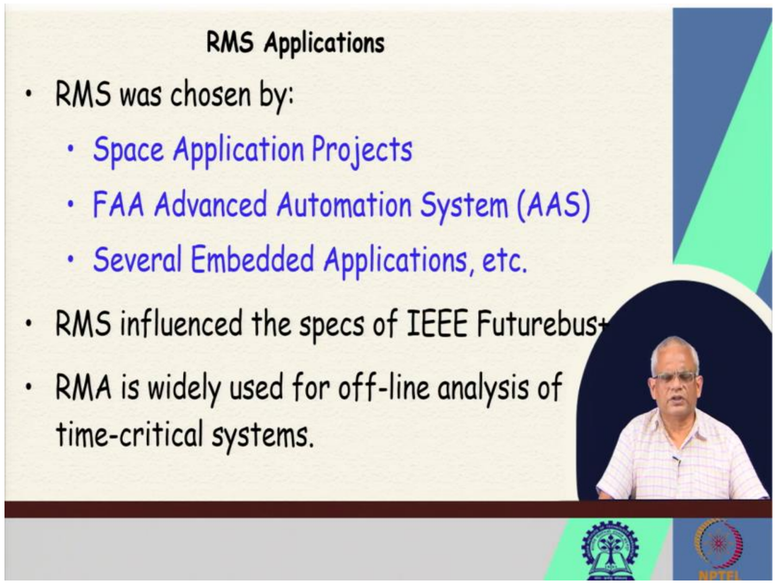
Real Time Systems
Professor Rajib Mall
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture 20
RMS Generalizations

Welcome to this lecture. In the last lecture, we had looked at the completion time criterion, important design criterion for real time systems using the rate monotonic scheduler, whether a given task set is schedulable if it fails the Liu Layland criterion, we had discussed about the Liu Lehoczky's completion time criterion and then we had compared EDF and rate monotonic scheduler and so on. Now, let us continue from that point onwards.

(Refer Slide Time: 0:54)



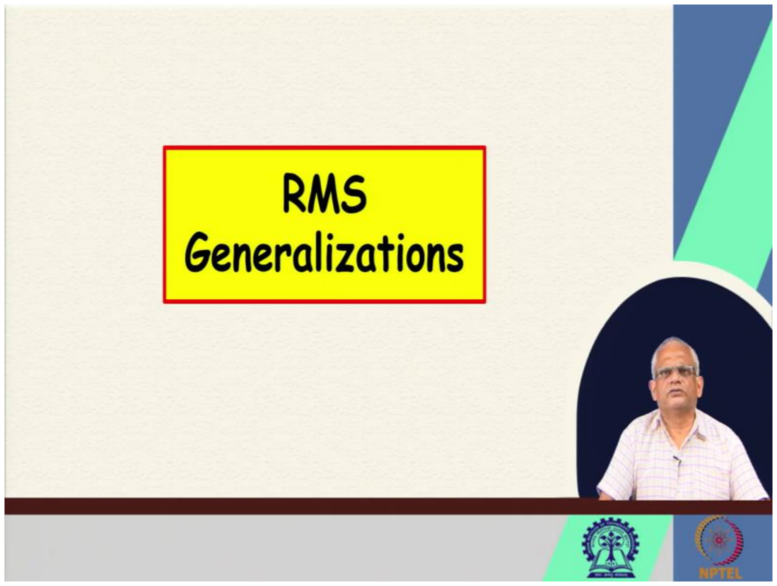
RMS Applications

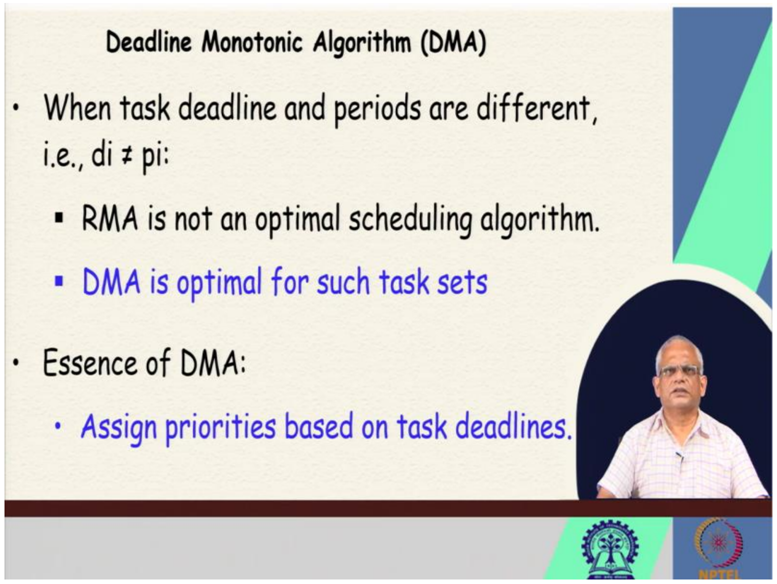
- RMS was chosen by:
  - Space Application Projects
  - FAA Advanced Automation System (AAS)
  - Several Embedded Applications, etc.
- RMS influenced the specs of IEEE Futurebus+
- RMA is widely used for off-line analysis of time-critical systems.

Let us look at some of the popular applications of the rate monotonic scheduler. The rate monotonic scheduler was chosen by many space application projects. And the reason is that it is efficient, sturdy, runs in very small systems, the scheduler is simple, efficient. It was also chosen by the Federal Aviation Agency advanced automation systems, used in a vast number of embedded applications.

And it also designed, it also influenced the design of the real time communication protocol, the future bus protocol and is widely used for offline analysis of time critical systems. So, no doubt it is a very important scheduler, the rate monotonic scheduler.

(Refer Slide Time: 2:05)


RMS Generalizations


Deadline Monotonic Algorithm (DMA)

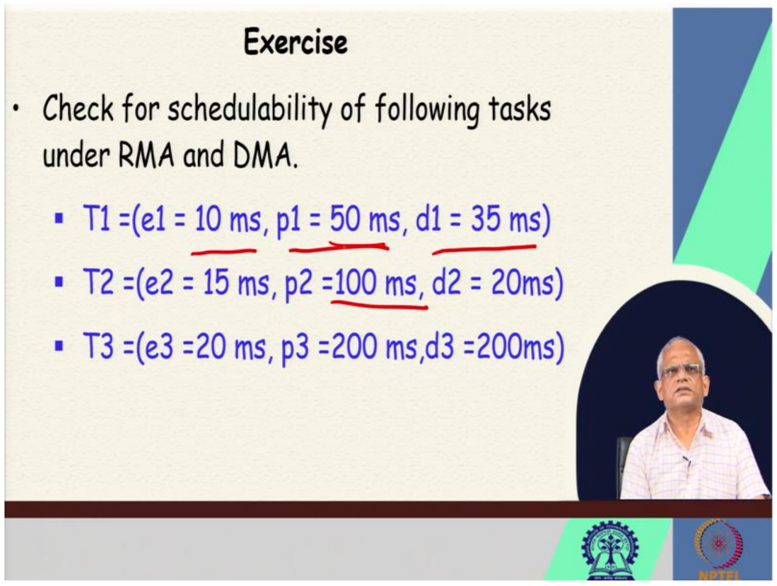
- When task deadline and periods are different, i.e., di ≠ pi:
  - RMA is not an optimal scheduling algorithm.
  - DMA is optimal for such task sets
- Essence of DMA:
  - Assign priorities based on task deadlines.

Now, let us look at some generalizations in the rate monotonic scheduler, some issues that we did not consider, we will consider those issues. The first issue that we discuss is the variation of the rate monotonic scheduler called as the deadline monotonic scheduler or the deadline monotonic algorithm DMA. So far, we have assumed that the period and the deadline for the tasks are same, the period is the same as the deadline and that is true for a vast majority of tasks in many applications.

But then, there may be few tasks that we occasionally come across for which the period and deadline are different, the deadline may be less than the period or the deadline may be slightly more than the period. And for those cases, the rate monotonic scheduler is not the optimal scheduling algorithm. The deadline monotonic scheduler is optimal. In the deadline monotonic scheduler, we do not assign priorities based on the task periods.

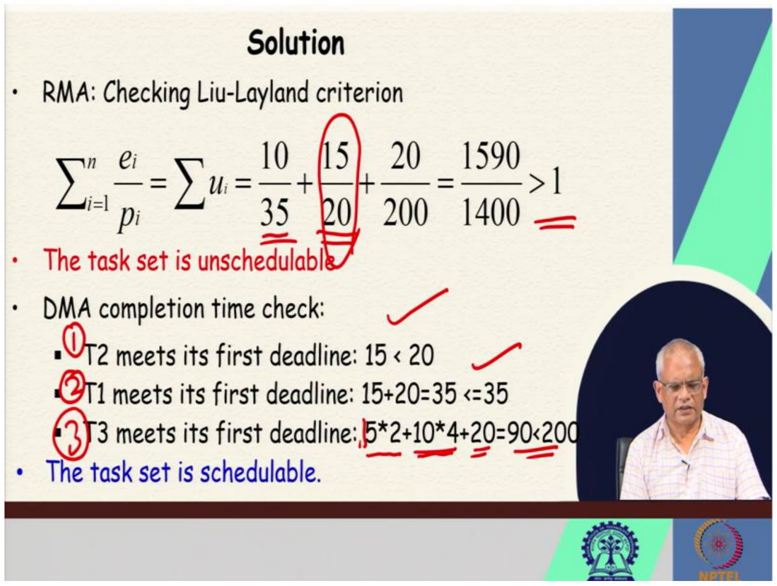
Remember that in the rate monotonic scheduler, we had said that the rate of arrival, the higher is the rate of arrival, or the lower is the task period, the higher is the priority assigned to it, the task with the lowest period is assigned the highest priority. But in the DMA, we do not consider the period we look at the deadlines of the tasks. And we assign priorities based on the task deadlines. The task having the subtest relative deadline is assigned the highest priority. So that is the variation here.

(Refer Slide Time: 4:25)



Exercise

- Check for schedulability of following tasks under RMA and DMA.
  - T1 =(e1 = 10 ms, p1 = 50 ms, d1 = 35 ms)
  - T2 =(e2 = 15 ms, p2 =100 ms, d2 = 20ms)
  - T3 =(e3 =20 ms, p3 =200 ms,d3 =200ms)



Solution

- RMA: Checking Liu-Layland criterion

$$\sum_{i=1}^{n}\frac{e_i}{p_i}=\sum u_i=\frac{10}{35}+\frac{15}{20}+\frac{20}{200}=\frac{1590}{1400}>1$$

- The task set is unschedulable
- DMA completion time check:
  - ① T2 meets its first deadline: 15 < 20
  - ② T1 meets its first deadline: 15+20=35 <=35
  - ③ T3 meets its first deadline: 15*2+10*4+20=90<200
- The task set is schedulable.

Now let us check the schedulability of the task set using a deadline monotonic scheduler and the rate monotonic scheduler. And in this task, the period of the task is 50. But the deadline is much before that the relative deadline is 35. And the execution time is 10. And there is another task whose period is 100 but the deadline is 20 and the third task, deadline and period are the same 200. Now is this task set schedulable under the rate monotonic scheduler.

In the rate monotonic scheduler, let us check the Liu Layland criterion. First let us consider the sum of utilization, the sum of utilization if you check here because it needs to complete by 35 so written 10 / 35 and a straight application of the Liu Layland criterion, 20 is the deadline we have written here deadline because it needs to complete by that time and we see that it is greater than 1. So, we say that under the rate monotonic scheduling, the tasks set will miss its deadline.

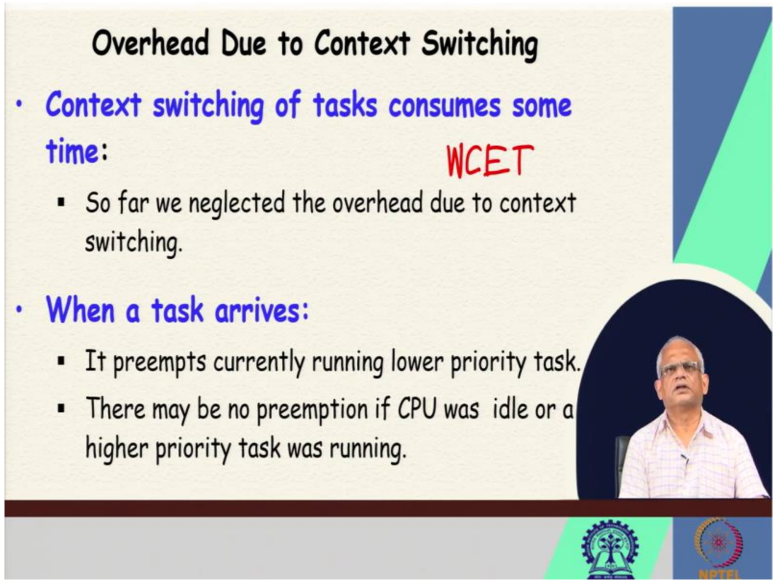
But what about the deadline monotonic scheduler, in the rate monotonic scheduler we will assign based on okay, so, this is the schedulability for the utilization due to the different tasks. Now, let us we infer that the task set is unschedulable. But now, let us consider the deadline monotonic scheduler, here T2 is the highest priority task and by using the completion time theorem, it will meet its deadline because 15 < 20, 15 is its execution time and there are no other high priority tasks.

So 15 < 20 and T1 is the second highest priority task and T2, which is the highest priority task will arise only once during the execution of p1 sorry T1 and therefore, it is $15 + 20 = 35 \leq 35$.

So, you might ask that why does it occur only once because the period of this is 100 and the period of T 2 is 100 and the period of T 1 is 50. So, we p 2 can occur at most once in T 1.

So that is the reason that the T 1 will also meet its deadline. So, T 2 is the highest priority task T 1 is the second higher priority task and T 3 is the lowest priority or the third priority task and that also meets its deadline because at most two times the highest priority tasks can occur 15 * 2 and 10 * 4 the second, the highest priority tasks can occur 4 times, second highest priority tasks can occur 4 times and then itself consumes 20 times, 30+40+20 < 200. So all the 3 meet their deadline under the deadline monotonic scheduling.

(Refer Slide Time: 9:11)



Overhead Due to Context Switching

- Context switching of tasks consumes some time:
  WCET
  - So far we neglected the overhead due to context switching.
- When a task arrives:
  - It preempts currently running lower priority task.
  - There may be no preemption if CPU was idle or a higher priority task was running.

Now let us consider context switching because the task execution times are very small typically few milliseconds only 10 milliseconds and so on. And the context switching overhead can be significant. For example, it may be like 1 second. So in 10 millisecond if we always ignore 1 millisecond, we might be doing a mistake. We might find that theoretically, the task set is schedulable.

But then when you actually run it may not be schedulable, it misses its deadlines. And the reason may be that we did not consider the context switching overhead. So, in the schedulability analysis, when the execution time of the tasks is comparable to the context switching time, we need to consider context switching time, which we had so far ignored. Now let us see how do we consider the context switching time in the schedulability analysis.

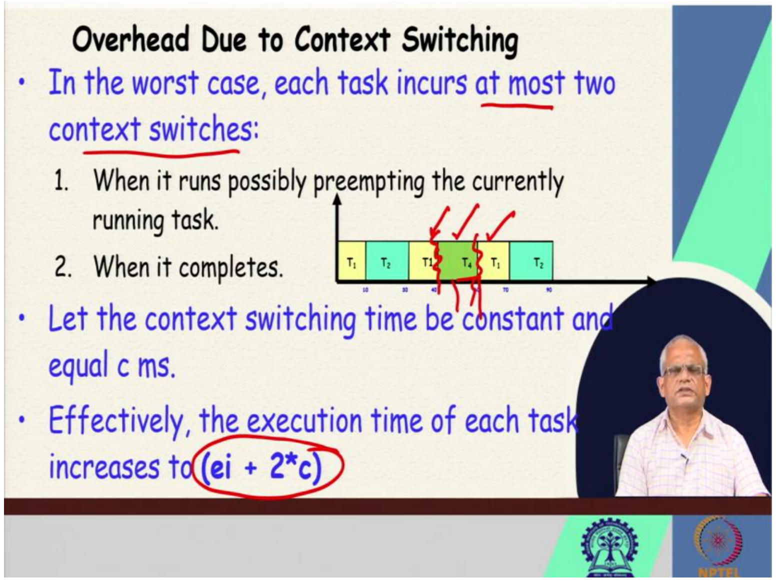
When a task is preempted a higher priority task, there is a context switching time, because the context of the task which is already running needs to be saved. And the context for the higher priority task needs to be loaded. And that is significant. Considering that the tasks that are executed are very small. For example, just read a sensor and just process it, maybe 5 millisecond for that, but then the context switch may take 1 millisecond.

So we cannot really ignore 1 millisecond compared to 5 milliseconds. And the context switching occurs when a task arrives; it preempts the currently running lower priority task if there is a lower priority task running. But the thing is that if there is no preemption or if there is a higher priority task running already, then the task on arrival does not cause preemption. So you need to consider that. But just remember that all the analysis that we are doing.

The schedulability analysis is based on the worst case scenario. For the task set to compute the execution time, we will compute the worst case execution time, or the WCET, the worst case execution time. And I was saying the other day, that to compute the worst case execution time, we run the task on the same processor, under various situations with different events and different workloads and so on, and find the worst case performance.

And also, while considering the context switch, maybe sometimes two context switch may not happen, maybe just one took place. But since sometimes two can take place, we need to consider two, because that is the worst case. So the overhead due to context switching, we need to consider the worst case context switching that will assume that it always preempt the currently running lower priority task.

(Refer Slide Time: 13:11)



Overhead Due to Context Switching

- In the worst case, each task incurs at most two context switches:
  1. When it runs possibly preempting the currently running task.
  2. When it completes.
- Let the context switching time be constant and equal c ms.
- Effectively, the execution time of each task increases to $(ei + 2^*c)$

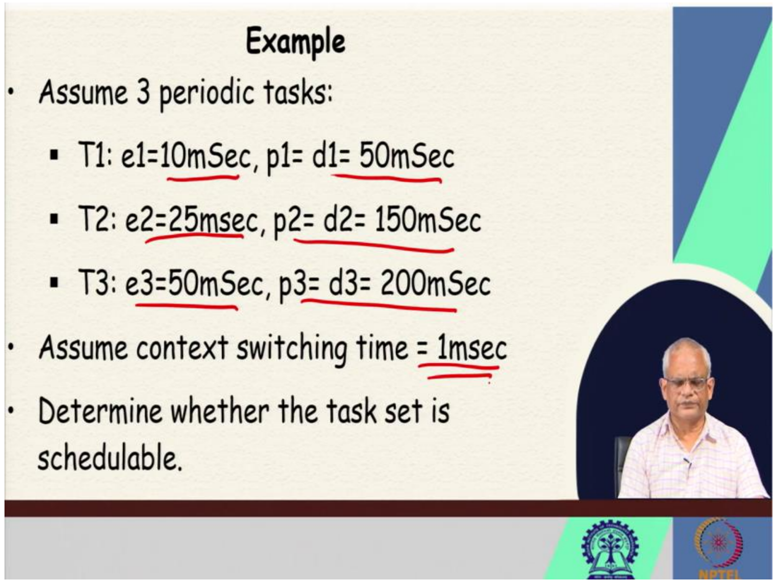
Now, the important statement here we are making is that in the worst case, each task incurs at most two context switches. So that is the worst case. But we will consider the worst case we will design our system for the worst case, so that even when the worst case occurs, our system will still continue to function. So the very important statement here is that in the worst case, each task incurs at most two context switches. Now why is that? Let us just reason that out. Now, let us consider any task here.

Let us consider T4 now, let us consider that T1 was running which is a lower priority task and let us say T4 is a higher priority task. Now it preempted T1 at this point as T4 became ready it preempted T1. So, there is a context switch here. And as soon as T4 completed, again T1 will start running and therefore there is another context switch. So, this is one example that there is two context switches, and you can consider various examples. And you will see that at most there are two context switches.

You may have considered that T4 is again preempted by another task and so on. But finally, you will see that every task introduces two context switches. The two context switches are when it preempts a lower priority task, when it becomes ready, it preempts a lower priority task and when it completes, there is a context switch. And if you want to check out whether there is still a higher priority task T5, which preempted T4 and then T5 on completion there is a context switch, but those 2 context switches are due to T5.

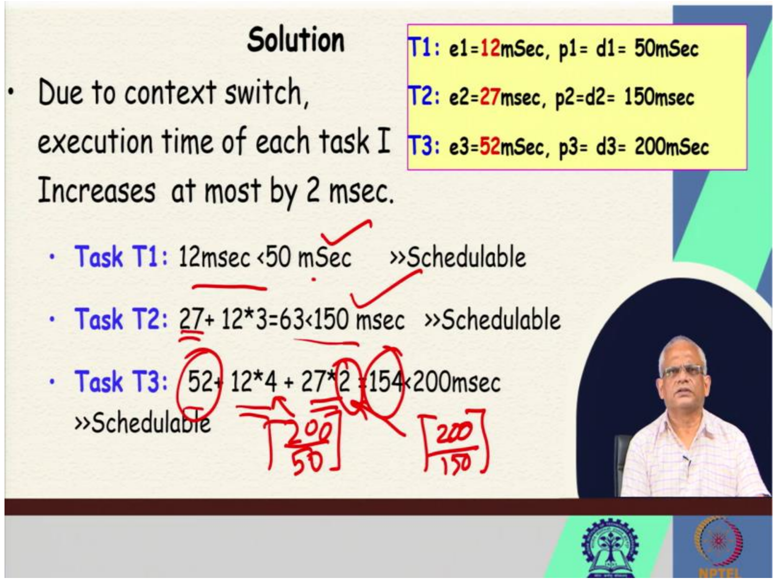
So for T4 we will incur only two context switches. So, for every task we need to consider two context switches it is as if the execution time of every task increases by 2c, if c is the context switch time, even the higher priority tasks their execution time increases by 2c and the current task its execution time increases by 2c if we do that we would have taken care of the worst case context switch overhead.

(Refer Slide Time: 16:27)



Example

- Assume 3 periodic tasks:
  - T1: e1=10mSec, p1= d1= 50mSec
  - T2: e2=25msec, p2= d2= 150mSec
  - T3: e3=50mSec, p3= d3= 200mSec
- Assume context switching time = 1msec
- Determine whether the task set is schedulable.



Solution

T1: e1=12mSec, p1= d1= 50mSec
T2: e2=27msec, p2=d2= 150msec
T3: e3=52mSec, p3= d3= 200mSec

- Due to context switch, execution time of each task I Increases at most by 2 msec.
  - Task T1: 12msec <50 mSec >>Schedulable
  - Task T2: 27+ 12*3=63<150 msec >>Schedulable
  - Task T3: 52+ 12*4 + 27*2 =154<200msec >>Schedulable

$\left\lceil \frac{200}{50} \right\rceil$ $\left\lceil \frac{200}{150} \right\rceil$

Now, let us use this idea in the example, let us say there are 3 periodic tasks T1 execution time is 10 millisecond and the deadline and period are 50 milliseconds, T2 execution time is 25 milliseconds and the deadline and period are 150 milliseconds and T3 execution time is 50 and period and deadline are 200 milliseconds and it is given that context switch time is 1 millisecond and 1 millisecond is not insignificant compared to 10 25 etc.

So, we need to consider that in our schedulability analysis. And it becomes really simple if we want to consider the context switch overhead, because we need to just increase the execution time of all the tasks by 2 into the context switch time. So, this is the task set that is given 10 millisecond and 50 25 and 150, 50 and 200.

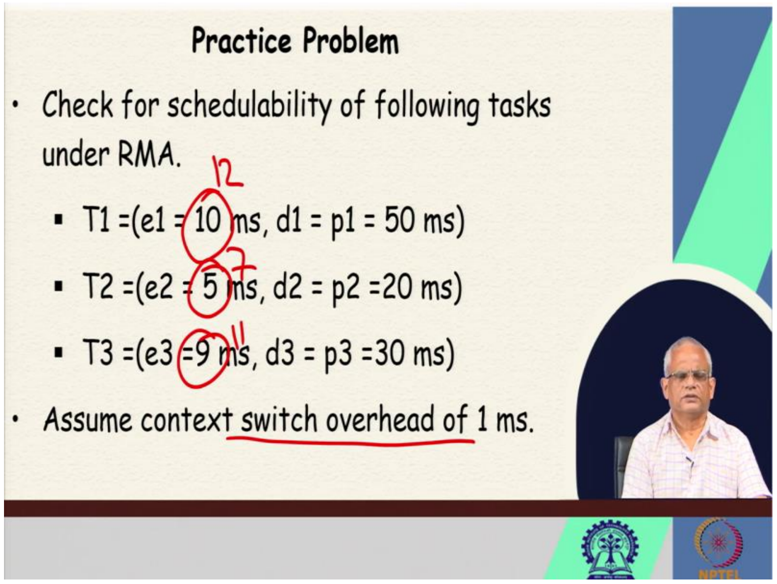
Now, the effect of the context switch can be taken care by increasing all the execution time by 2 so 10 will become 12 25 27 50 by it will become 52. So, that is what we have done increased all the execution time by 2 millisecond because the context switch over it is 1 millisecond. And this is satisfactory consideration of the injection time then we can apply our utilization bounds.

Let us straight away apply the completion time theorem for the task T1 12 < 50 schedulable, for tasks T2 27 is its execution time and the higher priority task occurs 3 times, so 27+12*3 = 63 < 150 schedulable tasks T3 for this T2 occurs 2 times, T1 occurs 4 times sorry, its own execution time is 52 and T1 occurs 4 times and T2 occurs 2 times.

So, totally 154 < 200. I hope this you are able to understand the same formula that we are applying e_i and $\lceil p_i / p_j \rceil$ and here p_i is 150 sorry, for T3 p_i is 200. And we are applying for T1 which is 50. And therefore $\lceil 200 / 50 \rceil = 4$ and for the third term here, $\lceil 200 / 150 \rceil = 2$.

So, that is how we have got this expression and we find that even after considering context switch still, it is schedulable, but we might have cases where the task set if we ignore the context switch overhead is schedulable, but once we consider the context switch overhead it is no more schedulable, there may be cases like that we have to be careful.

(Refer Slide Time: 20:18)

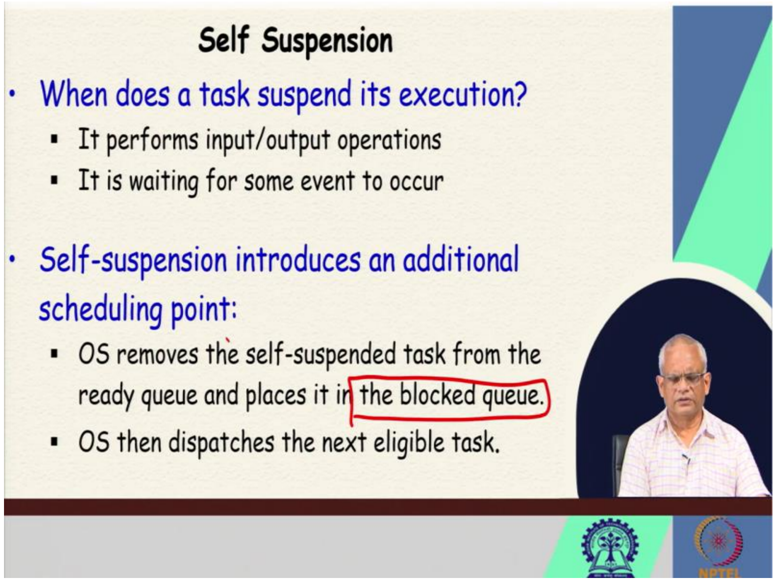


Practice Problem

- Check for schedulability of following tasks under RMA.
  - T1 =(e1 = 10 ms, d1 = p1 = 50 ms)
  - T2 =(e2 = 5 ms, d2 = p2 =20 ms)
  - T3 =(e3 =9 ms, d3 = p3 =30 ms)
- Assume context switch overhead of 1 ms.

Now, we have a practice problem here we have 3 tasks T1 T2 T3, with their execution time, deadlines and periods are the same. And there is a context switch overhead of 1 millisecond, we need to check whether this is schedulable under the rate monotonic scheduler. And to consider this, we need to increase each of the task execution time by 2 millisecond. So, this becomes 12, this becomes 7 and this becomes 11. And then we apply the Liu Layland or if the Liu Layland fails the completion time theorem. So, we will just leave this for you to work out.

(Refer Slide Time: 21:14)



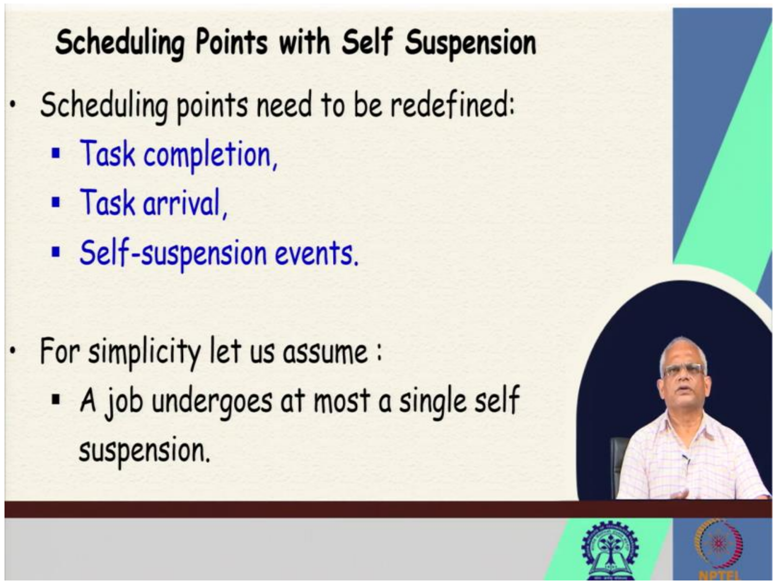
Self Suspension

- When does a task suspend its execution?
  - It performs input/output operations
  - It is waiting for some event to occur
- Self-suspension introduces an additional scheduling point:
  - OS removes the self-suspended task from the ready queue and places it in the blocked queue.
  - OS then dispatches the next eligible task.

Now, let us consider another important issue which is self-suspension. A task may self-suspend itself under various conditions. One is that if it is, task is performing some input output operation, it is reading a sensor or it is producing a command and so on. It may, we say that it is self-suspending, we will see what happens in self-suspension and also if it is waiting for some event to occur, for example, it might wait for a result from another task and then also itself suspense itself. And when it self suspends, its execution is stopped, it is put in the ready queue.

So, when it self suspends, it is put in the blocked queue. So, the self-suspension introduces adding an additional scheduling point. And once the tasks self suspends, then the next task in the ready queue is dispatched.

(Refer Slide Time: 22:42)



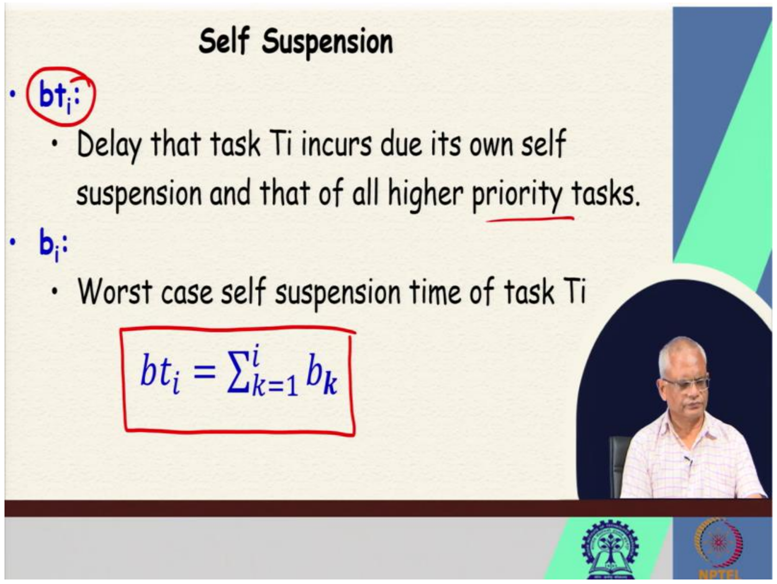
Scheduling Points with Self Suspension

- Scheduling points need to be redefined:
  - Task completion,
  - Task arrival,
  - Self-suspension events.
- For simplicity let us assume :
  - A job undergoes at most a single self suspension.

Now, we need to define, redefine our scheduling points. So far, we considered the scheduling points that are the point at which the scheduler becomes active and looks for a task from the ready queue for dispatching, we had considered task completion and task arrival as the two events on which the rate monotonic scheduler becomes active and starts the scheduling process. But when we consider self-suspension that becomes another event, the self-suspension event in which the scheduler would become active.

Now, to simplify our analysis, let us make a simplifying assumption that a task produces at most one input sorry, it reads one input or it produces one output or it waits for one event, any of this. So a job undergoes at most a single self-suspension. But we can relax this. If there are multiple instance of self-suspension, we will just need to elaborate the result that we are going to present in the next slide.

(Refer Slide Time: 24:19)



Self Suspension

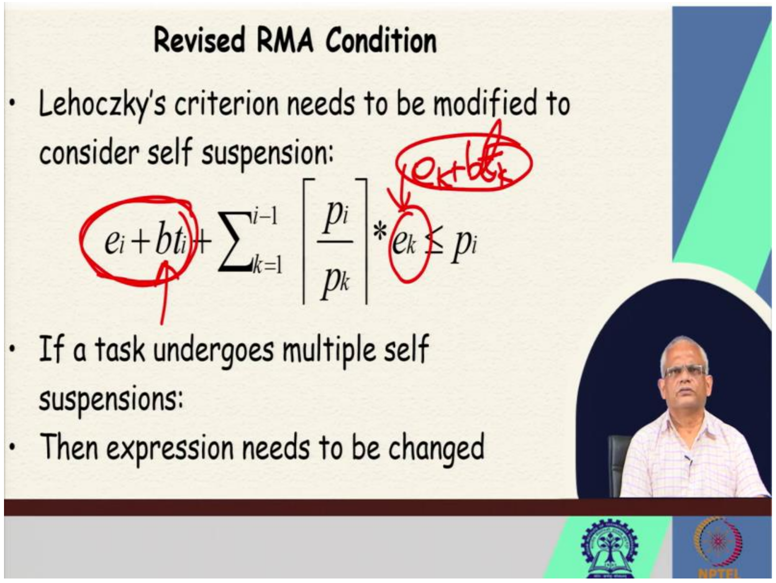
- bt_i:
    - Delay that task Ti incurs due its own self suspension and that of all higher priority tasks.
- b_i:
    - Worst case self suspension time of task Ti

$$\boxed{bt_i = \sum_{k=1}^{i} b_k}$$

Let us denote the term bt_i, it is the delay suffered by the task T_i due to its own self suspension and also due to the self-suspension of all higher priority tasks. And b_i is the worst case self-suspension time of T_i. Now, how do we compute bt_i, the worst case self delay the T_i you will incur on account of self-suspension of itself and all other higher priority tasks, because self-suspension of a lower priority task will not affect a task.

And the result is that the total delay suffered in the worst case is the sum of the self-suspension times of itself and all other higher priority tasks.

(Refer Slide Time: 25:28)



Revised RMA Condition

- Lehoczky's criterion needs to be modified to consider self suspension:

$$\boxed{e_i + bt_i} + \sum_{k=1}^{i-1} \left\lceil \frac{p_i}{p_k} \right\rceil * \overset{e_k+bt_k}{\boxed{e_k}} \leq p_i$$

- If a task undergoes multiple self suspensions:
- Then expression needs to be changed

So, if we consider self-suspension, we need to revise the rate monotonic utilization bounds, let us do it for the completion time theorem and the completion time theorem, the execution time of the current task is e_i + bt_i because this is the delay due to all high priority tasks and $\Sigma \lceil p_i / p_k \rceil * e_k$. But if a task undergoes multiple self-suspensions, then the expression needs to be changed. So, conservatively we need to consider that each task execution time increases by bt_i.

So, these tasks execution time increases bt_i and e_k is execution time increases to e_k + bt_k. So, just like we are adding the overhead of context switch to the execution time, here also we add the context, the self-suspension time of itself and all higher priority task into the execution time. That is the worst case of course. But then, in all these real time analysis, we consider the worst case situation.

So, we will, we are at the end of this lecture. We will stop here and we will look at few other issues and then we will look at resource sharing among the real time tasks, how we handle resource sharing without really creating scope for deadline misses. So, that is a very important topic, but before that, we need to discuss few more issues regarding the rate monotonic scheduler. We will stop here. Thank you.