

Real Time Systems
Professor Rajib Mall
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture 19
Rate Monotonic Schedulability: Miscellaneous issues

Welcome to this lecture. In the last lecture, we are trying to investigate a very important design problem in real time systems that is given a set of tasks, if the task set feasibly schedulable on a uniprocessor using the rate monotonic scheduler, since the rate monotonic scheduler is prevalent, it is used across many systems and uniprocessors in small embedded systems, they are being used in large numbers.

It is a important problem to check whether a given task set is schedulable on a uniprocessor and we initially simplified the problem saying that the task set is independent, no resource requirement and so on. And possibly one of the very important results in checking the schedulability of a set of tasks is the Liu Layland result.

The Liu Layland result is given in terms of a utilization bound for a set of n tasks and the utilization bound decreases with increase in the number of tasks and is given as multiplication of two terms that is n which is the total number of tasks into $2^{1/n} - 1$. But then, we said in the last class that the Liu Layland result is a sufficient condition that is if Lui Layland condition is satisfied, the task set is guaranteed to be schedulable on a uniprocessor using the rate monotonic scheduler.

But the result is bit conservative in the sense that even if a task set fails the Liu Layland criterion there is a possibility that the task set may be actually schedulable. And we were trying to investigate how to check if the task set fails Liu Layland's criterion, how to determine whether it is actually schedulable. And we had taken a few example and we are trying to apply the Liu and Lehoczky's completion time theorem. Now, let us start from that point.

(Refer Slide Time: 3:18)

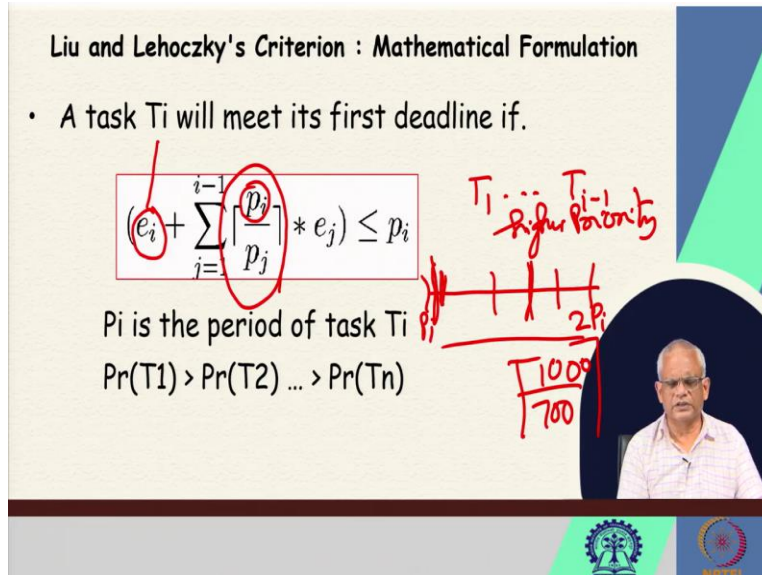
Liu and Lehoczky's Criterion : Mathematical Formulation

- A task T_i will meet its first deadline if.

$$e_i + \sum_{j=1}^{i-1} \left\lceil \frac{p_j}{p_i} \right\rceil * e_j \leq p_i$$

p_i is the period of task T_i
 $\Pr(T_1) > \Pr(T_2) \dots > \Pr(T_n)$

Handwritten notes:
 $T_1 \dots T_{i-1}$
Higher Priority
 $\frac{1000}{700}$



The Liu Lehoczky's criterion is called as the completion time theorem and the mathematical formulation is given by this, here e_i is the execution time of task T_i and p_i is the period of task p_i and T_1 to T_{i-1} are the higher priority tasks. And we know that a higher priority task when it is ready will preempt T_i and will run and therefore T_i will get delayed. So, the amount of delay suffered by a higher priority task is given by this expression.

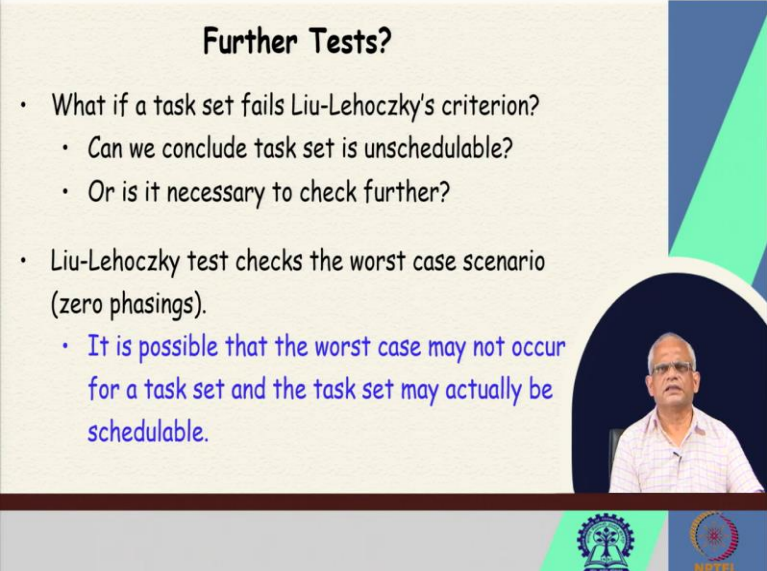
This is the number of time that task p_j will execute during p_i . If this is p_i and there is a i^{th} instance of p_i and this is the second instance of p_i then p_j if it occurs once here then it may occur again after p_j and so on. So the number of instances of p_j is given by $\lceil p_i / p_j \rceil$ and we are discussing why it is a ceiling. And we said that if this is 1000, and let us say, $p_j = 700$, then let us say it occurred at around 700 once, it occurred after the first one for, immediately after p_i , and then it may occur around 700 once.

So, two instances, even though its period is 700, but two instances and we get that by $\lceil 1000 / 700 \rceil$. So, we do that for all higher priority tasks and multiply that by the execution time because each time it records it executes for e_j . So, this is the mathematical formulation of the Liu Lehoczky's completion time criterion.

And if that is less than the period, that means the task will complete. And we assume that the period is equal to deadline. So that is the essential idea behind the mathematical formulation of

the Liu Lehoczky's criterion. And we had also seen how we can draw the schedule manually and check.

(Refer Slide Time: 6:44)



Further Tests?

- What if a task set fails Liu-Lehoczky's criterion?
 - Can we conclude task set is unschedulable?
 - Or is it necessary to check further?
- Liu-Lehoczky test checks the worst case scenario (zero phasings).
 - It is possible that the worst case may not occur for a task set and the task set may actually be schedulable.

The slide features a video inset of a man in a light-colored shirt speaking. The background is light beige with a blue and green geometric design on the right side. At the bottom, there are logos for IIT Bombay and IIT Madras.

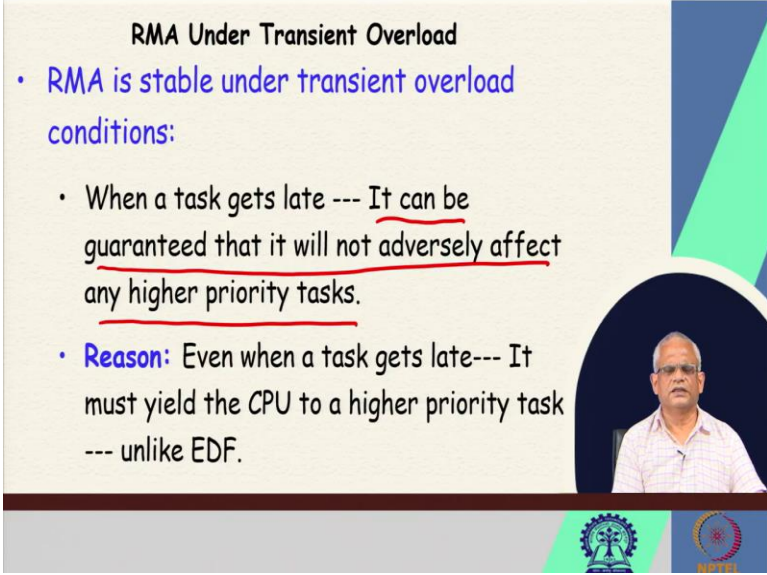
But suppose a task set also fails Liu Lehoczky's criterion, it had failed Liu Layland's criterion and then we are saying that whether at all it can fail, it can pass the Liu Lehoczky's completion time criterion but suppose it fails again then is the task set not schedulable at all? Can we say that or is there any further tests, okay, even if task set fails Liu Lehoczky's criterion, there is a very small chance that it can be actually schedulable, the reason is this that the Liu Lehoczky's criterion considers that all higher priority tasks occur in phase with this task that is zero phasing.

But then in actual situation, it may not be the case. So with zero phasing, it may fail, but then that may not occur for some specific task instances. So if a task set fails Liu Lehoczky's criterion still, there is a small chance, but that this task set is actually schedulable. But the chances are very small. And we do not have a closed formula like Liu Lehoczky's and Liu Laylands to check that case.

The only thing that possibly we can do is draw the schedule manually and check according to rate monotonic schedule, but I do not think that anybody while designing does design to that extent because the designs they leave some margin of accommodation, even if the tasks takes a little bit more time we should be able to tolerate that and not pack it with tasks so that even a small change will lead to a failure.

So if it tasks it fails Liu Lehoczky's criterion we need not look further, because it will make it a very aggressive design and live very little leeway. If some task a little bit gets delayed, then it will create a failure situation. So, if the task set fails Liu Lehoczky's criterion, it is better to leave that and look for a more powerful processor or maybe simplify the code so that it takes less execution time and it satisfies the Liu Lehoczky's criterion at least if not Liu Layland's criterion.

(Refer Slide Time: 10:03)



RMA Under Transient Overload

- RMA is stable under transient overload conditions:
 - When a task gets late --- It can be guaranteed that it will not adversely affect any higher priority tasks.
 - Reason: Even when a task gets late--- It must yield the CPU to a higher priority task --- unlike EDF.

The slide features a video inset of a man in a light-colored shirt speaking. At the bottom, there are logos for IIT Bombay and IIT Madras.

Now, let us investigate the behavior of the rate monotonic scheduler, under transient overload. One good thing about the rate monotonic scheduler is that it is stable under transient overload condition. The transient overloads occur due to many reasons; maybe a lot of tasks came up at some time, a lot of sporadic tasks and so on.

Or maybe all tasks came in phase at one instant or maybe one task due to some condition, it just kept on waiting for event and got delayed or maybe a task took a path which is not normally taken the code and it took a longer path in the code and got delayed. So, what happens in that case, you have a transient overload condition develops, how does the system behave under the rate monotonic schedule?

The good thing is that it is guaranteed even if a task gets late; it will not adversely affect any of its higher priority tasks. But how can we tell it so confidently? What is the reason behind this assertion that if a task gets late due to some reason, the higher priority tasks will not get adversely affected.

The reasoning behind this is that the way the rate monotonic scheduler works is that even if a task gets late, it has to yield the CPU when there is a higher priority task which is ready; always the higher priority task preempts the lower priority task. Even the lower priority task is missing its deadline does not matter.

The higher priority task will preempt the lower priority task. This is unlike the EDF, in the EDF if a task is getting delayed, its priority, virtual priority keeps on increasing. But this is not the case with rate monotonic scheduler. And here the higher priority task will never be affected by a lower priority task getting delayed. So that is a good thing. The rate monotonic scheduler is stable under transient overload conditions.

(Refer Slide Time: 12:57)

Implementation of RMA

- **A naïve RMA implementation:**
 - Maintain tasks in a FIFO Queue.
 - Insertion $O(1)$
 - Searching $O(n)$
- **A better implementation:**
 - Maintain tasks in a priority Q
 - Insertion $O(\log(n))$
 - Searching $O(1)$
- **Efficient implementation:**
 - Multilevel feedback Q

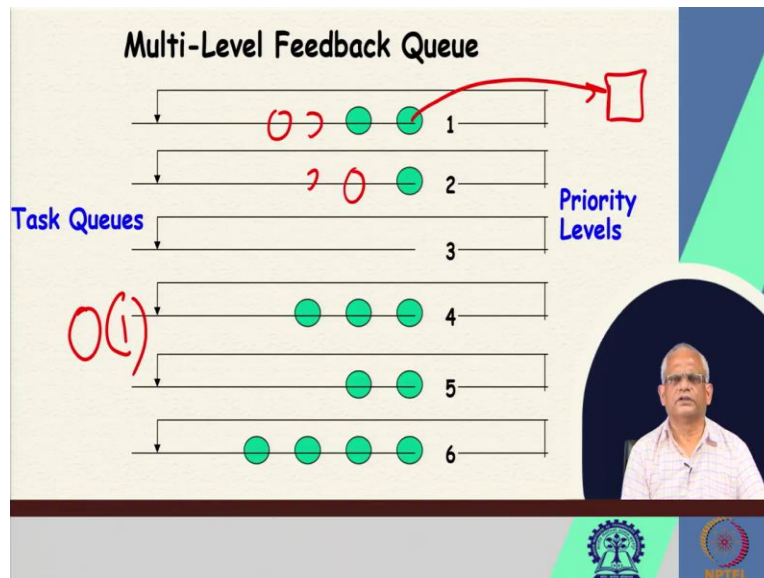
The slide includes a hand-drawn diagram in red showing a sequence of circles representing tasks in a queue, with an arrow pointing to a box labeled 'P' representing the processor. A video inset shows a man speaking, and logos for institutions are visible at the bottom.

Now, let us look at how the rate monotonic scheduler is implemented. In the very basic implementation of the rate monotonic scheduler, we maintain all tasks in a single FIFO queue. And when a task gets ready, we just add it to the FIFO queue. So, this is the FIFO queue and as the task becomes ready, even a higher priority task gets ready, it is added at the end of the queue and is the processor and the tasks are in the queue and the scheduler at every scheduling instant that is a task arrival and task completion.

It will look through the entire task set in the queue and find out which has the highest priority and it will dispatch that task. So, searching will take $O(n)$, not a very efficient implementation. A better implementation is to keep the task set in a priority queue. We know that a priority queue

is a data structure, a heap data structure where the insertion takes $O(\log n)$, but searching takes $O(1)$. So overall it is $O(\log n)$ definitely improvement over the very basic implementation. But can we do better? Yes, the way it is typically implemented is using a multi-level feedback queue.

(Refer Slide Time: 14:53)



Let us just look at that. If you look at the commercial operating system after some time we will discuss about the commercial operating systems. And there, we will see that they support the rate monotonic scheduling. By having a set of real time priority levels. We will see that all the real time system, all the real time commercial operating systems, they have two types of priority bands, one is called as the real time band, where the priority assigned by the programmer does not change at all.

And then there is another priority set a priority, which is the dynamic band where non real time tasks are assigned and their priority keeps on changing. We will investigate why and so on later. Now, the number of priority, real time priority levels supported by the commercial operating systems is something like 16 or 32. And each priority level, we have a multi-level feedback queue. So, if tasks are priority 1 arrive, they are queued here, tasks a priority 2 arrive they are queued here and so on.

Now, the scheduler at every scheduling point needs to just scan from here, highest priority, if there is a task at the highest priority, it just takes that and assigns to the CPU and that is over, the task has been selected. If there are none other tasks in the highest priority, then it looks at the

next and so on. And since this is a fixed number of queue, 16 or something and therefore, it is constant time $O(1)$.

So insertion in the multi-level feedback queue, the insertion as well as the searching both take $O(1)$ time, that is a very efficient good implementation and for small embedded system, it really helps and even otherwise also, because these are time constrained, the scheduler is required to be fast and therefore, the rate monotonic scheduling implementation is extremely efficient and therefore preferred.

(Refer Slide Time: 17:49)

RM Schedulability of Harmonically Related Tasks

- A set of periodic tasks is harmonically related, iff:
 - For every pair of tasks T_i and T_k : $T_i, T_k \in T$
 - If $P_i > P_k$, then $P_i = n * P_k$, where n is an integer.

(Example $p_1=10, p_2=20, p_3=60$) 300 Example

T1:	e1=20mSec,	p1=50mSec
T2:	e2=30mSec,	p2=150mSec
T3:	e3=90mSec,	p3=300mSec

- A set of harmonically related tasks is RMA schedulable:
 - If the sum total of the utilization due to the tasks is less than 1.

Now let us look at some issues with real time task scheduling using a rate monotonic scheduler. First, let us look at the schedulability of harmonically related tasks in a rate monotonic scheduler. Now, what is a harmonically related task? A harmonically related task is that for every task i and k belonging to the set of tasks T , if $T_i, T_k \in T$ and $P_i > P_k$, then $P_i = n * P_k$. So that means the tasks which are of higher period, they are multiples of the tasks having lower period.

So if we have let us say, this is one example, let us say you have 3 tasks and their periods are 10 20 60. For the task 20, the lower priority task is 10 and 10 divides 20., for 60 there are 2 lower priority tasks 10 20 and 60. So 10 and 20 both divide 60. We might have another higher priority task with period let us say 300. So 300 is divided squarely by all the 3 tasks.

So this is the harmonically related task set. So just to summarize a harmonically related tasks that is one where any task with period P_i is a integral multiple of all each task which has a lower period, the period of a task is integral multiple of the period of a task having a lower period. So, this is an example, the execution time is 20 30 and 90, but the periods are 50 150 and 300. So, 50 for 150, 50 divides and for 300 both 150 and 50 divided.

So, this is another harmonically related task and for harmonically related task, we do not have to look for Liu Layland schedulability or Leo Lehoczky's and so, on. Here, as long as the task set is harmonically related, the utilization due to tasks needs to be only less than 1 and then it is schedulable. So, the periods are multiples of each other.

And in that case, we can achieve 100 % utilization of the processor, the application that we are developing if we notice that the periods happen to be multiples of each other and there we can easily schedule them on a uniprocessor because 100 % utilization can be allowed and still the task set is feasibly scheduled.

(Refer Slide Time: 21:40)

Schedulability of a Harmonic Task Set

- By completion time theorem:

$$e_i + \sum_{k=1}^{i-1} \left\lceil \frac{p_i}{p_k} \right\rceil e_k \leq p_i$$
- For harmonically related tasks
Ceiling can be removed since the periods are integral multiples. So,

$$\frac{e_i}{p_i} + \sum_{k=1}^{i-1} \frac{e_k}{p_k} \leq 1$$
 or,

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

But can we mathematically show why for harmonically related set of tasks, the utilization bound is 100 % okay let us try here, by completion time theorem, we know that a task is schedulable if its execution time, and the execution time for all higher priority tasks during that period is less than the period of the task.

So, these are all higher priority task $k = 1$ to $i - 1$ and the number of occurrence of a higher priority task k is $\lceil p_i / p_k \rceil$ and the execution time is e_k , now since it is a harmonically related task, we know that p_i will be divided squarely by p_k or p_i is a multiple, integral multiple of p_k . And if it is an integral multiple, then the ceiling is not required.

It is only when there is a fraction we approximate it to the next higher level and that is the role of the ceiling. But since it is the integral multiple of p_k , the ceiling has no role and we can remove the ceiling. So you can write $e_i / p_i + \sum_{k=1..i-1} e_k / p_k \leq 1$ or we can bring $\sum_{i=1..n} e_i / p_i \leq 1$

So that is what the lowest priority task. And if it is shown for lowest priority task is less than 1 it will also hold for $n - 1$ that is last but lowest priority and so on. So, that mathematically proves that for a higher, for a harmonically related set of tasks. The utilization bound is 1, it is very important result.

(Refer Slide Time: 24:07)

RMS vs. EDF	
<ul style="list-style-type: none"> Implementation multi-level priority queue, $O(1)$ 	<ul style="list-style-type: none"> Heap, $O(\log n)$
<ul style="list-style-type: none"> Processor utilization 0.69 	<ul style="list-style-type: none"> Processor utilization --full utilization
<ul style="list-style-type: none"> Context switches --many 	<ul style="list-style-type: none"> Context switches few
<ul style="list-style-type: none"> Guarantee test nontrivial 	<ul style="list-style-type: none"> Simple

Now let us compare the rate monotonic scheduler with the earliest deadline first scheduler, the rate monotonic scheduler typically in a commercial operating system, it is implemented as a multi-level queue, multi-level, first in first out queue, I am sorry, it is not priority queue it is multi-level first in first out queue and the complexity is $O(1)$ and for EDF it is a priority queue or a heap and the complexity is $O(\log n)$ and therefore, the rate monotonic scheduler is more efficient than the EDF.

Now, as far as the schedulability is concerned, if the number of tasks is large enough, then the CPU level utilization is only 69 % whereas, in an EDF full utilization or 100 % utilization can be achieved. Another comparison is that in the rate monotonic scheduler, the context switches are many, much more than the EDF, in EDF the context switches are less than the rate monotonic scheduler and we are arguing the other day saying that in rate monotonic scheduler, as long as there is a higher priority task is ready it will preempt any running task.

But in EDF if they have the same deadline then it does not, the running task and the task that is waiting at the same deadline, absolute deadline, it does not really preempt and also the task as long as the deadline is less than the other tasks there is no preemption. So, the context switches are less in EDF. But then, we said that for a large number of cases they produce identical schedules.

So, the context switches are similar, but then for more aggressively utilization, more aggressive utilization of the processor, we had taken an example actually and if you look at that example, you will see that the context switch of the rate monotonic scheduler is more than EDF. So, we can say that in general the context switches in the EDF is less than the rate monotonic scheduler, but in a general case, they may be identical, but never is the rate monotonic scheduler produces less context switch, its either the same or more.

Now, the guarantee test to check whether a set of tasks is actually schedulable under rate monotonic scheduler on a uniprocessor we had a series of steps to apply. We had the we had the Liu Layland result and we had the Liu Lehoczky's completion time result and even after that, we do not know if that fails the Liu Lehoczky's completion time, we do not know whether the task set is still not schedulable or there is a chance that it is schedulable.

On the other hand in the EDF as long as the utilization is less than 100 percent it is guaranteed that EDF will feasibly schedule that task set and we can see that the rate monotonic scheduler has some big advantages namely the efficiency of the scheduler. The implementation is much simpler and efficiency but otherwise for all other cases. So, only this case the rate monotonic scheduler scores over EDF as far as the efficiency and simplicity is concerned but for all other considerations that we have shown here, the EDF is better.

So, for this case first case, the rate monotonic scheduler is better. But, there are some things which we have not shown here which are big disadvantages for EDF. But for that the rate monotonic scheduler does very well. And if you remember we had said that resource sharing and transient overload, behavior under transient overload the rate monotonic scheduler is stable whereas the EDF is not stable and that is a very important thing in real time systems that stability under overload conditions and the second is resource sharing, efficient resource sharing supported by rate monotonic scheduler, but not by EDF.

So, over that the rate monotonic scheduler really scores over EDF and therefore the rate monotonic scheduler is used across a vast majority of the real time and embedded applications, the applications of EDF are relatively few. So, we will, we are at the end of this lecture, we will stop here and from this point, we will continue in the next lecture. Thank you.