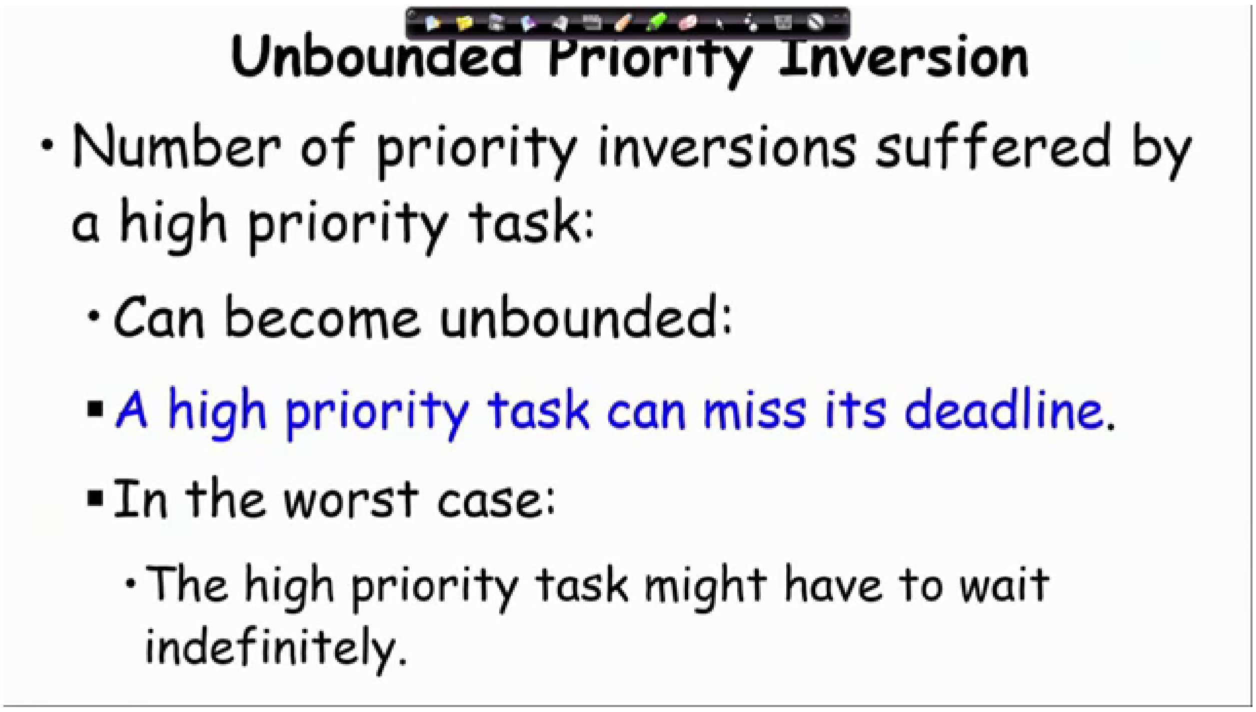
(Refer Slide Time: 21:44)



This is another example to explain what is really unbounded priority inversion because this is an important problem in real time applications. Many applications in the past have failed because they could not take here of unbounded priority inversion problem adequately. So, here a low priority task T L is holding the non-preemptable resources R. So, T L is in its critical section executing and using the non-preempt able resource R after sometime in the execution. So, the x axis is the time axis a high priority task became ready and after sometime into its execution it needed the resources R, but T L is already locked it and the high priority task would have to only wait until T L can release the resource.

But in the meanwhile tasks which are higher priority than T L, but lower priority than T H they keep on arising and executing and therefore, T L is not able to get the CPU and it cannot complete its execution using the non-preempt able resource and as a result the high priority task suffers multiple inversions and we cannot really predict how many inversions it will suffer if it suffers too many inversions then it can definitely miss its deadline.

(Refer Slide Time: 23:53)

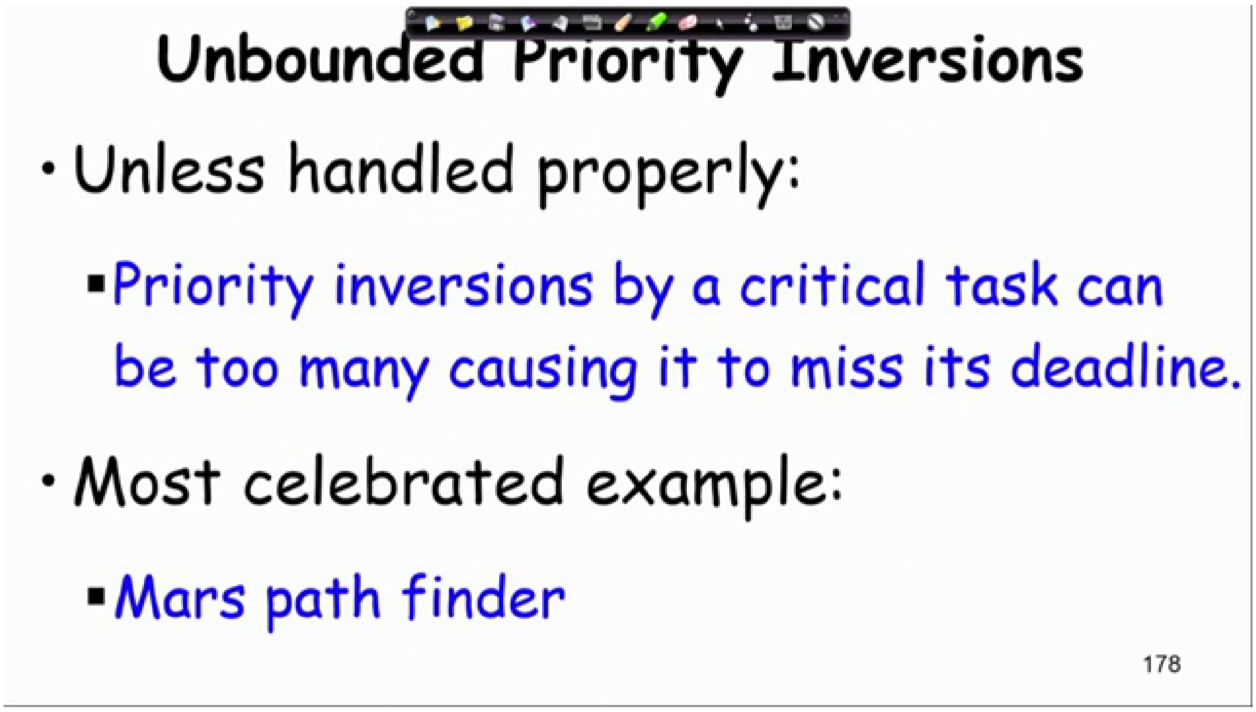


Unbounded Priority Inversion

- Number of priority inversions suffered by a high priority task:
  - Can become unbounded:
  - A high priority task can miss its deadline.
  - In the worst case:
    - The high priority task might have to wait indefinitely.

In the worst case the number of priority inversion suffered by high priority task can become unbounded and this situation can cause a high priority task to miss its deadline and therefore, every real time application developer must take adequate precautions against unbounded priority inversion must understand what this problem is, when does it arise and how to overcome the problem of unbounded priority inversion. If the problem is not taken care then in the worst case when the worst situation arises definitely the high priority task would miss its deadline.

(Refer Slide Time: 24:41)



Unbounded Priority Inversions

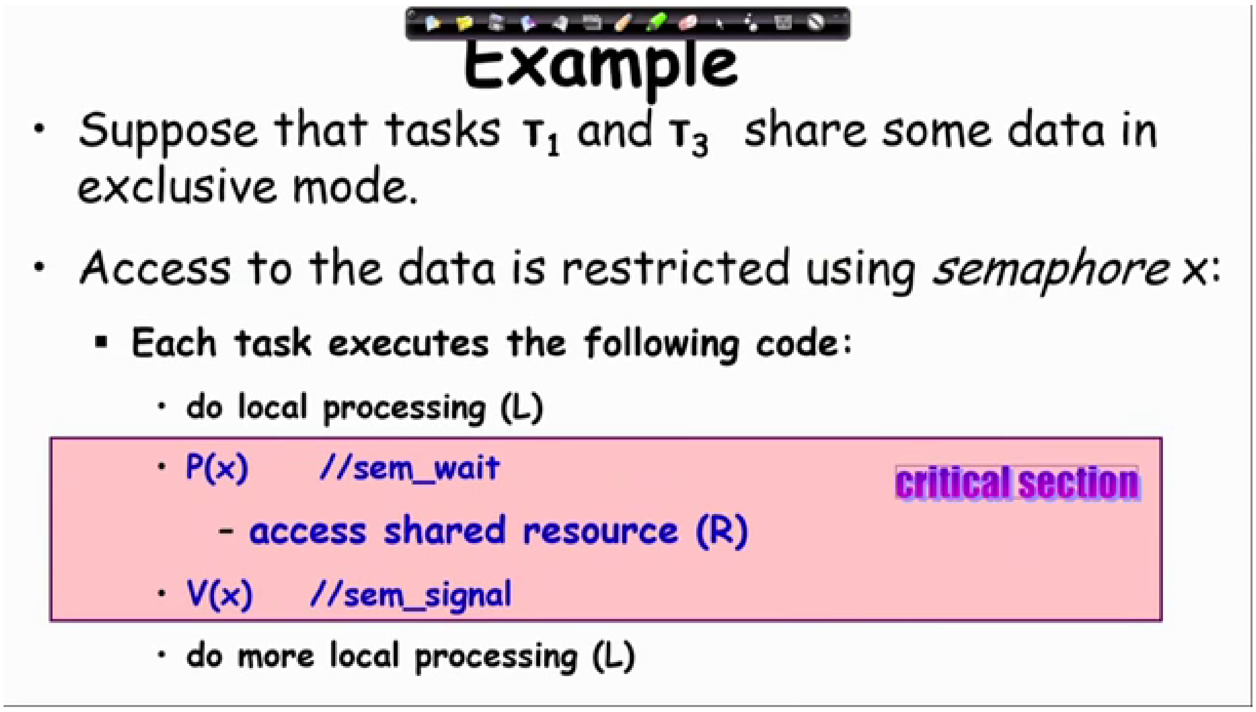
- Unless handled properly:
  - Priority inversions by a critical task can be too many causing it to miss its deadline.
- Most celebrated example:
  - Mars path finder

178

Let me repeat again that unless a designer and application real time application developer handles the unbounded priority inversion problem properly then there will be too many priority inversions and a critical task can miss its deadline.

In that past there have been many examples where the designers could not adequately take care of these and resulted in system failure. Out of these many examples possibly the most celebrated example is the mars path finder. Let us see what happened in the mars path finder.

(Refer Slide Time: 25:30)



Example

- Suppose that tasks τ_1 and τ_3 share some data in exclusive mode.
- Access to the data is restricted using *semaphore* x:
  - Each task executes the following code:
    - do local processing (L)
    - P(x) //sem_wait
      - access shared resource (R)
    - V(x) //sem_signal
    - do more local processing (L)

critical section

But before we look at what happened in the mars path finder let us just understand some simple concepts associated with a critical section, semaphores and priority inversion. Let us assume that two tasks T1 and T3 are sharing resource a non-preempt able resource which then need to execute in the exclusive mode and this is the summary of the code of the processes the task T1 and T3 where they do some local processing and then they wait for the semaphore. And once they get access to the semaphore they access the shared resource and then finally, they release the semaphore the two primitives that are supplied by the operating system for accessing the resource are semaphore wait in typical operating system book. So, it is denoted by the operation P x and V x is the semaphore signal to release the resource and this part of the code is called as a critical section of the code and then later they may do more local processing.

We are just going to complete this lecture at this point of time do not have much time left. We will stop here and from this point we will continue the next lecture.

Thank you.