Real Time Systems

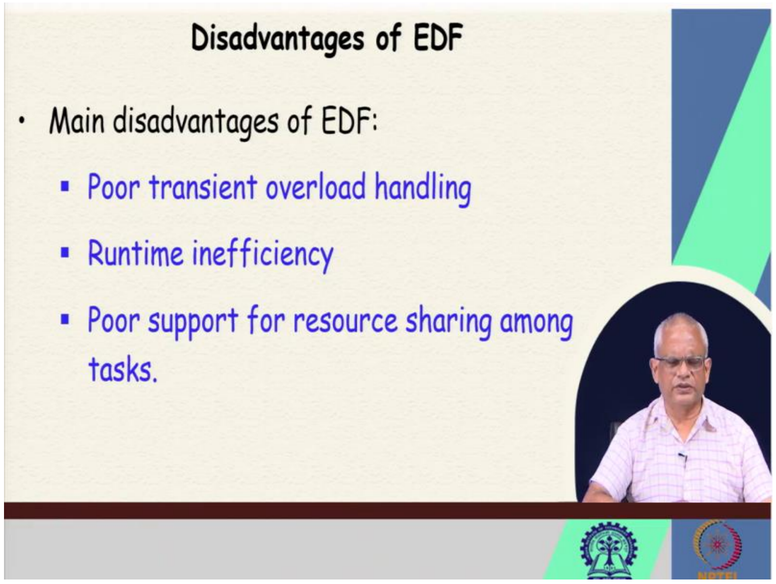
Professor Rajib Mall
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture 16

Variants of EDF and Rate Monotonic Scheduling

Welcome to this lecture. In the last lecture, we were discussing about EDF, the Earliest Deadline First Scheduling algorithm. It is an important algorithm because it is the optimal algorithm. If EDF cannot run a task set feasibly then there can be no other scheduler, which can run that task set. And the EDF algorithm is very simple, it is optimal, but then it is very rarely used and we are trying to examine the reasons behind why EDF is not being used, what are its disadvantages.

(Refer Slide Time: 2:10)



Disadvantages of EDF

- Main disadvantages of EDF:
  - Poor transient overload handling
  - Runtime inefficiency
  - Poor support for resource sharing among tasks.

And we identified three main reasons, three main disadvantages of EDF. The three main disadvantages are; one is that poor transient overload handling. What it really means is that whenever a task that is running is delayed due to some reason, maybe took a path in the program, which is very uncommon, it is a long path, and it is being delayed.

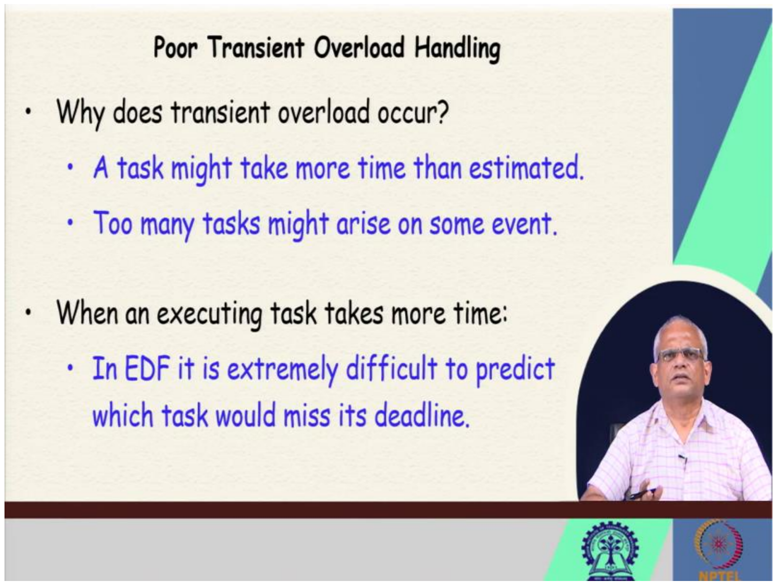
It got into an infinite loop somehow or maybe that it is waiting for some event or some resource it was delayed and then this algorithm EDF, it behaves very counter intuitively at this time. It just keeps on increasing the priority of that task to higher and higher levels; that means that other tasks can have to wait, this task is not preempted, because it has the shortest absolute deadline.

And in that situation, which tasks will miss their deadline is not we cannot say you cannot predict and also which critical task will meet deadline we cannot predict at the beginning. If some task is delayed, depending on the situation at that time, even the most critical task may miss its deadline, so this is a big disadvantage of EDF.

The second disadvantage is runtime inefficiency. The naive very simple implementation we discussed about keeping them in a queue is O (n) for n tasks, a complexity, time complexity is O (n). But even if we use a more sophisticated data structure still it is O(log n), but we will see that for the rate monotonic scheduler, the complexity is 1, which is much more efficient compared to EDF. The third and more vexing problem is the poor support for resource sharing among tasks.

As long as tasks, they share some resources, which is normal in any practical application the tasks share resources maybe partially produced results, the share, part of the result is completed by a tasks passed on to another task again, complete some part and again gives us back etc., that normally happens. But if that is the case, then tasks can miss their deadline using EDF. So, these are three major problems with EDF and this is possibly the reason why it is not popular.

(Refer Slide Time: 04:08)



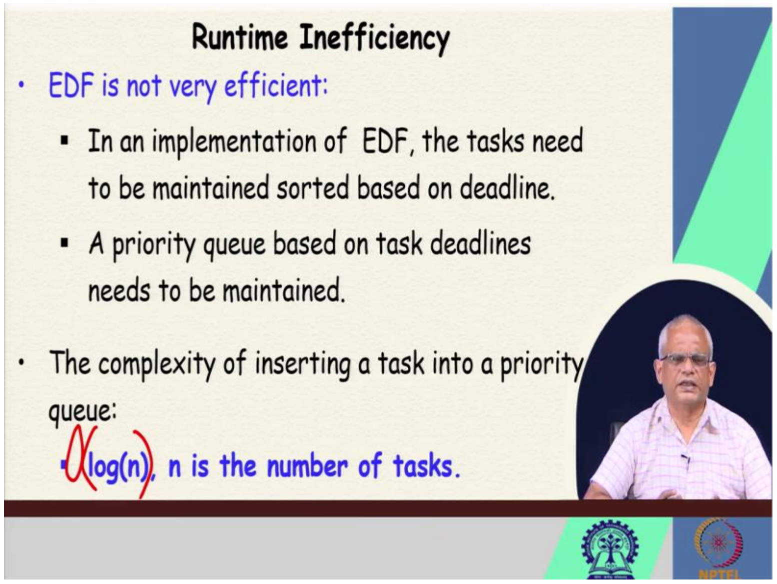
Poor Transient Overload Handling

- Why does transient overload occur?
  - A task might take more time than estimated.
  - Too many tasks might arise on some event.
- When an executing task takes more time:
  - In EDF it is extremely difficult to predict which task would miss its deadline.

The poor transient overload handling is that a transient overload occurs when a task takes more time than estimated, maybe because it got into an infinite loop or it took different path in the program and so on. Or the overload can occur when there are too many tasks that arise at some event. And when such transient overload occurs in EDF it becomes to predict beforehand that which task is going to miss its deadline, it is a big problem.

Because every system there are some critical tasks and the tasks which are not so critical, for example, a task which logs the result for an audit trail, the results are log, that is not so critical because even if the logging is delayed little bit, no problem, but then the one which let us say a chemical plant, it should shut off the chemical reaction in some situation if that gets delayed, then that is going to be dangerous. So, in such situations, EDF becomes difficult to use, because we do not know that whether some time the task most critical tasks will miss their deadlines.

(Refer Slide Time: 05:36)

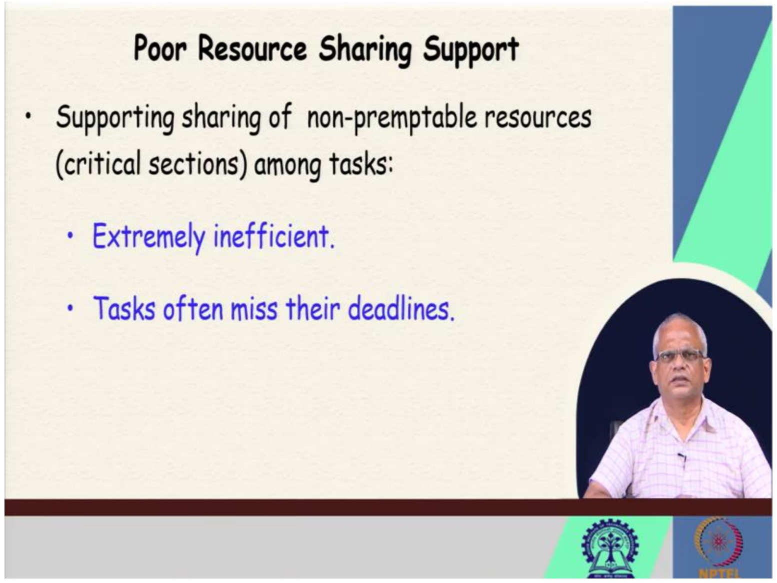


Runtime Inefficiency

- EDF is not very efficient:
  - In an implementation of EDF, the tasks need to be maintained sorted based on deadline.
  - A priority queue based on task deadlines needs to be maintained.
- The complexity of inserting a task into a priority queue:
  - $O(\log(n))$, n is the number of tasks.

Runtime inefficiency in a very simple data structure like queue text O(n), but even if you use a priority queue, it may take O(log n) and log n O(log n) is also O (log n) is also not a very efficient scheduler, because the rate monotonic can schedule in O(1) time.

(Refer Slide Time: 06:09)



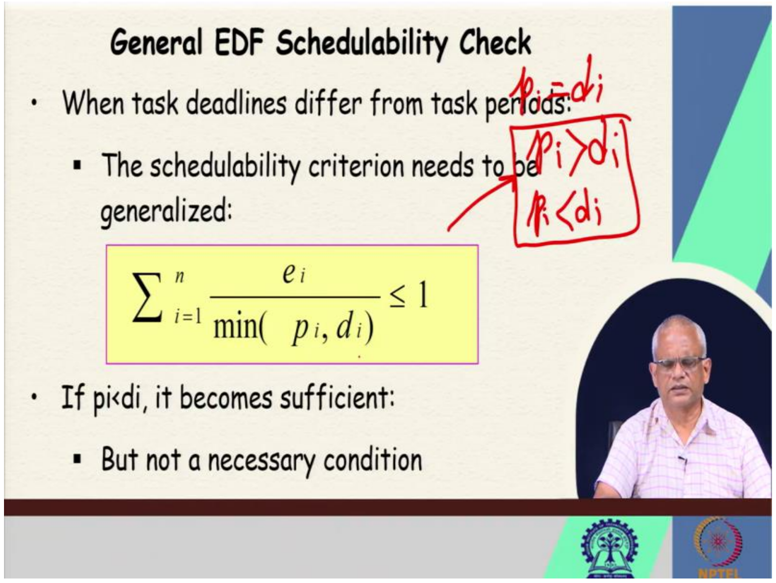
Poor Resource Sharing Support

- Supporting sharing of non-premptable resources (critical sections) among tasks:
  - Extremely inefficient.
  - Tasks often miss their deadlines.

And when there are sharing of non preemptible resources, critical sections among tasks, first is that if we devise a solution, which will let the task share without causing much problem, then the algorithm becomes extremely inefficient. And if we use a simpler algorithm for resource sharing the tasks can miss their deadlines due to something called as a priority inversion.

And this issue, we will discuss a little later in more detail that why EDF has so poor characteristic with respect to resource sharing support, whereas the rate monotonic algorithm, we have efficient ways for tasks to share resources without any causing deadline misses.

(Refer Slide Time: 07:04)



General EDF Schedulability Check

- When task deadlines differ from task periods: $p_i = d_i$
  - The schedulability criterion needs to be generalized: $\boxed{\begin{matrix} p_i > d_i \\ p_i < d_i \end{matrix}}$

$$\sum_{i=1}^{n} \frac{e_i}{\min(p_i, d_i)} \leq 1$$

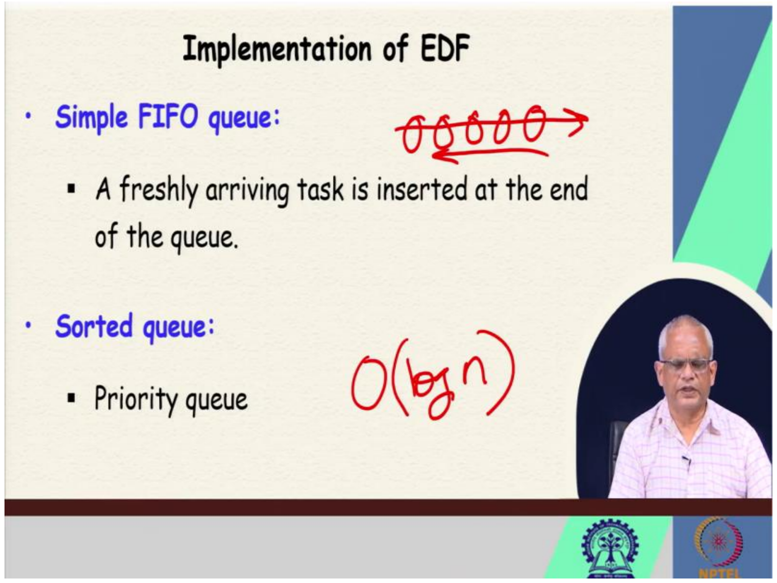
- If pi<di, it becomes sufficient:
  - But not a necessary condition

So far, we have been discussing that the task period and the deadline are the same, $p_i = d_i$. All the examples we had taken and we had implicitly assumed in our schedulability expression for EDF is that $p_i = d_i$. But it may so happen that the p_i, the period is greater than deadline, so deadline occurs before the period and in rare cases maybe $p_i < d_i$.

So, these two cases, the expression that we discussed does not hold, we need to generalize the schedulability check expression. Now, let us just give that expression, we need to change e_i / p_i into ei / min (p_i,d_i). So, if p_i is greater than d_i, then it will be e_i / d_i, and if $p_i < d_i$, it will be e_i / p_i, not really we are discussing about the first case actually, $p_i > d_i$ then it will be $e_i / d_i \leq 1$.

But in the second case, for the first case when $p_i > d_i$ then this expression e_i / min (p_i,d_i) which is basically becomes $e_i / d_i < 1$, this is the criterion for schedulability of a set of tasks, but for the second case when $p_i \leq d_i$, it becomes a sufficient criterion, but not necessary criteria. We will not go details into this, but you can read up in the books we will not discuss in the lecture here, the implications of this, just mentioning the general schedulability check.

(Refer Slide Time: 09:36)



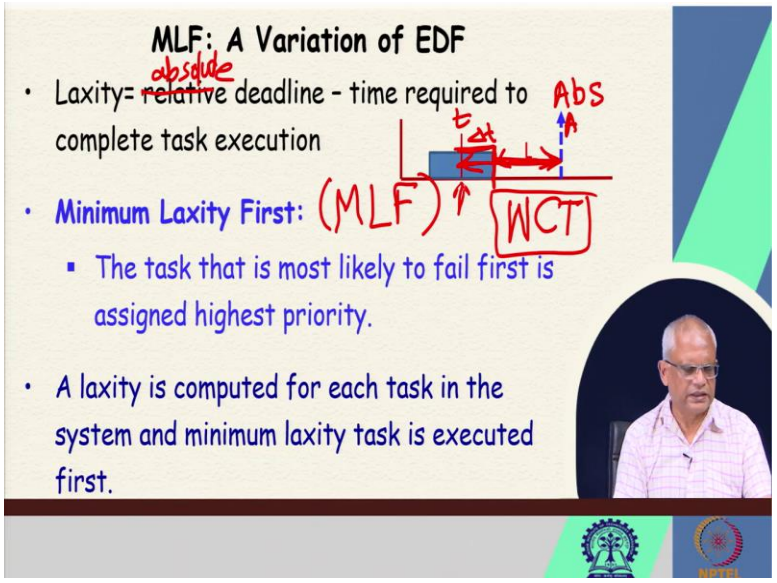
Implementation of EDF

- Simple FIFO queue:
  - A freshly arriving task is inserted at the end of the queue.
- Sorted queue:
  - Priority queue $O(\log n)$

Now let us look at the implementation of EDF. The simplest implementation is a FIFO queue, so we just have one queue, and here the tasks are just inserted at the end of the queue and the scheduler selects, it needs to traverse through the queue and select the one which is having the smallest absolute deadline. But we can also have a priority queue.

So, here we insert into the queue based on their absolute deadline, each time a new task comes arrives that is an instance of the task, we need to examine its proper place in the queue and for that the efficient data structure is a priority queue, but here, insertion of a task is O(log n). And that will be the complexity of implementing EDF using a priority queue.

(Refer Slide Time: 11:01)



MLF: A Variation of EDF

- Laxity= ~~relative~~ absolute deadline - time required to complete task execution
- **Minimum Laxity First:** (MLF)
  - The task that is most likely to fail first is assigned highest priority.
- A laxity is computed for each task in the system and minimum laxity task is executed first.

Abs
A
t
Δt
L
WCT

Now, before we conclude our discussion on the EDF we are not spending too much time on the EDF, because it is not a very popular scheduling algorithm and it is not very efficient. So, we are not spending too much time, we will just discuss a small variation of the EDF which is called as the maximum laxity first, sorry, the minimum laxity first MLF, the minimum laxity first algorithm. And here unlike the earliest deadline first where the absolute deadline is checked at any point of time, and the task having the smallest absolute deadline is picked up for scheduling.

Here the laxity of the task is computed and the task having the minimum laxity is taken up for scheduling. Now, how do you define laxity? Laxity is the relative deadline - the time required to complete the task execution. So, if this is the deadline and this is the execution time, and we are trying to compute at this time, the laxity will be L which is from the current time till the deadline of the task, the absolute deadline on the task.

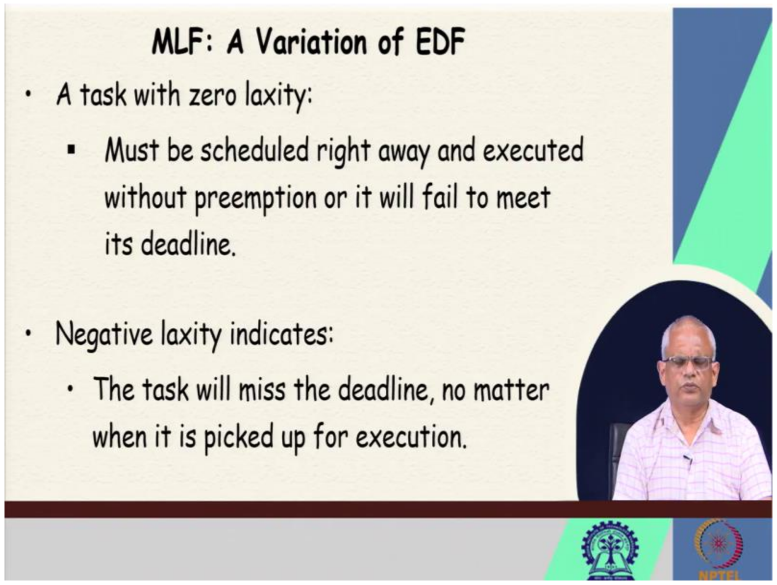
So, if this is the absolute deadline and this is the, so let me just change it to absolute deadline, I think that will be more accurate, the absolute deadline is this and the current time instant is t and the time required for the task to complete is let us say Δt, then if this is A, then A - Δt, this is the one is the laxity. We can think of laxity as how much I can delay this task, before it will fail. So, the one having the smallest laxity is going to fail first.

So, the complexity of computing the laxity is that we have to consider how much computation time is remaining for that, which is a bit difficult, because normally we give a very well worst case complexity and each time we need to consider how much computation remaining and then how much lax time is there and then subtract that from there, it is much more involved.

Earlier, we were just computing the scalability or the scheduling of a task based on its deadline, but here we also need to consider the computation, the worst-case computation time. And the worst case computation time or the WCT is measured with respect to the worst case path the longest path in the execution of the task.

So, for a task to compute the worst-case execution time, we need to check for various events, various types of situations where it takes different paths through the task, which is the worst-case path. And that gives us the worst case computation time and based on that the execution time is decided, and we need to use that here. So, it is a variation of the EDF, it is slightly more complicated, but for some situations may slightly give better results than EDF.

(Refer Slide Time: 15:48)

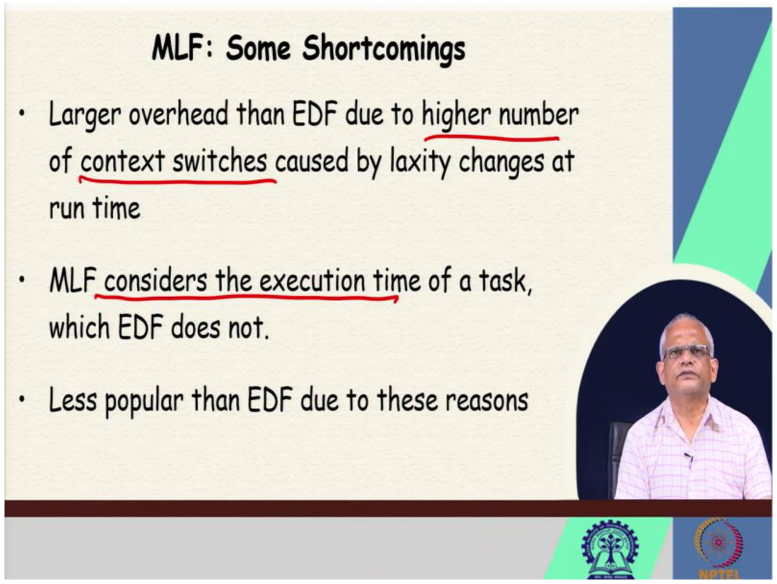


MLF: A Variation of EDF

- A task with zero laxity:
  - Must be scheduled right away and executed without preemption or it will fail to meet its deadline.
- Negative laxity indicates:
  - The task will miss the deadline, no matter when it is picked up for execution.

And here a task with zero laxity needs to be scheduled straight away, because unless it is scheduled right, then it is bound to miss its deadline. So, task with zero laxity has to be scheduled, otherwise there will be a failure in the system. But what about a task with negative laxity? Negative laxity means that this task will miss its deadline no matter when it is picked up for execution.

(Refer Slide Time: 16:27)



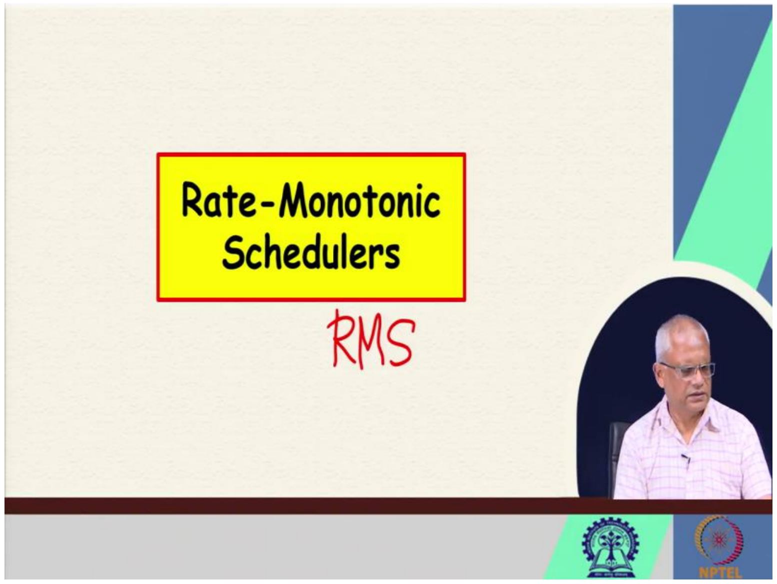
MLF: Some Shortcomings

- Larger overhead than EDF due to higher number of context switches caused by laxity changes at run time
- MLF considers the execution time of a task, which EDF does not.
- Less popular than EDF due to these reasons

If we think of it, the minimum laxity first there will be more number of context switches. The reason for this is that the laxity changes at runtime, as the task computes the laxity keeps on changing and here a task which was very high priority need not be so, when another new task arrives, we find that the laxity has changed.

So, intuitively we can argue that there will be higher number of context switches in MLF compared to EDF and also in EDF we need to consider the execution time and remember the execution time is the worst case execution time and therefore, it does not really give a true picture of the actual execution time. And that way the EDF it just considered the deadline and how will we compute the worst case execution time, the success of MLF depends on that, it is less popular than EDF due to these reasons.

(Refer Slide Time: 17:57)

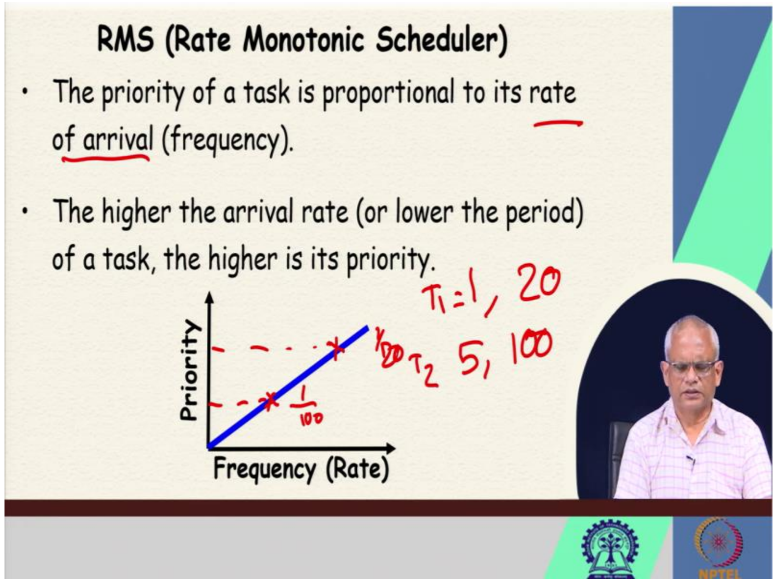


Rate-Monotonic Schedulers

RMS

Now, let us look at the static scheduler. So, far we have been looking at the EDF - Earliest Deadline First, which is a dynamic priority scheduler. Let us look at static priority scheduler, the Rate Monotonic Scheduler or the RMS and this is, I can say that it is the most popular real time task scheduler in embedded real time applications. Most of the operating systems they some way support real time, sorry, the rate monotonic schedulers to be used. Since it is so popular, let us look at it in more detail.

(Refer Slide Time: 18:57)



RMS (Rate Monotonic Scheduler)

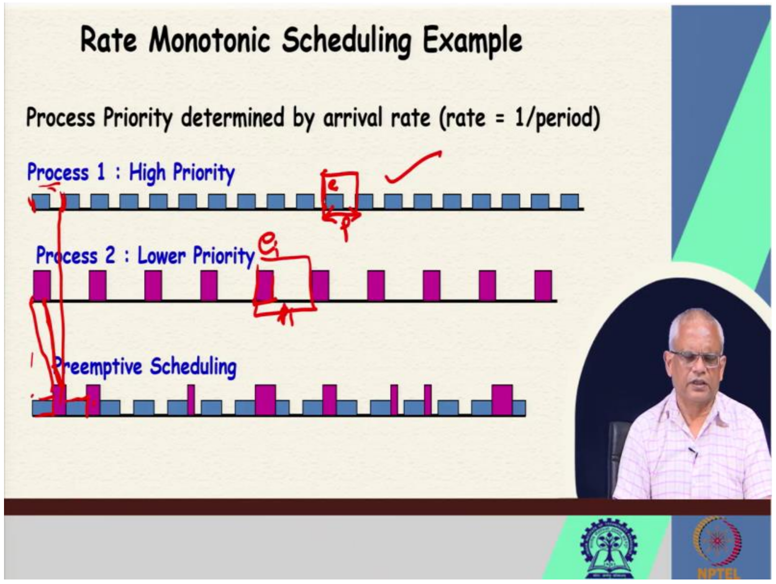
- The priority of a task is proportional to its rate of arrival (frequency).
- The higher the arrival rate (or lower the period) of a task, the higher is its priority.

Priority
Frequency (Rate)
$\frac{1}{100}$
$\frac{1}{20}$
$T_1 = 1, 20$
T_2 5, 100

The Rate Monotonic Scheduler really means that the priority of a task increases monotonically with its rate, the rate basically is the rate of arrival, the rate of a task is it’s the rate of arrival is that how short is its period, the higher is the rate of arrival, the shorter is its period, the higher is its rate of arrival, and the higher is the rate of arrival the higher is the priority of the task. The priority of a task increases monotonically with respect to its rate or the frequency.

So, if we have one task, which takes take’s 1 unit of execution time every 20 units and another task which takes 5 units of execution time in every 100 units, then the rate of this is 1/20, and the rate of this is 1/100. So, the rate of the first task T1 is much more than T2 and T1 will have a much higher priority, which is 1/20, compared to the second task, which is 1/100. So, that is the meaning of the rate monotonic scheduling.

(Refer Slide Time: 20:46)



Rate Monotonic Scheduling Example

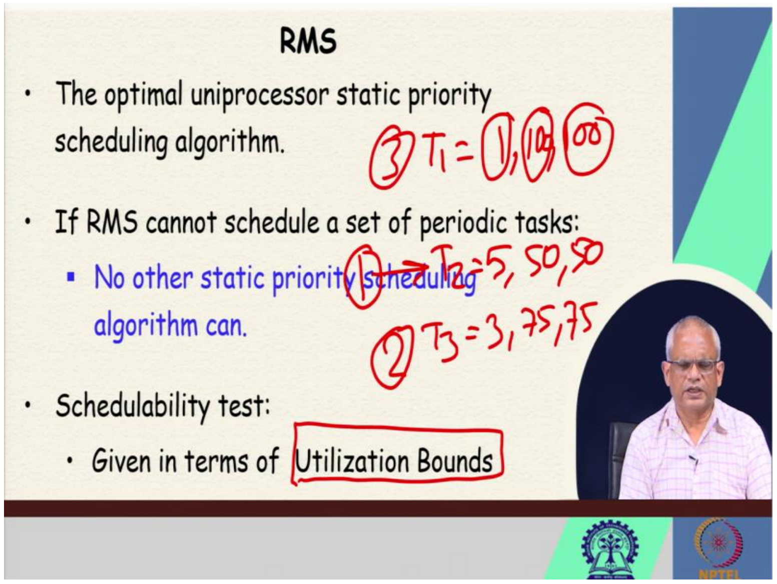
Process Priority determined by arrival rate (rate = 1/period)

Let us, understand the rate monotonic scheduling, let us say there are two tasks, written as process 1, process 2. See here that the process 1, the instances of the task appear very rapidly, the rate is very high and the process 2, the rate is low, so the for the first one, this is the rate, this is the period. I can show here that this is the period and this is the execution time and this is the period. For the second task, this is the execution time and this is the period.

So, this is e_i and this is p_i. And obviously, the first task has will be given higher priority according to the Rate Monotonic Schedulers, because it is occurring very frequently, the rate is very high compared to the second task, which is occurring less frequently. And if we look at how they will be scheduled by the scheduler, first the process 1 will be taken up for execution, as soon as process one completes, then only the process 2, see both of them are occurring together, but process 2 will take up only when process 1 completes.

And again as soon as the process 2 arrives process sorry process 1 arrives, process 2 will be preempted and the process 1 will be run and is it process 1 completes the remaining part of the process 2 will be taken up and so on. So, as long as there is a instance or process 1 that will run it has higher priority, it will preempt even the running currently running process 2.

(Refer Slide Time: 22:59)



RMS

- The optimal uniprocessor static priority scheduling algorithm.
- If RMS cannot schedule a set of periodic tasks:
  - No other static priority scheduling algorithm can.
- Schedulability test:
  - Given in terms of Utilization Bounds

3 $T_1 = 1, 100, 100$
1 → $T_2 = 5, 50, 50$
2 $T_3 = 3, 75, 75$

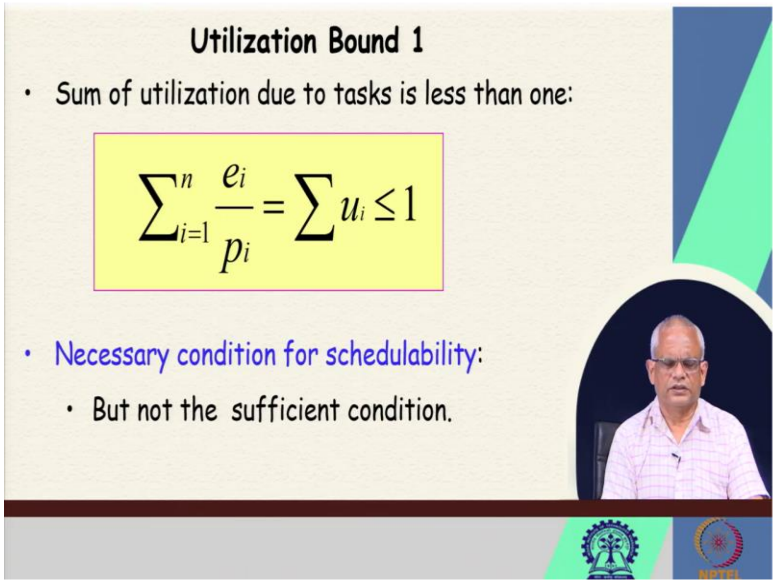
As we have been saying that this is the optimal uniprocessor static priority task scheduling algorithm, if a task set cannot be scheduled using the rate monotonic scheduler, then there cannot be any other static priority scheduler which can schedule that task and we meant by static priorities that once the priority of a task is computed it does not change.

So, if we have three tasks let us say T_1 is 100, 100 and T_2 is 5, 50, 50 and T_3 is 3, 75, 75, so the execution time of T_1 is 1 period is 100 and deadline is 100. The second task execution time is 5 period is 550 and deadline is 50, so in this case using the RMS we will assign the highest priority to T_2, let us say the highest priority is indicated by a number like 1.

Priority 1 has the highest in operating system, then the program will assign priority of 1 to T_2, priority of 2 to T_3 and priority of 3 to T_1. And if a task set that cannot be run using RMS then no other static priority algorithm can run that task set feasibly. Now, how do we know given a task set like this? How do we know whether that is feasibly schedulable in RMS or not?

For that, we will give some schedulability expressions in terms of the utilization bounds; the utilization bound is what is the maximum utilization up to which the task set can be considered schedulable under RMS. Now, let us look at those utilization bounds.

(Refer Slide Time: 25:39)



Utilization Bound 1

- Sum of utilization due to tasks is less than one:

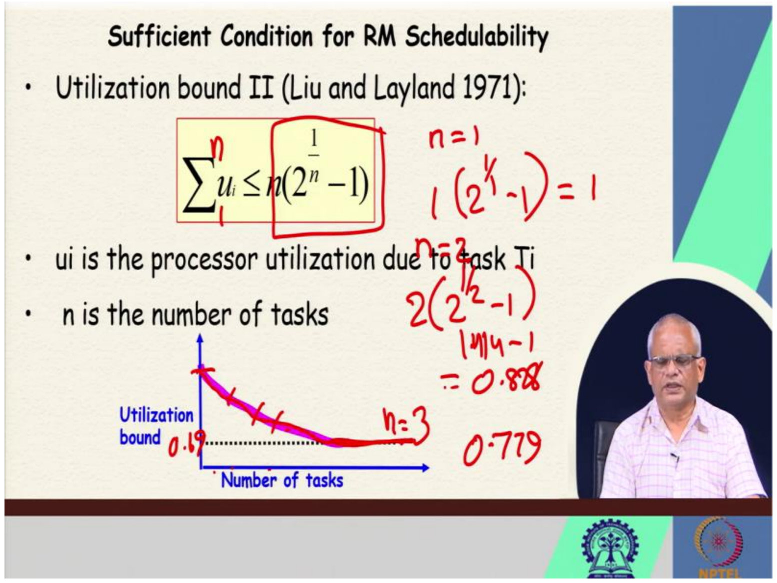
$$\sum_{i=1}^{n} \frac{e_i}{p_i} = \sum u_i \leq 1$$

- Necessary condition for schedulability:
  - But not the sufficient condition.

The first is utilization bound 1 which is the utilization due to all tasks should be less than 1, of course, any tasks requiring utilization more than 1 cannot be run and trivially this utilization bound holds for all scheduling algorithms, this is a trivial utilization bound but we need to check the utilization is more than 1 we can straight away say that it is not schedulable, with any of the scheduler, if the utilization bound utilization due to the tasks is more than 1 it cannot be scheduled.

So, this is a necessary condition for scheduling if the task set is meeting the utilization bound that is some of utilization is less than 1, then it is necessary for scheduling but if it is met, it does not mean or does not guarantee that it is schedulable under RMS, the rate monotonic scheduler, but unless this is met, we need not check further, so this is a necessary condition.

(Refer Slide Time: 27:07)



Sufficient Condition for RM Schedulability

- Utilization bound II (Liu and Layland 1971):

$$\boxed{\sum_{1}^{n} u_i \le n(2^{\frac{1}{n}} - 1)}$$

- ui is the processor utilization due to task Ti
- n is the number of tasks

$n=1$

$1(2^{1/1}-1)=1$

$n=2$

$2(2^{1/2}-1)$

$1.414-1$

$= 0.828$

0.779

Utilization bound
0.69
n=3
Number of tasks

Now, let us look at the sufficient condition, the sufficient condition was given by Liu and Layland way back in 1971, nearly 50 years back, so they gave this expression with rigorous proof that if the $\sum_{1..n} \le n(2^{1/n} - 1)$, then the task set is guaranteed to be schedulable. So, this is a sufficient condition.

If this is satisfied, we did not look any further, but on the other hand even if this is satisfied, it not satisfied, then the task set may be schedulable that we will look at a little later, but to consider the implication of this, let us say we have only a single task, so then n = 1 and this bound for number of tasks n = 1, it becomes $1(2^{1/1} -1) = 1$.

Or in other words, as long as there is a single task, even up to 100 % utilization, that task can be feasibly scheduled, but what about n = 2 then this expression becomes $2(2^{1/2} - 1) = 1.414 – 1 = 0.828$, so nearly 83 %.

So, up to 82.8 % utilization two task sets can be run. Now, what about 3, T = 3, n = 3, so here if we compute $3(2^{1/3} – 1) = 0.779$, we will get 0.779. So, the utilization bound decreases with the number of tasks, with a single task it is 100 %, 2 tasks it is 0.828, 3 tasks it is 0.779, then 0.773 and so on.

And but as the number of task increases it saturates at around 0.69, even if we have infinite tasks, the bound remains at 69, we can show that result and the main idea here is that a task set can be scheduled, the sufficient condition for scheduling is that $n(2^{1/n} - 1) \geq \sum_{1..n}$.

And this bound $n\ (2^{1/n} -1)$, if we substitute various values of n, the number of tasks 1, 2, 3, 4, etc., we will see that it falls, but then it stabilizes after some time. You can also check taking larger values of n, see how the graph behaves, the utilization bound behaves, and we will look further into the task schedulability.

Because this is a very important topic that given a set of tasks whether it will be rate monotonic schedulable, then only we will proceed with designing the system will assign priorities, the tasks and will take up implementation of the software and so on. First we need to check or analyse whether the task set is schedulable. So, we are almost at the end of this lecture, stop here, and we will continue from this point, in the next class we will take more example and more insight into the utilization bound that we will do in the next class. Thank you.