

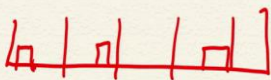
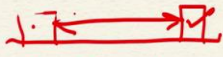

**Real Time Systems**  
**Professor. Rajib Mall**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Kharagpur**  
**Lecture No. 25**  
**Handling Task Jitter and Precedence Ordering**


Welcome to this lecture, we have been discussing about the rate monotonic scheduler and various issues that arise the rate monotonic scheduler, we looked at how to handle aperiodic tasks, how to handle sporadic tasks, how to handle the tasks if they have harmonically their periods are related, and so on. We also looked at how to handle the situation when the number of priority levels are insufficient that is the number of tasks or more than the number of priority levels.


Now, let us look at another important issue when using the rate monotonic scheduler, namely, the task jitters. And how do we handle the task jitters?

(Refer Slide Time: 1:07)

**Task Jitter: How to Deal With It?**

- **Task jitter:** 
- Magnitude of variation in arrival or completion times of a task. 
- **Certain applications require jitter be minimized as much as possible.** 
- For example: A real-time controller can become unstable in presence of jitter.





Let us start discussing about the first question that comes to mind is what is the task jitter? And why should you be concerned about it? Now, let us try to understand the jitter and how it arises. And then we will discuss how to deal with it. The task jitter is the magnitude of variation in arrival or completion time of a task.

We know that in a task  $T_i$  during its different invocations can get scheduled at different times during the interval sometimes it will get scheduled at the beginning sometimes towards the end, sometimes towards the absolute end. And the reason we have this is that different tasks have different periods and then some tasks have higher priority.

And therefore the task may not start executing immediately if there are higher priority tasks at that time. And the worst case, response time for a task occurs, it is in phase with the highest priority tasks. And the response time will be good if it is out of phase, it is higher priority task.

Now that is one of the reasons for task jitter. But there are other reasons for task jitter, let us try to find out why there is a variation in the arrival and completion time of a task. We call task jitter. There are two types of jitter. One type of jitter is the variation in the arrival time of tasks. And the other is the variation in the completion times of the task.

The reason that we are giving about the rate monotonic scheduler scheduling a different instance during the time period that would affect the completion time of a task. Sometimes it is completing early sometimes it is completing late and so on within the deadline of course, but then sometimes way before the deadline and sometimes towards the end of the deadline.

Similar is the case see the arrival time. And there are some applications which are very sensitive to the jitter. They want the jitter to be minimized as much as possible. Otherwise, if the jitter keeps on happening, that sometimes it is arriving early, and next time it is arriving too late and again early and so on, then the application can fail.

One example of that is a real time controller. The controller becomes unstable in the presence of jitter because sometimes it produces the response too early. And then the action that occurs in the environment. Next time and let us say sometimes the jitter due to the jitter the action that occurs in the environment may be as wide as this or it can be as small as this due to the jitter sometimes it completed very late and sometimes it had completed late and started early.

So that is the worst case you are the action could occur for very small time. And this situation makes the system unstable and needs to be avoided real time controller and there are many other examples where jitter is not tolerated even in a video transmission and so on jitter is a problem. So, how do we deal with jitter?

(Refer Slide Time: 5:27)

**Types of Jitter**

- Task completion jitter:
  - Arrival time jitter of the task
  - Arrival time jitter of higher priority tasks
  - Variation in task computation time due to multiple code paths.
  - Caches, bus locking, interrupts
- Response time jitter:
  - Non-preemptible critical sections
  - When exactly the task instance occurs, etc

The slide includes a diagram showing a horizontal timeline with four red arrows pointing upwards, representing task completion times. A red bracket below the first two arrows indicates a period of jitter. Another red bracket below the last two arrows indicates a period of jitter. A small inset image of a man in a white shirt is visible in the bottom right corner of the slide.

Let us first understand the types of the jitter and what causes this? One is the most important is the task completion jitter now, what are the reasons behind the task completion jitter one is the arrival time jitter of the task sometimes the task arrives early sometimes it is too late and naturally the completion time it will show up in the completion time.

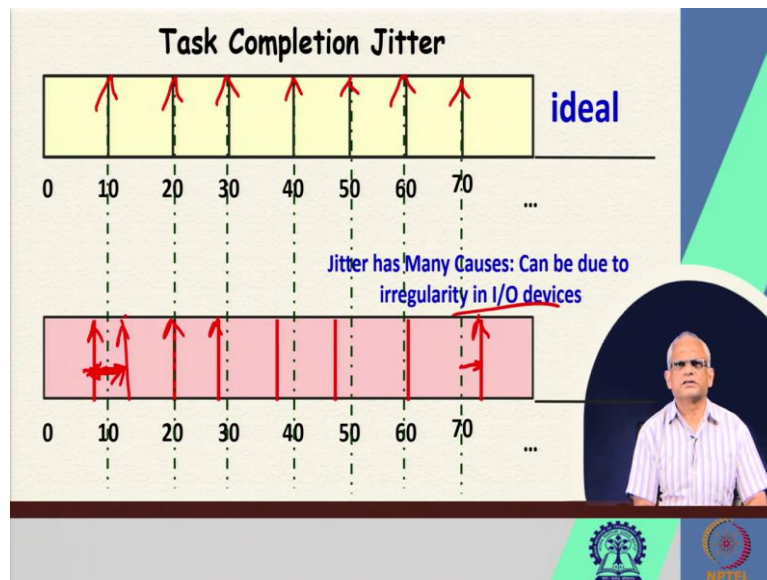
The arrival time jitter the higher priority tasks because if the higher priority task of jitter and they arrive coinciding with this task, then these task completion time will get affected, variation in task computation time due to multiple code paths; jitter can also arise because sometimes due to some other events or some state of the system a small code path is taken and sometimes the longest code path through the task is taken and the execution time of the task varies depending on some parameters and conditions.

And if the execution time of the task itself varies, then that will show up as completion time jitter and there are hardware issues. Like sometimes, interrupts may occur and the task gets disrupted, there may be bus locking due to transfer of data, the caches especially in a multiprocessor system, the caches sometimes they are lightly loaded, sometimes they're highly loaded and the response time maybe there is a jitter.

So, all these constitute the task completion jitter. Now, what about the response time jitter? The completion time jitter is that the ideal is that it should complete at nice periodic manner, the response the task completion time, but the response time jitter is starting with the arrival of the task, how long does it take to complete, sometimes it takes too long, sometimes it is too short and so on.

So, what causes the response time jitter? The response time jitter is caused by non-preemptible critical sections. Sometimes the task needs to wait for other tasks, which are using certain critical section and that will show up as the response time jitter and when exactly the task instance or etc. These reasons for the response time jitter.

(Refer Slide Time: 8:42)



Let us understand the task completion jitter through this simple diagram that for a real time control application, we want the controller to complete their action at a nice, well-spaced evenly spaced intervals 10, 20, 30, 40 etc. But then the completion time jitter causes sometimes it to complete earlier than 10 sometimes 20 and sometimes much later than the time it was expected to complete.

So, the jitter here is the earliest it can complete and the latest it can complete. So that constitutes the jitter time. The latest from the ideal time and the earliest from the ideal time. So, that interval is called as the jitter the latest here it is the latest is this and this the earliest and that is the jitter and we want to minimize the jitter.

We had seen that jitter has many causes due to the scheduler itself the rate monotonic scheduling, the way it works, the execution time of the task itself depending on the state of the system, the arrival time jitter of the task, arrival time jitter of the higher priority tasks the completion times of the higher priority tasks like sometimes the higher priority tasks take too much time to complete because of some environmental parameter setting, sometimes it completes early.

So, all those so up is task completion jitter and sometimes it may be due to the irregularity in IO devices.

(Refer Slide Time: 10:58)

**Dealing With Task Jitter: Approach 1**

- Applicable in situations where:
  - Set of tasks is highly schedulable
- In this case, if one or two tasks have high jitter requirements.
  - Assign these tasks high priorities

Handwritten annotations:  $U = 0.4$  and  $LL = 0.72$  are circled in red. A diagram shows a task  $T_i$  with a period  $T_i$  and a jitter  $J_i$  indicated by a red arrow.

How do you deal with task jitter? There are two main approaches we look at the first approach here; the first approach is suitable when the task set is highly schedulable. That is, the Liu Leyland bound is let us say 0.72. And our task set is operating at let us say 0.4 the utilization of the task set is 0.4 but the utilization bound due to the Liu Leyland criterion, Liu Leyland criterion is 0.72, but the actual utilization is 0.4.

And then we say that the task set is highly schedulable small changes in the task completion times etc. will not lead to deadline misses. So, in this situation, we can minimize the task jitter by using an approach which we are going to describe. So here, we find the tasks which are most sensitive to the jitter.

And the tasks which are most sensitive to the jitter we assign those tasks very high priority maybe one or two tasks, we increase their priority and we know the period transformation technique, we can use the period transformation technique to improve the to increase the priority of these tasks.

And then we will see that the jitter of these comes down. Now what is the reason why the jitter of these will come down? Because the jitter of the higher priority tasks will not go into affect them because these are themselves high priority. So, if the task was operating at some priority level here, and there are other tasks, and now we increase the priority level of task  $T_i$  to here, then the tasks that are in between they are not going to affect or they cannot cause

jitter for  $T_i$  only few tasks which are higher priority, their variation in execution time their variation in arrival and so on. They can cause jitter to this.

And also, if there is a complete arrival time jitter of  $T_i$ , they may get adjusted at the expense of the lower priority tasks. So, the jitter is definitely going to reduce, it is not going to become 0. But it is definitely going to reduce and this is a simple technique by which you can reduce the jitter of one or two tasks when the system is highly schedulable.

We want the system to be highly schedulable because we are violating the rate monotonic principle here. And the task set utilization is low. And therefore if we do this increase in the priority, then the schedulability of the task set is not going to get affected.

(Refer Slide Time: 14:29)

**Dealing With Task Jitter: Approach 2**

- Assume a task set is barely schedulable.
- Each task with jitter constraint is split into two or more subtasks ( $T_{i.1}, T_{i.2} \dots T_{i.n}$ ):
  - One which computes the output,
  - One which passes the output on, etc.
- Set last task's priority to very high values:
  - Period is the same for all tasks

The slide includes a diagram of a task execution timeline. A dashed box labeled '10' contains three subtasks labeled 'ST1', 'ST2', and 'ST3'. A red arrow points to 'ST3' with the number '2,3' written above it. A small inset video shows a man speaking.

Now what do we do if the task set is barely schedulable? How do we deal with jitter? We cannot just increase the priority of the task to high values because these are barely schedulable and if we do that, then some of the tasks are definitely going to miss the deadline. Here in the approach two, we split the task into two or more sub tasks like this. So, this is sub task one, sub task two, sub task three and this is the task and there are some subtasks which takes the input some computes based on the input some produces the output and so on.

Now, here the period of the all the tasks are the same, but, we raised the last subtasks priority to very high value. So, the all these are let us operating at a priority level 10. And then we let the priority level of the last task to be two or three depending on what are the priority level of the highest priority task one two, etc.



We find a suitable priority level and increase the priority of the last task. So, that even if there is small jitter here, this is going to get adjusted here. So, only a small part of the task we are increasing the priority level and this is the approach recommended when the task set is not highly schedulable.

(Refer Slide Time: 16:30)

**Precedence Relationships Among Tasks**

- How to handle precedence relations in event-driven schedulers:
  - Do not make a task ready until its predecessors have completed.

```
graph TD; T1((T1)) --> T2((T2)); T1 --> T3((T3)); T2 --> T5((T5)); T3 --> T4((T4)); style T4 stroke:#f00,stroke-width:2px
```

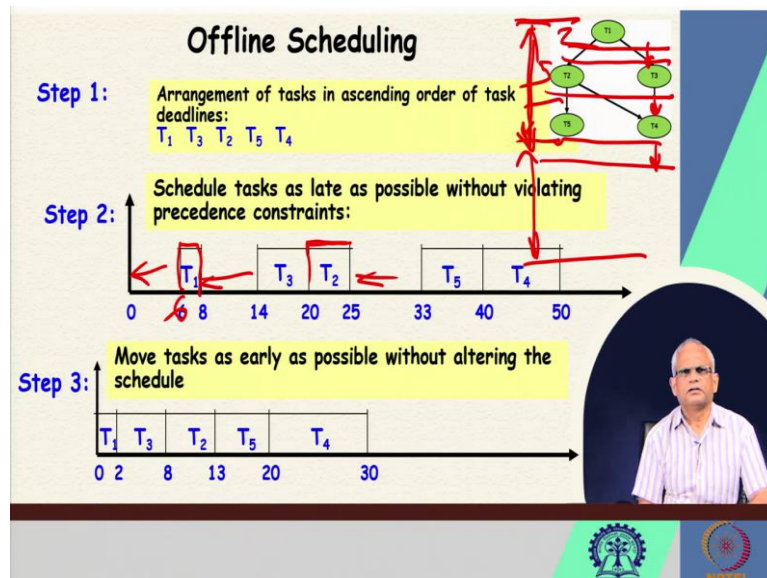
The slide also features a small inset image of a man in a white shirt and glasses, and logos for IIT Bombay and IIT Madras at the bottom.

Another issue that we might have to handle in the rate monotonic scheduler framework is handling precedence relationships among tasks. Sometimes, we have tasks where one task depends on the result of another task or one task can act only when another task has completed we can represent the task in the form of a directed acyclic graph and this task these all these tasks they will repeat with some period because this is depending on this this is depending on this and therefore, they will repeat with some period  $P_k$  let me just say.

So, in the next interval again we have to do this that the task one must complete first and then only T2, T3 can execute and after that T5, T4 can execute in the event driven scheduler, this is rather simple to handle. In the Event driven scheduling, we make a task ready only when the predecessor tasks have completed.

For T4 becomes ready for execution only when T2 and T3 have completed. So, we need the small modification to the scheduler, but that would require specifying the tasks on which it depends and then it will make it ready only after those tasks are completed. So, if this facility is supported by the scheduler, we can handle precedence relationship among tasks.

(Refer Slide Time: 18:38)



But how do I handle this in offline scheduling? We will again take the same example that this repeats with some period in the next interval, the same set of tasks will also have to be executed an offline scheduler like cyclic scheduler, how do we develop the schedule so, that the precedence constraints are met.

The first thing is we arrange in terms of the deadline  $T_1$  as  $T_1, T_2, T_3, T_4$ , and  $T_5$  etc. They have their deadlines and we arrange the tasks in order of the deadlines. So,  $T_1$  needs to complete here let us say  $T_2, T_4, T_5$  etc. So, this the deadline for this is the deadline for this and this the deadline sort of steadily because the overall deadline we need to split into sub deadlines and we arrange them in ascending order of that deadline.

Now, the second step is that we schedule a task as late as possible without violating that deadline, let us say the task  $T_1$ 's deadline is 8, and it needs 2 units of time to execute. Then we will schedule it exactly at 8 and 2 units here. So, it will start at 6. Now let us say  $T_2$ , the deadline is 25. Now we will schedule  $T_2$  the latest. So, it completes exactly at 25.

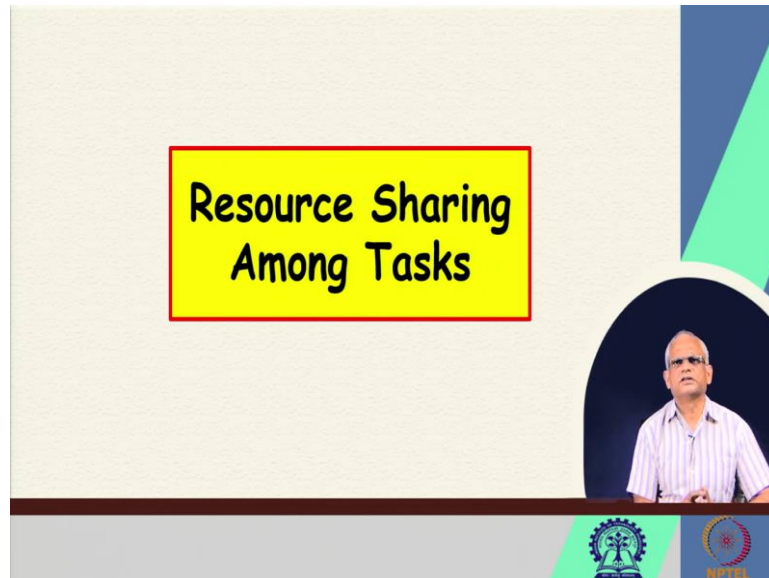
And then it takes 5 units, let us say  $T_2$  takes 5 units, and needs to complete by 25. So, we schedule at 20. Now  $T_3$ , let us say, should complete by 25, and takes 6 units of time. So, from before 20, we assign 6 units, so it becomes 14.  $T_4$  let us say completes at 50 and takes 10 units of time, so it will execute within 40 to 50.  $T_5$  takes 7 units time and let us say the deadline is 45.

So, between 33 to 40, we can assign so all the tasks have been assigned, so that they complete as late as possible. And once we have formed this, we just shift all of them forward. That is a



step three. So,  $T_1$  instead of 6, we assign it to 0 and completes at 2. Now,  $T_3$  can be scheduled at 8 completes at 8 and so on. And here we can see that all the precedence ordering is met.

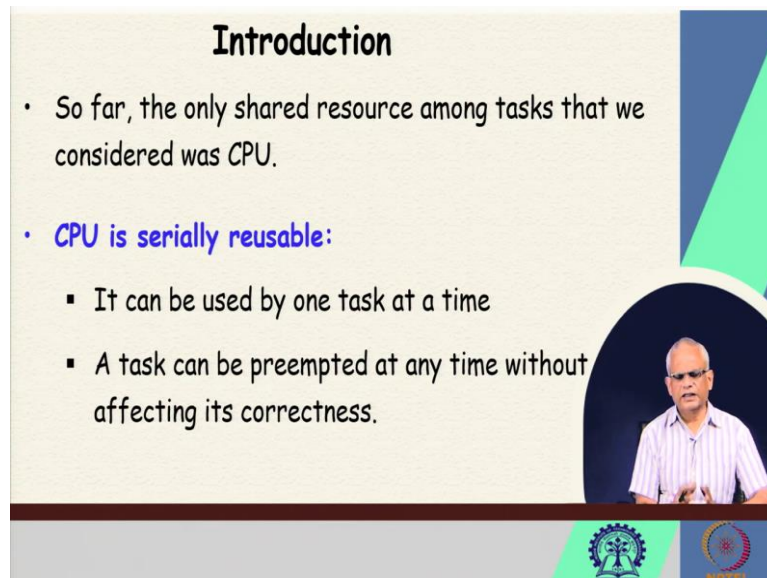
(Refer Slide Time: 22:06)



Now, we have completed all our discussions compared to the simplified rate monotonic scheduler, where we had ignored about resource sharing among tasks. We had assumed that all tasks are independent, but in a realistic situation. Tasks do share resources. For example, they share some memory locations where one task updates, the other task reads, or maybe multiple tasks update that memory location.

So, how do I support this resource sharing among tasks in a rate monotonic scheduling environment? Now that is our discussion. And this is a very important topic, because if we do not take care about how the tasks can share resources, then there can be deadline misses.

(Refer Slide Time: 23:06)



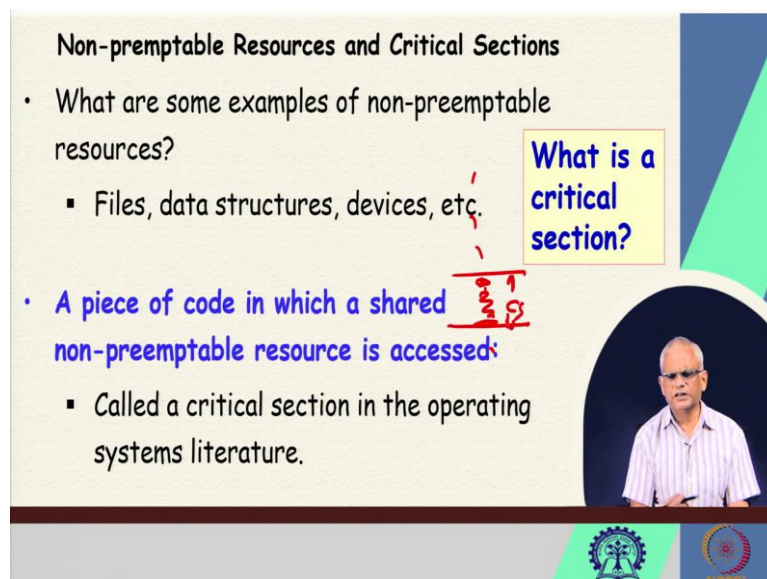
### Introduction

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

So far, the tasks were all independent, that sort of we had assumed the only resource they shared among themselves was the CPU. But then, CPU is serially reusable, what it really means is that it is used by one task at a time, but then the task can be preempted. And it can again start using so it is serially reused without causing much problem.

But there are many resources, which are not serially reusable, we cannot reuse we cannot just preempt and the task comes and reuse unlike the CPU. The correctness of the result can be affected.

(Refer Slide Time: 24:01)



### Non-preemptable 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?**

*Handwritten annotations: A red bracket underlines 'shared non-preemptable resource' and a red arrow points from the question box to it.*

These are called as the non-preemptable resources. And the critical sections let us understand the non-preemptable resources and critical sections. Initially, we will just discuss for one or

two minutes, which is the basic of this resource sharing is there in the traditional operating system looks for one or two slides.

And then we will discuss aspects which are specific to real time tasks sharing resources. One question is that what are some examples of non-preemptable resources? We have several examples of non-preemptable resources. These are not serially reusable, for example, files, data structures, devices, we cannot just preempt one task at any time, and another task, which is it and the other tasks, comes and starts using it.

Here, a task needs to complete its usage before releasing we cannot just preempt it otherwise, the result will be wrong, the system will fail. So that is a non-preemptable resource like file data structure and device. But are these the critical section, what is the critical section a critical section is actually a piece of code in the program in which a shared non-preemptable resource is being used.

So, if this is the code, then in some part of the code, we use the critical resource, we get the resource and do some processing and release the resource. So, this is the critical section of the code. The critical section is a part of the code where we use some non-preemptable resource.

(Refer Slide Time: 26:17)

**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

The slide features a video inset of a man in a striped shirt speaking. There are handwritten red annotations: a vertical line with a horizontal bar through it, and a red circle around the text 'Unbounded priority inversion'. The slide also has a decorative blue and green geometric shape on the right side and logos at the bottom.

And here, the idea is that in the critical when the program is in its critical section, if this is the critical section of the program, the program can be preempted from whatever it is doing, when it is not in its critical section, but when it is in critical section, we cannot just preempt it from using its resources.

And then if we do that, that will lead to incorrect result the traditional operating system solution to critical section problem is to use semaphores, that it has accurate resource. And it really just resources other tasks wanting to use it, they will have to wait. But unfortunately, the semaphores do not work well in the real time situation.

The main reason is that it causes priority inversion. The priority inversion problem is that a low priority task is executing at the expense of a higher priority task, we know that according to rate monotonic scheduling, whenever there is a higher priority task which is ready a low priority task should not execute which should be the highest priority task executing.

But in a priority inversion situation, the high priority task cannot execute. And the low priority task executes that we call as a priority inversion. The simple priority inversion is not such a big problem, actually. But what is the big problem here we face is the unbounded priority inversion. This is a very serious problem.

It can cause even the most conservatively designed a system to fail its deadlines; the system will fail unless we pay particular attention to the unbounded priority inversion. So we will first understand what is this unbounded priority inversion? How does it occur? And then we will see what are the solutions to unbounded priority inversion the semaphore does not prevent unbounded priority inversion.

We will see some solutions to the unbounded priority inversion and how to handle this. We are at the end of this lecture. And we will continue from this point, we first discuss about what exactly is a priority inversion. How does it arise? And can the simple priority inversion be handled easily? And when does the unbounded priority inversion occur? And how to handle this? What are the solutions and so on that we will discuss in the next lecture? We will stop here. Thank you.