Real-Time Systems

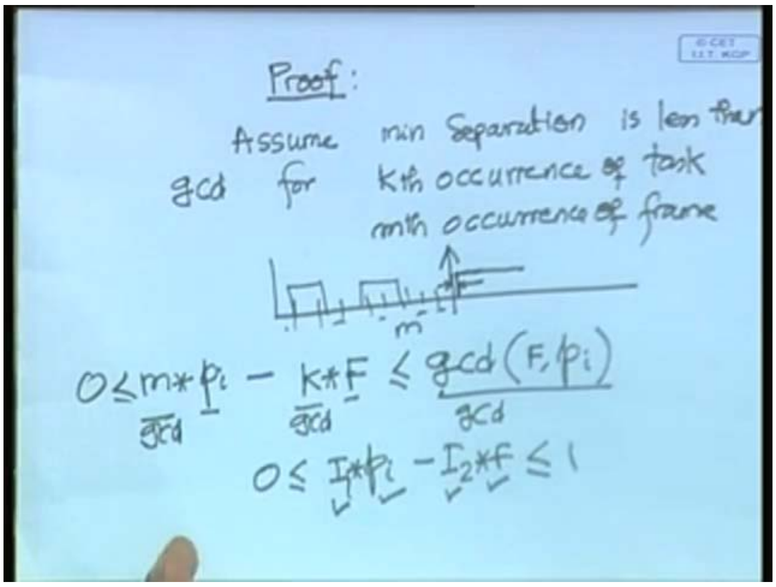
Prof. Dr. Rajib Mall
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture No. # 08
Event – Driven Scheduling

Let us proceed from where we left last time and we will basically look at event-driven schedulers. Cyclic schedulers, we had seen to some extent, and some parts we could not really finish in the last lecture. We will just quickly review those and then we will look at event-driven scheduling which is the main focus of this hour.

The cyclic schedulers, we had seen that they are driven by clocks and the frame size the scheduler works up it every frame boundary, and looks at what to do. And then we are trying to work out examples, trying to find out what can be a suitable frame size, which can on which the tasks can run satisfactorily. And we said that frame size is a important constraint, and one thing possibly it did not very convincingly tell you last time was about the minimum separation of a the frame boundary and the task arrival, which is equal to gcd f comma p i and I said that see one of your predecessors, he actually says gave the simplest proof. So let me just redo that proof now, because it was not really very convincing argument that I did not portray the argument that he had given.

(Refer Slide Time: 02:03)



Proof:

Assume min Separation is less than gcd for Kth occurrence of task mth occurrence of frame

$$0 \le \underset{gcd}{\underline{m*p_i}} - \underset{gcd}{\underline{k*F}} \le \underset{gcd}{\underline{gcd(F,p_i)}}$$

$$0 \le I_1*p_i - I_2*F \le 1$$

So just look at it, what he had said is that we trying to work out the proof which was worked given by one of your predecessors on a bonus problem. Like, I gave you today about the bonus problem to find out the frame size is not frame size. I think the schedule that needs to be stood when the frame size does not divide the major cycle. Similarly, on a similar assignment somebody had given this proof. So, said that see assume that for some occurrence of the task and frame occurrence, the separation between the frame size and the task arrival is less than gcd.

Assume, let me just write that much assume that the minimum separation is less than gcd is less than g c d. For some let us say a Kth occurrence of the task Kth occurrence of task and let us say mth occurrence of frame. So, we have a situation where m frames have occurred and the task has repeated K times and then we have a situation where the task is starting. The difference between the frame start and the tasks arrival is less than gcd.

So, in that case we will have K into F and m into pi minus K into F is less than equal to gcd. And this as to be greater than 0, because if it is then they are actually coinciding. I mean it is the earlier frame if it is less than 0 then it is in the earlier frame. So, gcd squarely divides pi and F that is the basic definition of gcd so you divide gcd on all sides.

So, from here we can get 0 is less than equal to some I 1 into pi minus I 2 into F is less than equal to 1. So, we have for all these are integers pi, I 1, I 2, F. So two integers which makes the value lie between 0 and 1 is not possible. So, it is a contradiction this cannot occurs. So, that was the proof we had given very nice proof. It is there in the book is proof that was given by him.

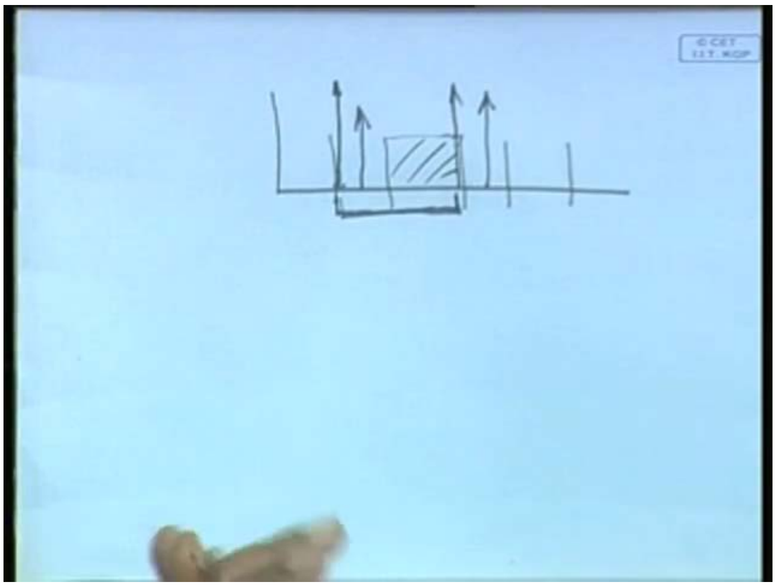
It is cannot be less than, but it can be greater than 1?

No, you are saying that gcd F comma pi gives the minimum separation between the frame boundary and the tasks arrival. It cannot be less than gcd.

Yes sir, it cannot be less than, but it can be greater not equal to gcd.

See, the worst case of a task scheduling occurs, when the task arrives too near the frame.

(Refer Slide Time: 06:24)



Let us look at the situation. Let us say these are the frames.

Sir, but I am talking about the proof sir. That proof is just saying that, the minimum separation cannot be less than gcd.

Yes.

It is not saying it is equal to gcd.

Does not matter. So, if you can or make it run for gcd, if you can schedule it.

That may be…

Then…

The proof just says that it may be, greater than or equal to gcd.

Yes, that is what we are saying gcd is the minimum.

Yeah, gcd is the minimum separation between two tasks. So, if we can take care of that is you know worst case scheduling condition occurs, when it is less than gcd, more than gcd is not a problem.

Just look at here. See if you have a task arriving here, then let us say the dead line is here, then we can run it on this frame. Now if the task arrives here, very close here then possibly that we do not even have one frame.

So, the worst case for any task scheduling occurs, when it is too near to a frame boundary. So then the available frames are much less, I mean there may not be even one frame, if it occurs too near a frame boundary.

So, for large gcd any occurrence separation larger than gcd, even not do not have a problem.

Sir,

If it is lower than gcd, then we might have a problem.

Yes,

Sir, in the proof.

Yes, yeah.

Sir, we are assuming that it is the K th occurrence of a task.

Yes,

And m th occurrence of a frame.

Yes,

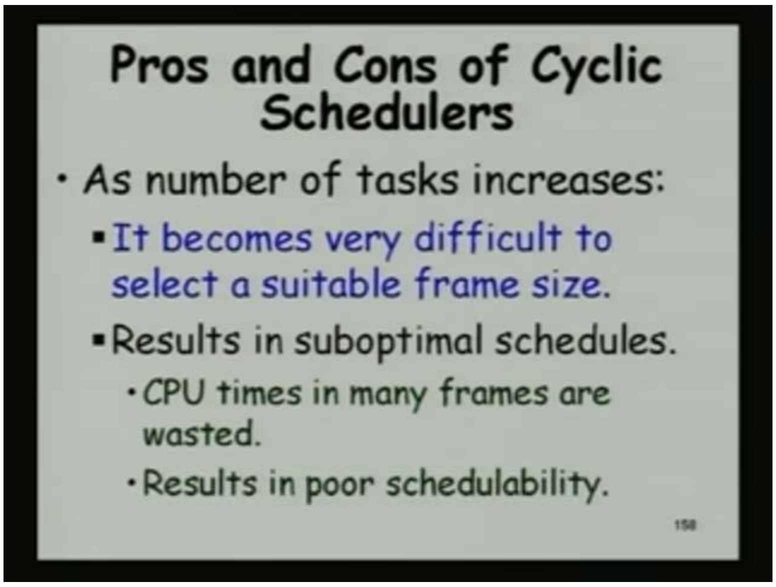
Sir, in that illustration that we have K into pi.

Yes,

That is a I mean, I just trying to give you the line of argument rather than the one. Actually what he says is that since, you have written K th occurrence of a task we should have K here and we should have m here, it does not really matter. Because K will also K into pi will divide gcd and m into F will also divide gcd.

So, these are all the proof. Many came with different proofs actually and these are all the proof which was the simplest and that we have documented in the book with acknowledge to him. So, let us proceed what we are seeing.

(Refer Slide Time: 09:00)



Pros and Cons of Cyclic Schedulers

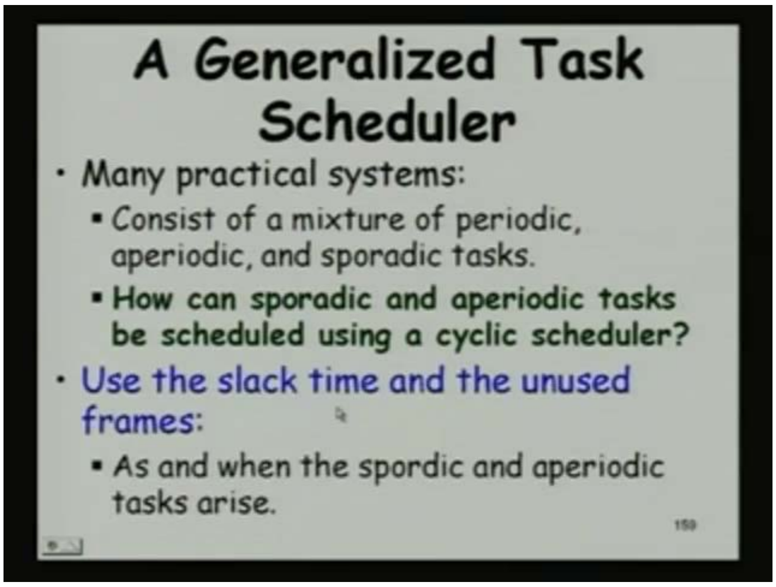
- As number of tasks increases:
  - It becomes very difficult to select a suitable frame size.
  - Results in suboptimal schedules.
    - CPU times in many frames are wasted.
    - Results in poor schedulability.

158

Now, the cyclic schedulers have one disadvantage. The disadvantage is that since we have to tryout various frames sizes and try splitting the tasks and zone. Many times we will find that we are not able to select a suitable frame size or may be, we are selecting a frame size, we does not lead to a good schedule. CPU times may be wasted in many frames.

And therefore, for many situations it we may not even find the schedule. Even though, if it is possible to run them using a cyclic scheduler we cannot run them. I mean at least we are not able to find a proper schedule and breaking tasks into many parts will make it still higher overhead context switches and so on.

(Refer Slide Time: 10:02)



A Generalized Task Scheduler

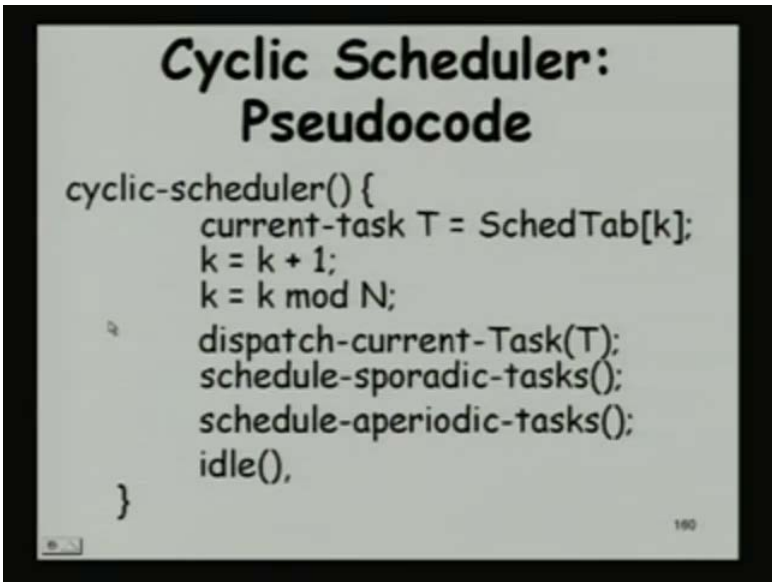
- Many practical systems:
  - Consist of a mixture of periodic, aperiodic, and sporadic tasks.
  - **How can sporadic and aperiodic tasks be scheduled using a cyclic scheduler?**
- Use the slack time and the unused frames:
  - As and when the spordic and aperiodic tasks arise.

159

Now, before you look at the event driven scheduler, just a few minute as you are saying earlier that what we look that till now and the pure cyclic scheduler we just handle periodic tasks, but there will be situations where it also might be require to handle at least few aperiodic and sporadic tasks. So, how can this are handled.

We need to use the unused frames and also the slack time that is left out, after a task completes. So when a sporadic or aperiodic task arises, the scheduler somehow needs to use the slack times and the unused frames.

(Refer Slide Time: 10:47)



Cyclic Scheduler: Pseudocode

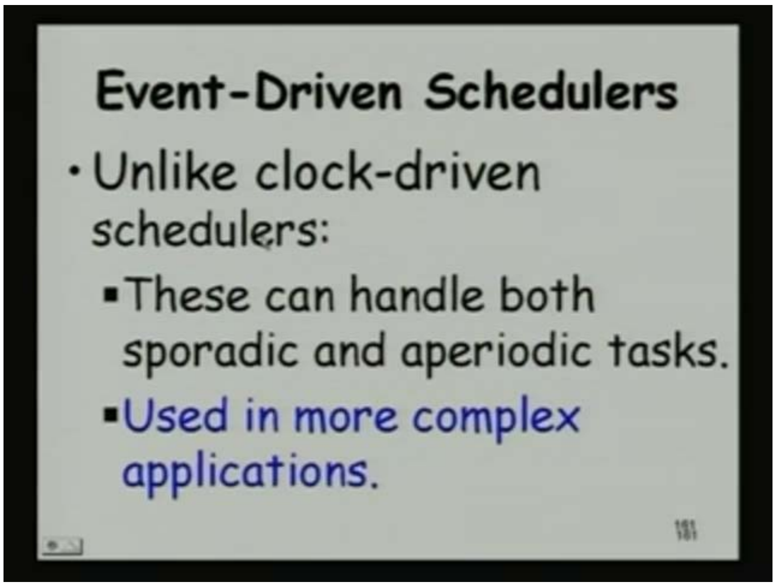
```
cyclic-scheduler() {
        current-task T = SchedTab[k];
        k = k + 1;
        k = k mod N;
        dispatch-current-Task(T);
        schedule-sporadic-tasks();
        schedule-aperiodic-tasks();
        idle(),
  }
```

160

This is the scheduler algorithm modified say earlier, which it was just looking at schedule table and then each frame size just trying to run the next task. Now it is doing the same thing actually trying to find out the next task to run, say K is equal to K plus 1 setting it to find the next task. And if it finds the end of the table just starts from the first task.

And then it runs the current task and if it completes even before the interrupt comes, it runs the sporadic tasks and even if there is still more time it runs the aperiodic tasks, otherwise it just idles. So, if there is no task to run in some frame it finds that nothing is to be run in some frame K, then it will run the current task and so on and similar is the case when the task completes early, until the clock interrupt comes. The aperiodic time or interrupt comes it will keep on doing this.

(Refer Slide Time: 12:09)



Event-Driven Schedulers

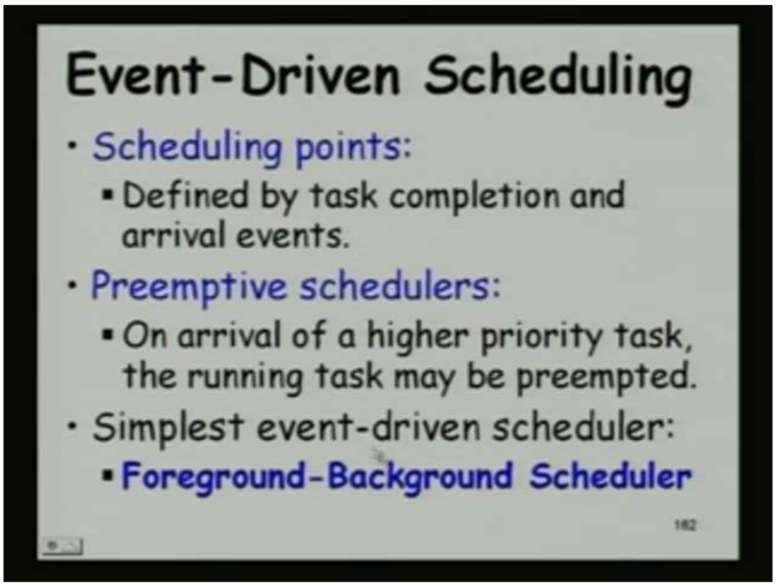
- Unlike clock-driven schedulers:
  - These can handle both sporadic and aperiodic tasks.
  - Used in more complex applications.

181

Now, let us look at the event driven schedulers here unlike the clock driven schedulers, these can handle both sporadic and aperiodic tasks very easily and definitely used in more complex applications. It is a necessary to be used the cyclic schedulers have their limitations only very simple systems they can handle.

For example, we are just discussing about a cell phone many tasks many are a aperiodic you are sending a message, receiving, making a call etcetera these are a sporadic tasks. And a cyclic scheduler cannot handle this.

(Refer Slide Time: 12:53)



Event-Driven Scheduling

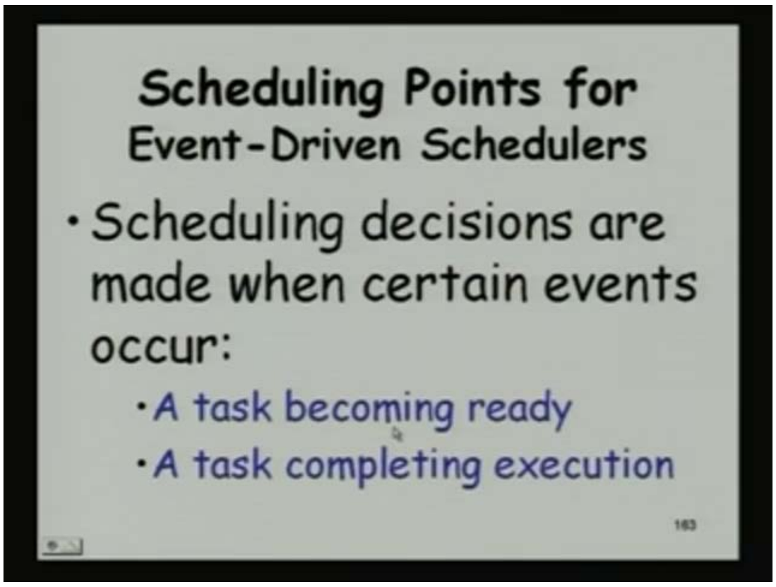
- Scheduling points:
  - Defined by task completion and arrival events.
- Preemptive schedulers:
  - On arrival of a higher priority task, the running task may be preempted.
- Simplest event-driven scheduler:
  - **Foreground-Background Scheduler**

182

In an event driven scheduler, the scheduling points are defined by task completion and arrival events. So, whenever either a task completes the scheduler will wake up and it will look at what to run next is the case when a new task arrives, it will be woken up and it will try to see. If the new task is a very high priority, then it can run and preempt the running task.

So, these are the event driven schedulers are preemptive schedulers. Because, if a higher priority task comes the current task may be preempted. So, just see here we are not talking of clock at all, but implicitly a task arrival might be on a clock cycle. The simplest event driven scheduling that we can think of is a foreground-background scheduler.

(Refer Slide Time: 14:00)



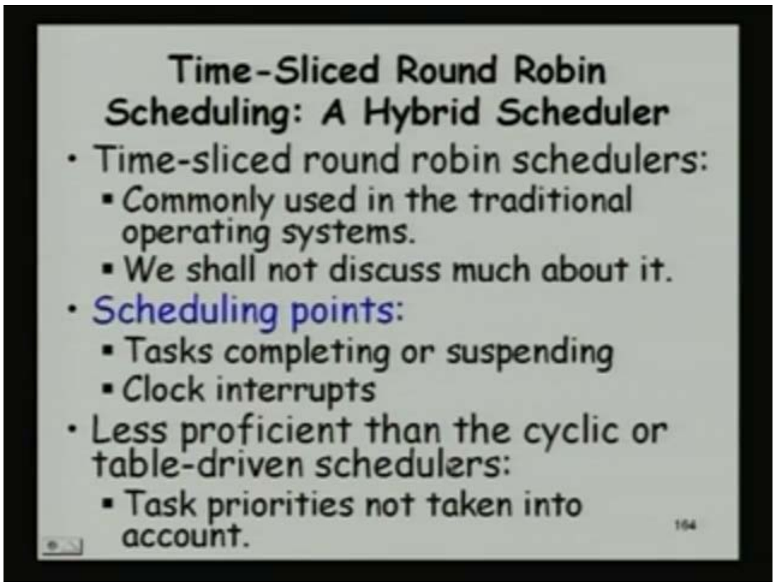
Scheduling Points for Event-Driven Schedulers

- Scheduling decisions are made when certain events occur:
  - A task becoming ready
  - A task completing execution

163

We will just discuss about the foreground-background scheduler shortly just a minute. Because that is the simplest and not really used in applications just for a concept.

(Refer Slide Time: 14:26)



Time-Sliced Round Robin Scheduling: A Hybrid Scheduler

- Time-sliced round robin schedulers:
  - Commonly used in the traditional operating systems.
  - We shall not discuss much about it.
- Scheduling points:
  - Tasks completing or suspending
  - Clock interrupts
- Less proficient than the cyclic or table-driven schedulers:
  - Task priorities not taken into account.

164

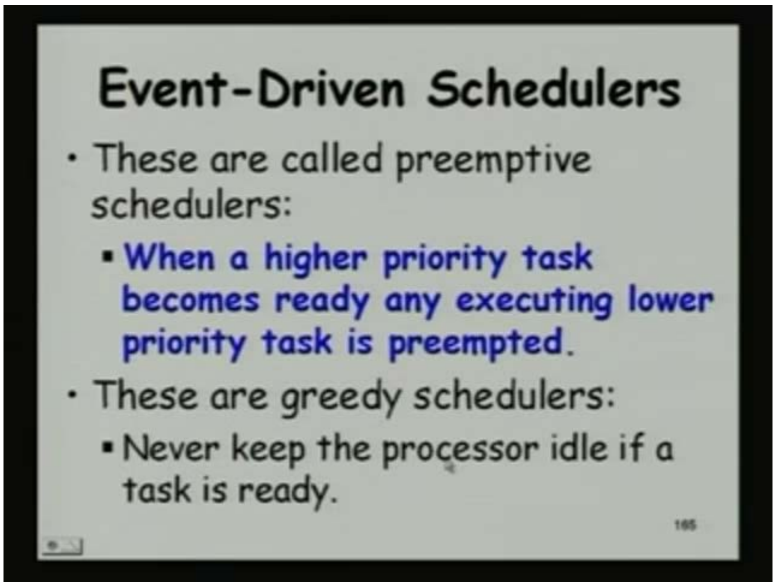
So, the scheduling points or a task becoming ready and a task completing its execution. Before the foreground background, let us look at something which we had already discussed in a first level operating system course is a time sliced round robin scheduler.

This is also an event driven scheduler, but it also has a concept of a time cycle I mean a time slice. So, this scheduler is actually discussed in the operating system books we will not spent time on this. Excepting just making a passing remark saying that, there are two types of events that are considered one is a task completing or suspending itself and also the clock interrupts.

Because after a time slice is over it needs to suspend the task and start the next task. We will see that these cannot or not suited to real time task scheduling not used. We will have quite a bit of discussion on UNIX later on and the windows.

We will see that the operating system they cannot really run real time tasks. We will identify the reasons.

(Refer Slide Time: 15:54)



Event-Driven Schedulers

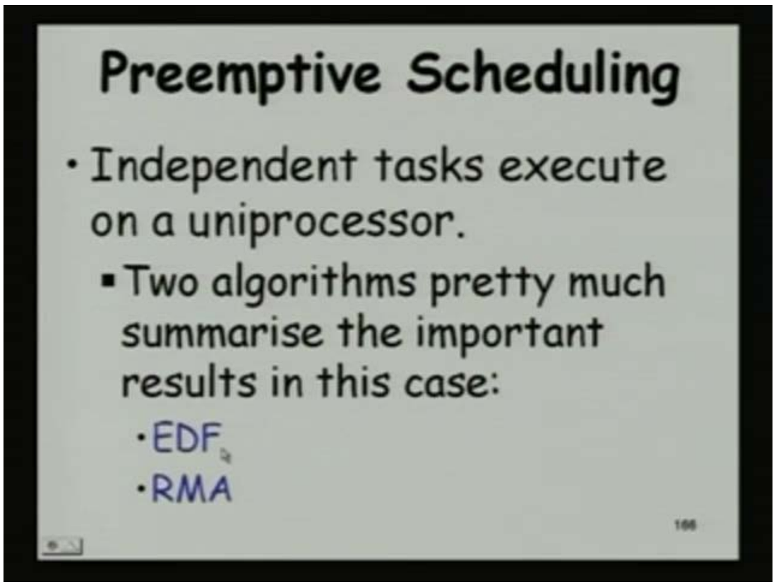
- These are called preemptive schedulers:
  - **When a higher priority task becomes ready any executing lower priority task is preempted.**
- These are greedy schedulers:
  - Never keep the processor idle if a task is ready.

165

The event driven schedulers they can they will preempt the currently running tasks, if a higher priority task arrives and these are also called as greedy schedulers. Because, as even as there is a task to run it will run them.

Unlike, let us say cyclic scheduler we had some part of a frame idle even though there is a task it is could have run. So, here as soon as a task is there it will be run. So, that is why these are also called as greedy scheduler in the literature.

(Refer Slide Time: 16:33)



Preemptive Scheduling

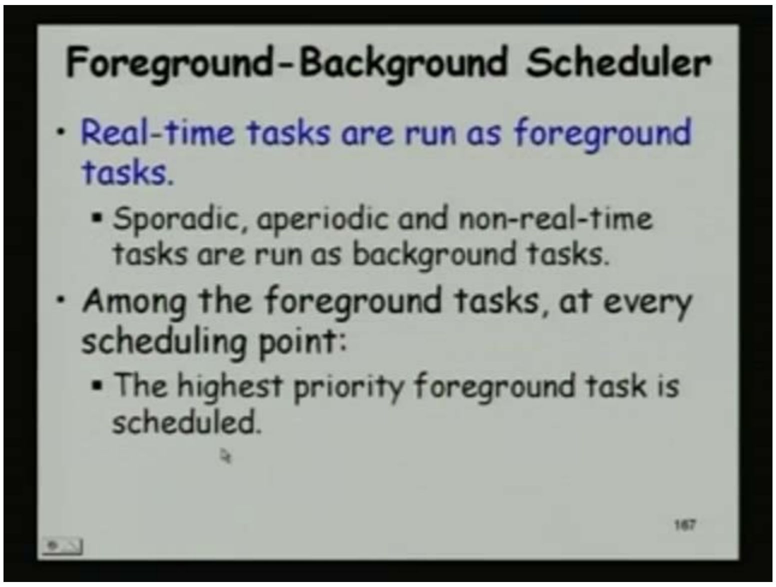
- Independent tasks execute on a uniprocessor.
  - Two algorithms pretty much summarise the important results in this case:
    - EDF
    - RMA

166

As, we are mentioning earlier lot of research was done on event driven scheduling on unit processor in the 1970s, thousands of paper published, but if you know two of these algorithms.

The EDF and RMA then you know all other algorithms, because these are the optimal ones. Let us, look at this we will see why these are optimal.

(Refer Slide Time: 17:07)



Foreground-Background Scheduler

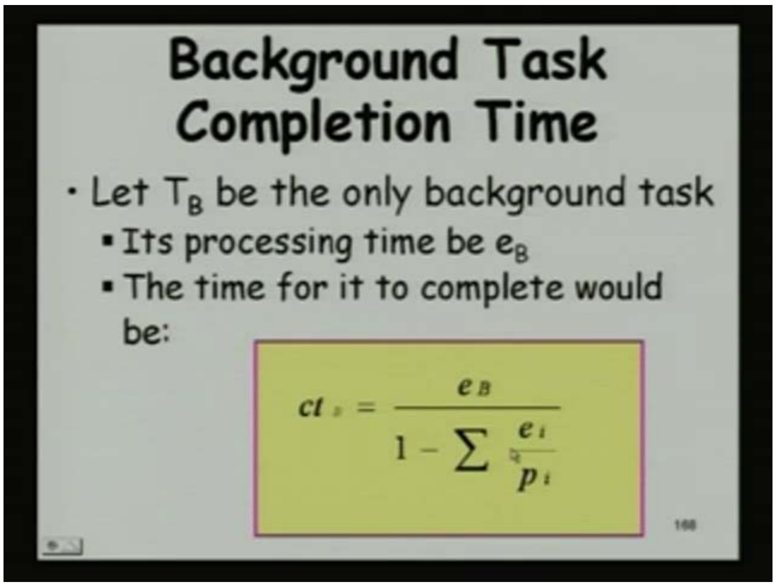
- Real-time tasks are run as foreground tasks.
  - Sporadic, aperiodic and non-real-time tasks are run as background tasks.
- Among the foreground tasks, at every scheduling point:
  - The highest priority foreground task is scheduled.

167

Before that, let us look at the foreground-background scheduler which is the simplest event driven scheduler. Here, the real time tasks are run as foreground tasks and non real time tasks are sporadic and aperiodic task, these are run as background tasks.

Among the foreground tasks say our real time tasks at every scheduling point. Scheduling point is basically either arrival of a real time task or completion of a real time task those are the scheduling points. The highest priority foreground task is scheduled and a background task runs, when no foreground task is ready.

(Refer Slide Time: 17:57)



Background Task Completion Time

- Let T_B be the only background task
  - Its processing time be e_B
  - The time for it to complete would be:

$$ct_B = \frac{e_B}{1 - \sum \frac{e_i}{p_i}}$$

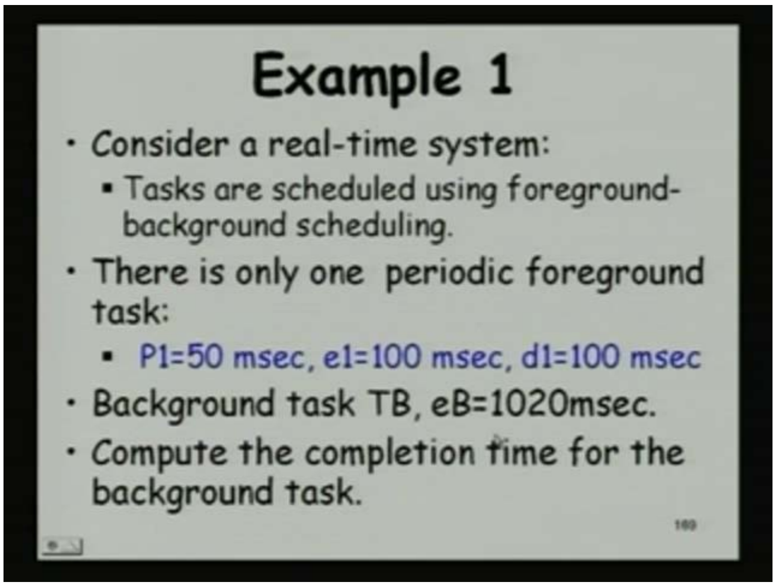
168

So, if T B is a background task and its processing task is e B then the time for the background task to complete, we can find out by finding the amused processor time by the foreground tasks.

Let us, say the foreground tasks are the periodic tasks e i divided by P i is the utilization they have on every P i. e i divided by p i is the utilization of the processor, for the task T i. So far every task, if you consider sigma e i divided by P i is the total utilization is not it.

So, every 1 millisecond or 1 unit of time sigma e i divided by P i is utilized by the higher the foreground tasks and one minus sigma e i divided by P i is the time that is available for the background task to run. And therefore, the background task we will take ct B time which is e B by 1 minus sigma e i divided by P i does that appear simple result. Is it ok? Anybody has difficult do this?

(Refer Slide Time: 19:28)



Example 1

- Consider a real-time system:
  - Tasks are scheduled using foreground-background scheduling.
- There is only one periodic foreground task:
  - P1=50 msec, e1=100 msec, d1=100 msec
- Background task TB, eB=1020msec.
- Compute the completion time for the background task.

160

Just a example so if there is one foreground task, periodic 1, 50 millisecond, this would have an e 1. e 1 is 50 millisecond, P 1 is 100 millisecond, d 1 is 100 millisecond. So, the foreground tasks consumes 50 millisecond every 100 millisecond and there is a background task which takes 1020 millisecond.

So, how much time will the background task need to complete. Can somebody do it? See the problem here, is this has to be corrected, e 1 is 150 and P 1 is 100 and the background task takes 1020 millisecond.

So, once the background task arrives, how much time will it require to complete?

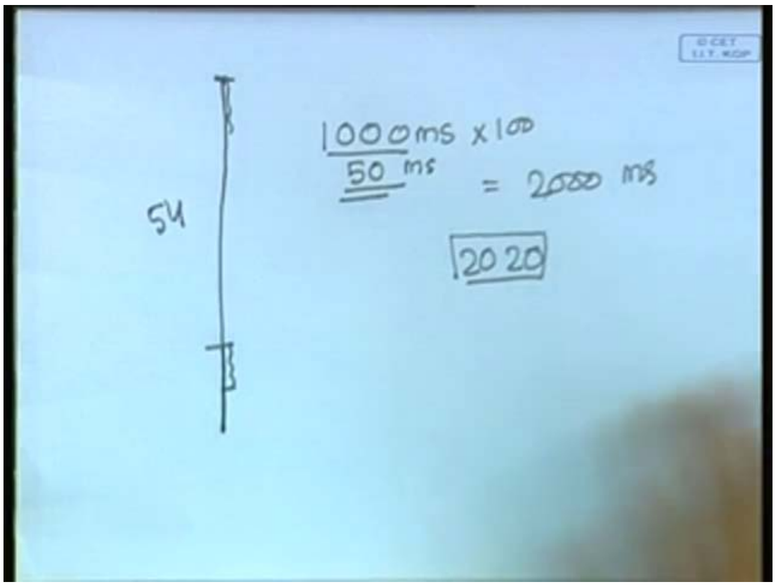
2040.

2040, How did you get that?

20 divided by e 1 minus 50 divided by 100.

Let us see the answer. We do not have the answer there worked out.

(Refer Slide Time: 20:59)



Let us see here, every 100 millisecond, it gets 50 millisecond. So, it will get 1000 millisecond into 100 in 2000 milliseconds. And then it will need another 20 to run, another 20 to complete, say 2020.

(( ))

Why 40? See once it is gets a frame to run it will run. I mean once it starts running it will take 20. Why 2040?

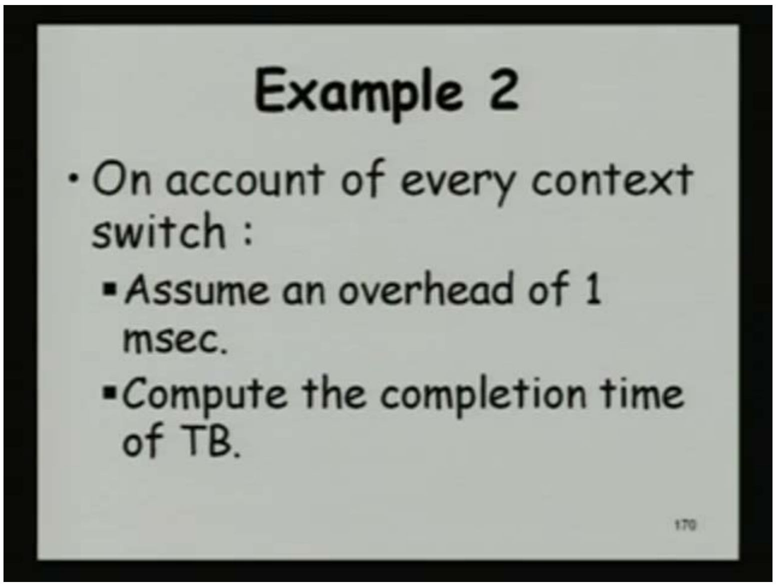
Sir, we can substitute the values for e i and P i and in that.

No, do not use the formula blindly just assume it runs it the task runs up to sometime. It keeps on getting time and next time, it just gets time it will complete it.

So, that is the same question as know in high school they use to ask the monkey climbing, this thing a pool which is do you remember that the problem is something like this. A monkey climbs a poll and each time the pool is no oily like class to 3 or 4. So, each time it jumps some 5 meters and it falls down 3 meters and the pole height is let us say 54 meters.

So, the final time it jumps it does not fall down something similar to that. So, here also at the last time, it runs see each time it just keeps on getting 50, but the last time it gets it will run it. So, 20 only it will take is not it does that appear.

(Refer Slide Time: 23:13)



Example 2

- On account of every context switch :
  - Assume an overhead of 1 msec.
  - Compute the completion time of TB.

170

So, let us just make that problem slightly more complicated. Assume that there is a context switch time of 1 millisecond required. Whenever, there is a switch from one task to another so what will be the completion time of T B? Now, simple extension of the previous answer. Yes, anybody would like to answer this question?

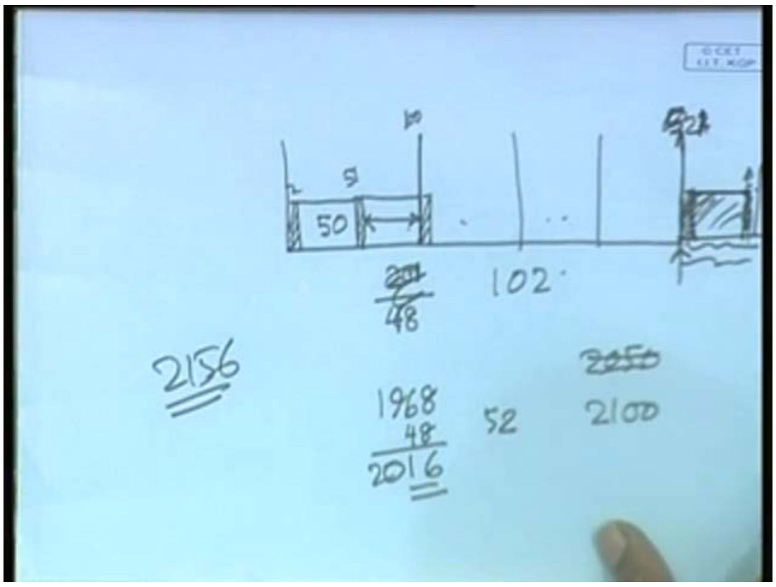
When we need to consider the context switch overhead of 1 millisecond every time a task new task starts, we had said no earlier recaptured the first level operating system course, when a task starts a context switch time is involved.

2060.

2060. Anybody has any different answer?

Anyone wants to add anything?

(Refer Slide Time: 24:36)



2041. See it is not very complicated see here that each time a task starts, we need a context switch time and it runs for 50. So, this is the 51. And then the background task starts and runs up to 100. Then the new task comes here.

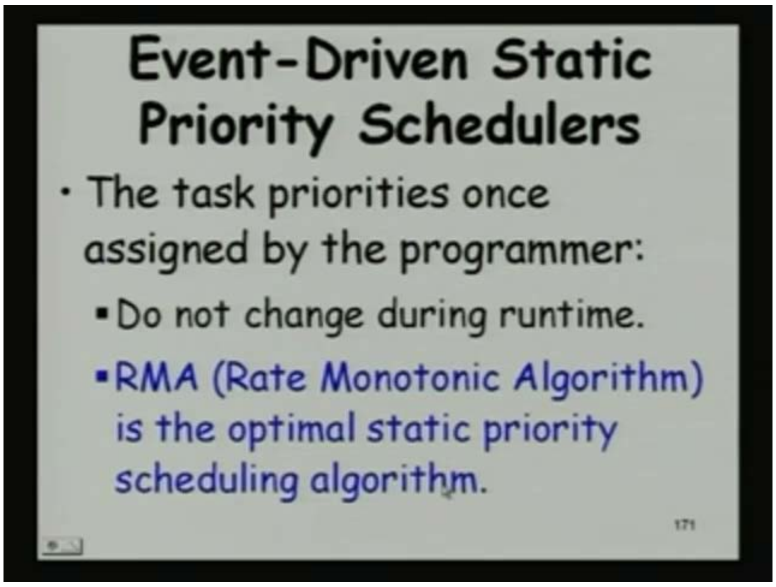
So, the background task effectively gets two 1 is taken out here, 1 is taken out here. So, it is getting every 100 millisecond, it is getting 48, instead of 50 it is getting 48.

So, 2000 divided by 48 20 what is that 1020. So, if we find a multiple of 48. It will be 480,960 and then 1920 and 48 so 21 that becomes 2050 is not it. So, 1968 and we have another 32 plus 20 so another 52 to run. Now it should have been 2100, is not it 2100 would have made it plus 48, 2016. So, another 4 it needs to run and when it runs 4 it will also incur. So, the situation is that it has run up to here and then another 4 has not run the last one is this is 52 times is not it no not 52, 22 times.

And then 21, 2100 times 100 into 21, 2100 and then another 4 is remaining. So, 50 will run here the task real time task will run 50, foreground will run 50 and this will run another 4 here. So, this is one taken out one taken out here. So, 52 plus 4 56. So 2156.

So, it is term it is keeping on getting 48, out of 100. So, we will find out when and last frame we need to be careful little bit, because the foreground task will come and start running here, and it will involve a context switch and then the remaining part 4 or something was remaining that will run, but again a context switch will be involved. The last one you need to be bit careful is that simple problem.

(Refer Slide Time: 28:16)



Event-Driven Static Priority Schedulers

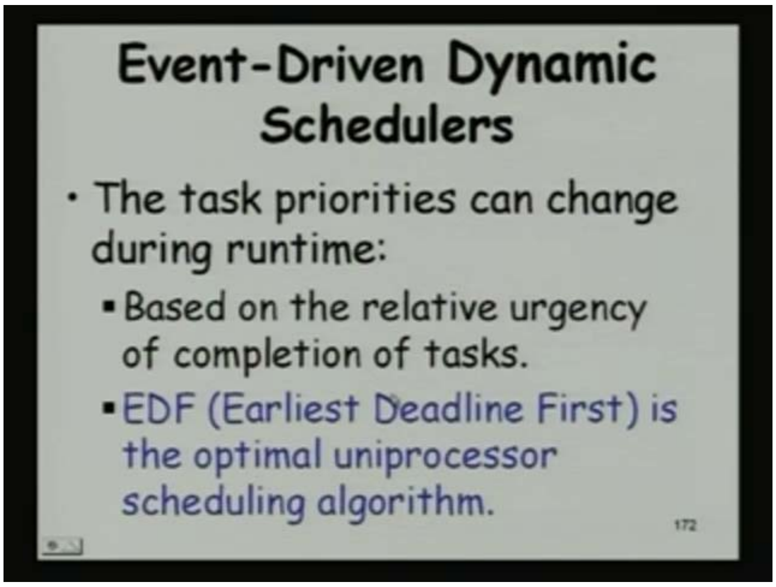
- The task priorities once assigned by the programmer:
  - Do not change during runtime.
  - RMA (Rate Monotonic Algorithm) is the optimal static priority scheduling algorithm.

171

So, far we did not really discuss about the schedulers that are actually use the event driven schedulers. So, we will see that we will first look at the static priority scheduler. We call it static priority schedulers, because the priority will be assigned by the programmer before the system starts.

And during the run time the priorities do not change. Whatever was assigned by the programmer, during the design time that will continue to hold and here, this rate monotonic algorithm is the optimal static priority algorithm, extensively used it is a very important one.

(Refer Slide Time: 29:09)



Event-Driven Dynamic Schedulers

- The task priorities can change during runtime:
  - Based on the relative urgency of completion of tasks.
  - EDF (Earliest Deadline First) is the optimal uniprocessor scheduling algorithm.

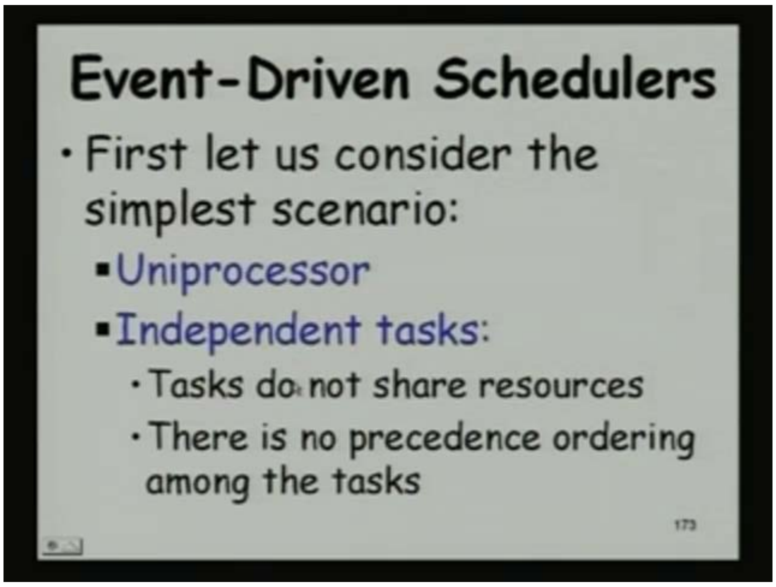
172

So throughout, we will fall back on this basic concept of a static priority computed by the programmer does not get changed. And then the other is the dynamic priority algorithm, which is the earliest deadline first.

So here the priorities can change during run time as the task, the system runs some tasks becomes more higher priority and another one which was earlier higher priority, it is priority might reduced depending on what is the dead line. So, based on the relative urgency of completion of tasks the priority is keep on changing as the system runs and many algorithm exist which a dynamic priority schedulers, but if we know EDF.

We know the best algorithm that is possible, it is the optimal uniprocessor scheduling algorithm and earlier we had said, optimal is one if it can run some tasks then the other dynamic priority schedulers they may or may not run it, but if another dynamic priority scheduler is able to run some task, then definitely EDF will be able to run it.

(Refer Slide Time: 30:29)



Event-Driven Schedulers

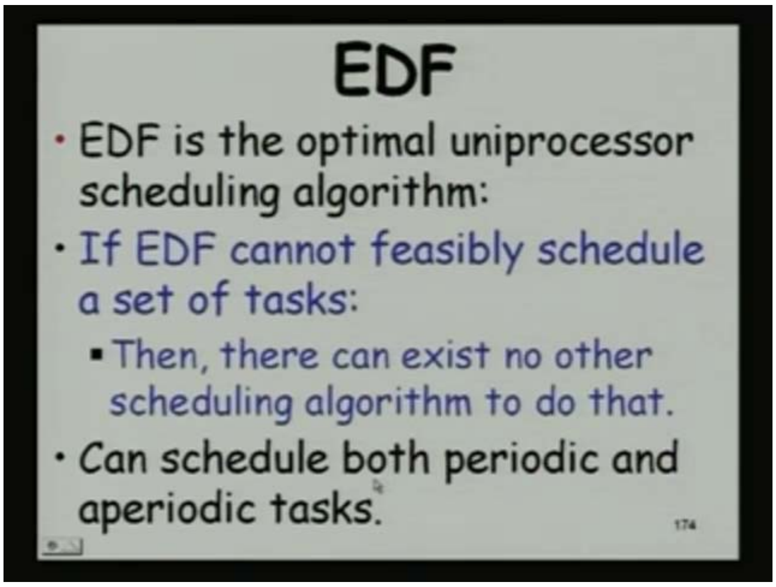
- First let us consider the simplest scenario:
  - Uniprocessor
  - Independent tasks:
    - Tasks do not share resources
    - There is no precedence ordering among the tasks

173

So, let us look at the simplest scenario when we have independent tasks, a task does not need result from another one no resource sharing between tasks and no precedence ordering, that is one task does not need to wait for another task to complete.

This is not really a practical situation that we have tasks which are independent not needing results from each other no constraints among them, but to understand the scheduler, we need to start with this and as we proceed we will make it more complex where resource sharing etcetera exists.

(Refer Slide Time: 31:12)



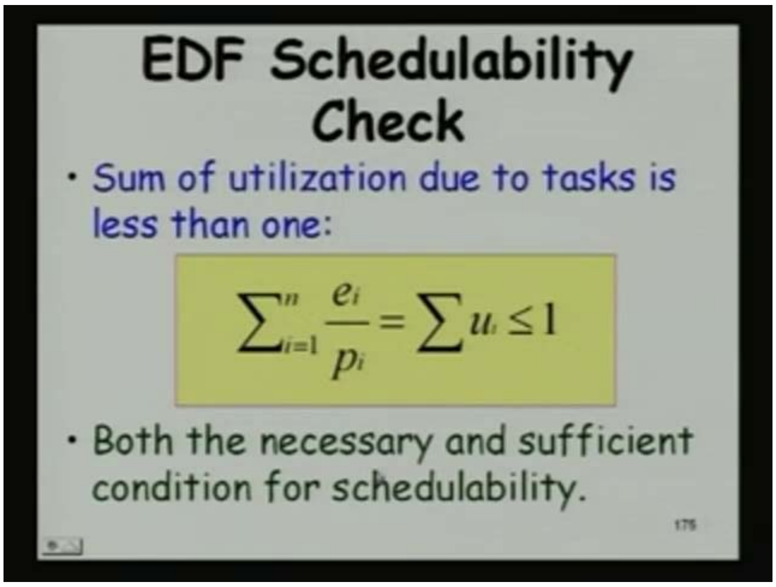
EDF

- EDF is the optimal uniprocessor scheduling algorithm:
- If EDF cannot feasibly schedule a set of tasks:
  - Then, there can exist no other scheduling algorithm to do that.
- Can schedule both periodic and aperiodic tasks.

174

So, this is the optimal algorithm if it cannot schedule some tasks then it is no use trying out other tasks, other schedulers something is not schedulable by EDF, because it is the optimal algorithm then other algorithms we cannot run this.

(Refer Slide Time: 31:40)



EDF Schedulability Check

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

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

- Both the necessary and sufficient condition for schedulability.

175

Handles both periodic and aperiodic tasks and given a set of task the check for schedulibility here is that a sigma e i divided by p i is less than 1. So, the utilization of the processor due to all tasks as long as it is less than 1 very natural, because you know if the number of the time require to the run is more than 1. We cannot run on any processor utilization 1. So, the constraint is actually nothing other than the common sense how much a processor can run.

So it basically we can say that, if a task set can (( )) the run then EDF will be able to run it. So, this is both the necessary and sufficient condition for schedulability. I think we can say that, it is a sufficient condition for schedule I think necessary is not there.

If it is this is satisfied then definitely it will be able to run, but it is not necessary that. What do we think is it necessary?

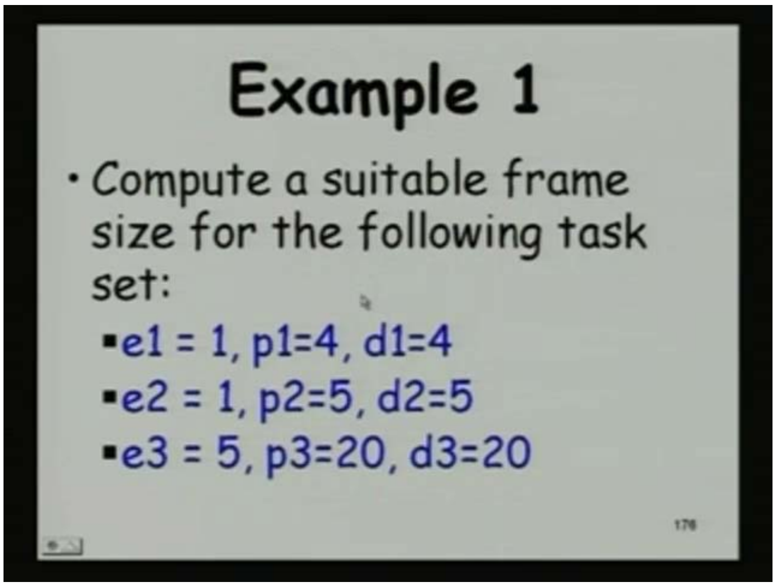
This condition is necessary here.

Ok.

Second condition.

It is both necessary and sufficient it is a necessary condition and also a sufficient condition fine.

(Refer Slide Time: 33:18)



Example 1

- Compute a suitable frame size for the following task set:
  - e1 = 1, p1=4, d1=4
  - e2 = 1, p2=5, d2=5
  - e3 = 5, p3=20, d3=20

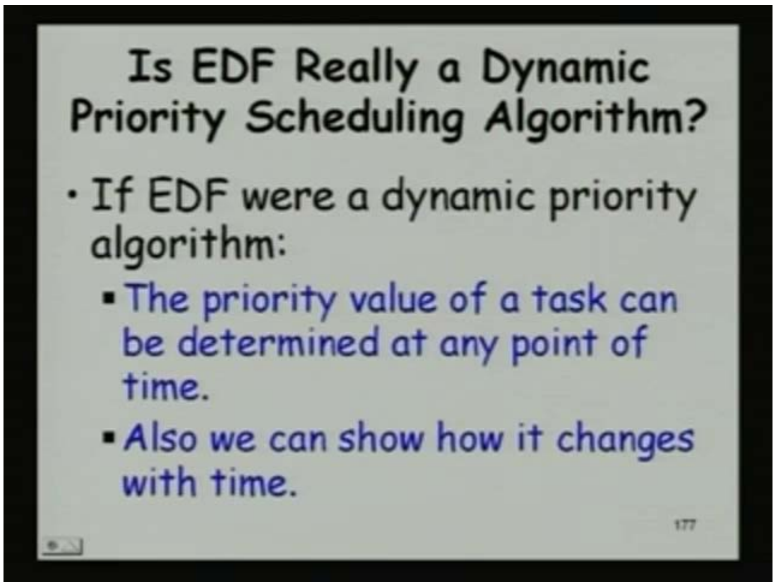
176

So, let us just look at an example whether EDF can run this. So, there 3 tasks periodic tasks and the period and deadline are the same, for each of them one is period 4, another 5, another 20 and the time is run time required execution time required is 11 and 5.

So, we just need to check the utilization for each of these tasks. So, this is 1 divided by 4 is 0.25, the first one. Second one is 0.2 and the third one is how much 0.25, say 0.25 plus 0.25 plus 0.2 is 0.7 less than 1.

So, less than 1 and this is schedulable an EDF.

(Refer Slide Time: 34:16)



Is EDF Really a Dynamic Priority Scheduling Algorithm?

- If EDF were a dynamic priority algorithm:
  - The priority value of a task can be determined at any point of time.
  - Also we can show how it changes with time.

177

We said that, EDF is a dynamic priority algorithm and if it were a dynamic priority algorithm we should be able to tell how the priority values, what is the priority value of a task, at any instant and also how this change with time and say that see at this point see it was 10 and then it has become 5 and so on. Then we can call it as a dynamic priority algorithm, but what about EDF. We just saw that the algorithm is simple that any task which has the shortest deadline, earliest deadline will be executed.

So, where is the concept of a dynamic priority here, anybody would like to answer this question?

As the deadline of the as well it will comes even for the task.

Yes.

Then it is priority should be increased earlier said that.

No, but what is that priority?

(( )) when a new task arise?

Yes.

If it is deadline is very nearer to the previously into the previous task, but it (( )) to a higher priority, we have the state sir at the every scheduling point.

Yes.

We will find out how much of the execute 12, on time is left (( )).

And then what is the priority? How will the priority be computed here?

Sir, whichever task is supposed to scheduled at first.

Yes.

That will be given the highest priority from at every scheduling point.

Suppose all the task has arrive?

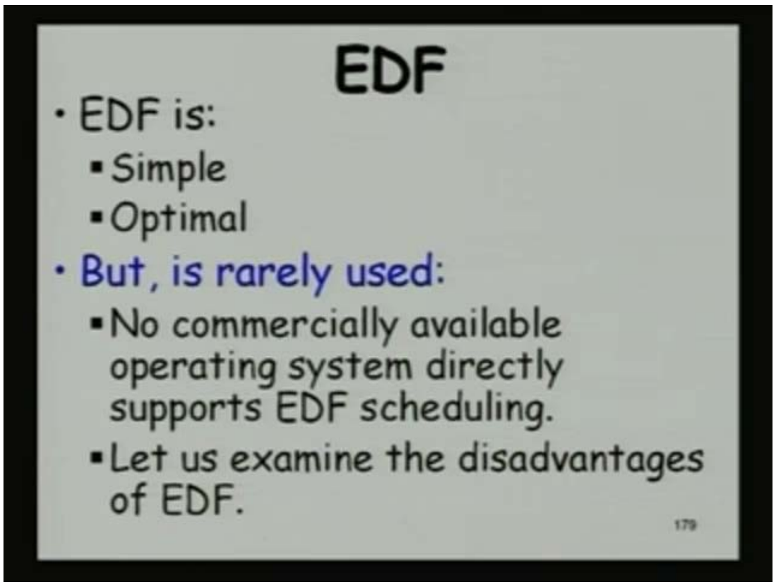
So, a task.

So, what basically he says that a task arrives its keeps on waiting, until its deadline is very near and as it keeps on waiting its priority increases, we can think of it.

Another task which was running and next time it came possibly its deadline was quite far and then the task which was earlier waiting is running. Because now that task has become a higher priority is not it.

Yes, the longer task waits in ready queue the higher is the chance being taken up for scheduling according to the EDF. So, even though we cannot really assign a concrete value saying that, it is priority is 20 or 30 or something, but we can imagine a virtual value which keeps on increasing with time until it is scheduled. And the next time it comes as a lower priority and as its deadline approaches or if it deadline is already low is less deadline is very less than its priority, will be increased.

(Refer Slide Time: 37:13)



EDF

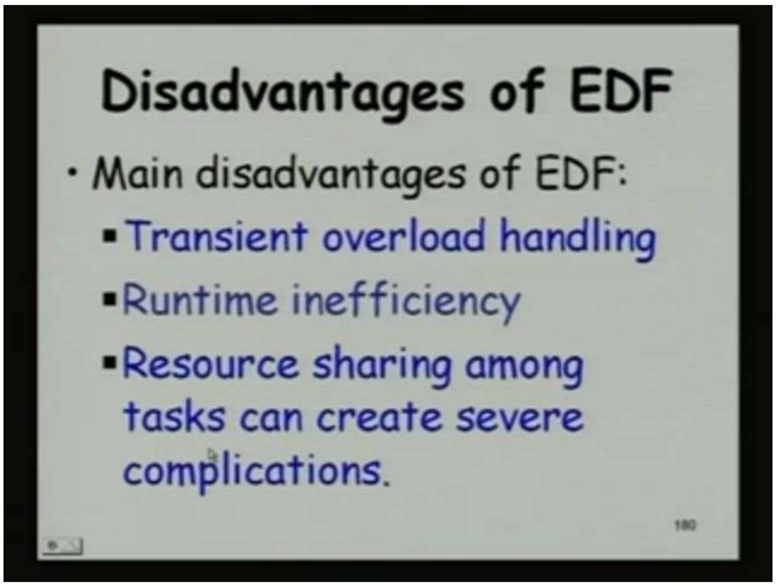
- EDF is:
  - Simple
  - Optimal
- But, is rarely used:
  - No commercially available operating system directly supports EDF scheduling.
  - Let us examine the disadvantages of EDF.

179

Very simple algorithm just examine the dead line of every task, find out the one which has the lowest dead line and then run it. And it is also the optimal one, we are not going to prove, why it is optimal the proof is slightly involved actually will not prove it, but it is the optimal one, but it is rarely used as you will see the commercial operating systems and the systems which use any operating system at all.

See that EDF is more of a concept rather than anything actually used. None of the commercial operating systems actually support EDF schedule directly. So, there must be some major disadvantages, even though it so nice properties here very simple, efficient, also optimal must be some disadvantages. Otherwise it could have been used heavily.

(Refer Slide Time: 38:23)



Disadvantages of EDF

- Main disadvantages of EDF:
  - Transient overload handling
  - Runtime inefficiency
  - Resource sharing among tasks can create severe complications.

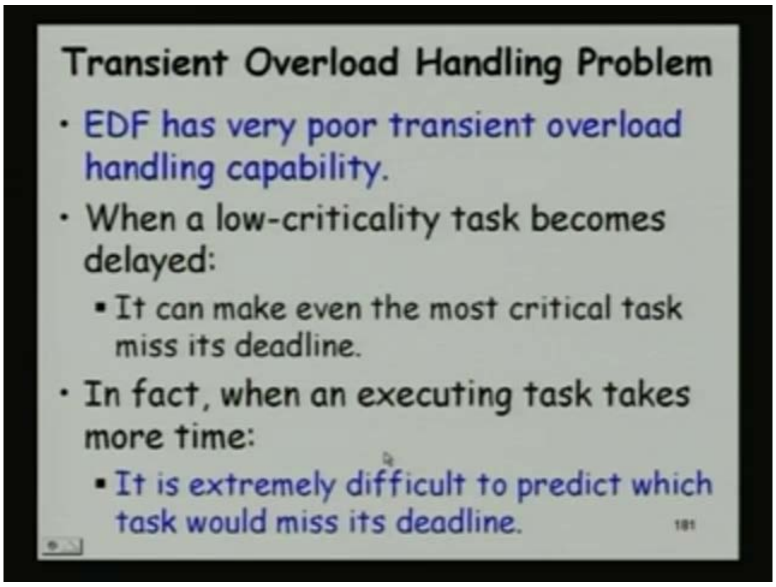
180

Let us see the disadvantages of EDF many disadvantages, but three main disadvantages: the first disadvantage is called as transient overload handling, second is the runtime inefficiency, the third is resource sharing of tasks creates severe complications.

Let us look at each of these point. The first sort coming is the transient overload handling problem EDF, see transient overload we had said earlier that sometime the events just occur together. For example, there is a fire situation and then the fire alarm assigns and it just set up a avalanche lot of events occurred immediately, after that it rang a bell and then the water this thing sprinkler are switched on and so on. Many events could happen at the same time.

So, these are the situations where suddenly many tasks need to run and that is called as the transient overload.

(Refer Slide Time: 39:38)



Transient Overload Handling Problem

- EDF has very poor transient overload handling capability.
- When a low-criticality task becomes delayed:
  - It can make even the most critical task miss its deadline.
- In fact, when an executing task takes more time:
  - It is extremely difficult to predict which task would miss its deadline.

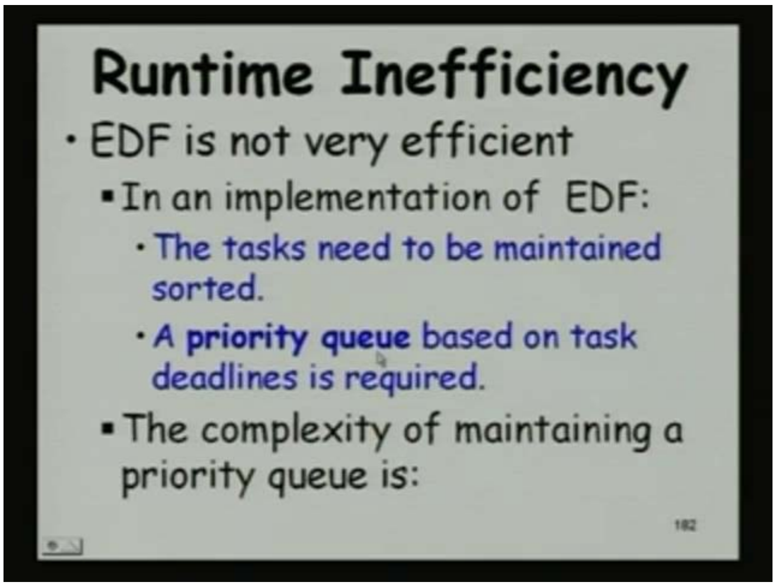
181

So, when a transient overload occurs then EDF can miss a deadline. And if it misses a deadline that is not too bad actually, but the main problem here is that the highest priority task might miss its deadline. It does not guarantee anything.

A low criticality, low priority task might run to completion, whereas high priority task might miss deadline. Another situation let us consider this that as for some reason, a low priority task is getting delayed may be it has gone into a loop or something.

So, in that situation also a higher priority task can miss deadline. So, whenever there is a transient overload, it is very difficult to predict which task will miss its deadline.

(Refer Slide Time: 40:49)



Runtime Inefficiency

- EDF is not very efficient
  - In an implementation of EDF:
    - The tasks need to be maintained sorted.
    - A **priority queue** based on task deadlines is required.
  - The complexity of maintaining a priority queue is:

182

The second problem is that EDF is not very efficient, just let us consider an implementation of EDF here you see a simple implementation may be we keep all the tasks sorted according to their deadline. And then we select that is shortest deadline.

So, each time an event occurs that is task completion or arrival we just adjust the queue of the tasks and we just select the one at the front of the queue, we can use a priority queue, on which we maintain the tasks based on their deadline. The shortest deadline as the highest priority is at the top of the queue.

What is the complexity of maintaining a priority queue? Yes, you have done. How is a priority queue implemented? Let us say, you are doing a C program or how you implement a priority queue?

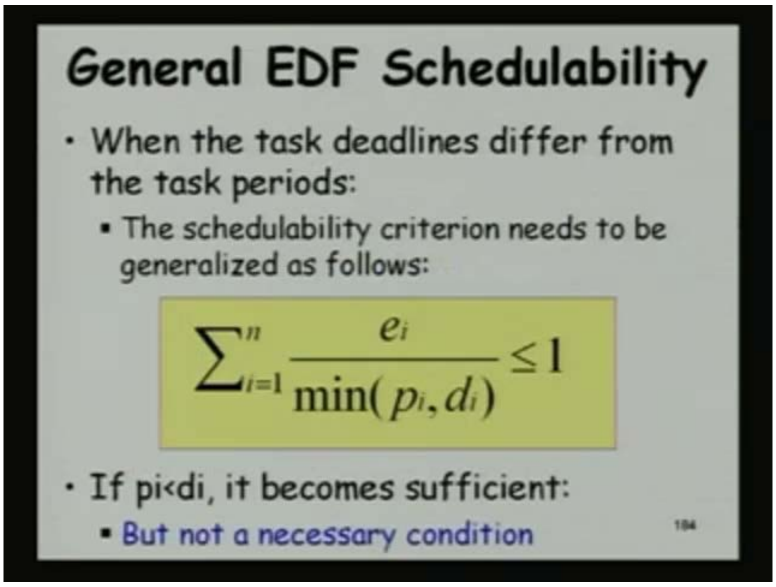
Yes, anybody knows how to implement a priority queue efficiently? The most efficient way priority queue can be implemented. Let us say you are writing a C program or something. We can use a heap.

A heap is the most efficient implementation of a priority queue and log(n) to the base 2 where, n is the number of tasks is the complexity of maintaining a priority queue, if you brush up your knowledge and the basic data structures and algorithms. You will see that priority queue is implemented using a heaped and that is the most efficient implementation of a priority queue and log(n) is the complexity of maintaining a priority queue and each time an event occurs we need to adjust the priority queue. So, it is not very efficient, because that overhead will be incurred every event occurrence.

The third problem is the most severe one also actually is the resource sharing among tasks. We had said that it will be very unlikely that you will have a system where the tasks do not share resources. All practical situations, we will see that the task share resources as we discuss further we will see that.

And here in this EDF, because very difficult to allow tasks to share resources. For example, the critical sections if we use EDF and then we have this resource sharing requirement then we will see that the tasks will miss their deadline. We will have the discussion little later that how resources be shared, because so far we have been discussing the tasks are independent do not share resources and so on. We are soon going to discuss about how resources, can be shared we will generalize our discussion now that independent tasks not sharing resources etcetera. We will generalize that, and then we will see that EDF would make it extremely difficult for task to share resources. So, let us proceed.

(Refer Slide Time: 44:33)



General EDF Schedulability

- When the task deadlines differ from the task periods:
  - The schedulability criterion needs to be generalized as follows:

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

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

184

We had seen the situation where the execution time and the period the period and the deadline were the same, but we can have sometimes a situation where the task period is different from its deadline and then we need to generalize our earlier expression. That is sigma e i divided by p i less than 1. We have to make it sigma e i divided by minus of p i divided by d i.

So, if the deadline is less than p i then we will have to choose the deadline value here, but if p i is equal to d i it does not really matter. So, this is the new expression when.

What will happen? If p i is greater than.

Ok.

Sorry p i.

p i it is less than d i.

So, that when p i is less than d i then this becomes a sufficient condition that means if this is satisfied then definitely it will be a schedulable, but not a necessary condition even if it is not satisfied we will see that it can be schedulable so.

As it is occurring before, the previous instance has completed. No, it has a dead line is occurred after the next instance has occurred.

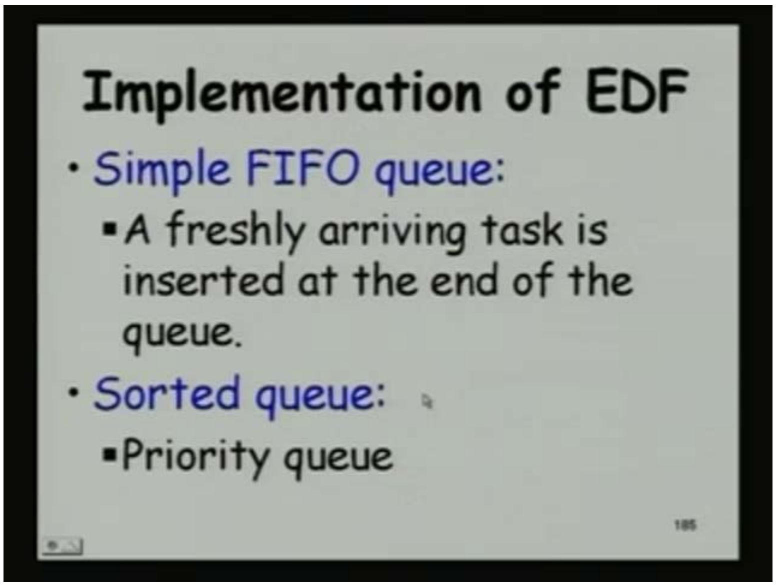
d i is greater than p i.

Greater than p i.

So, the deadline is even after the next occurrence is occurred.

So, in that case this is a sufficient condition. If this holds definitely they are schedulable, but if it does not hold we cannot say that they are not schedulable, we will have to check out.

(Refer Slide Time: 46:32)



Implementation of EDF

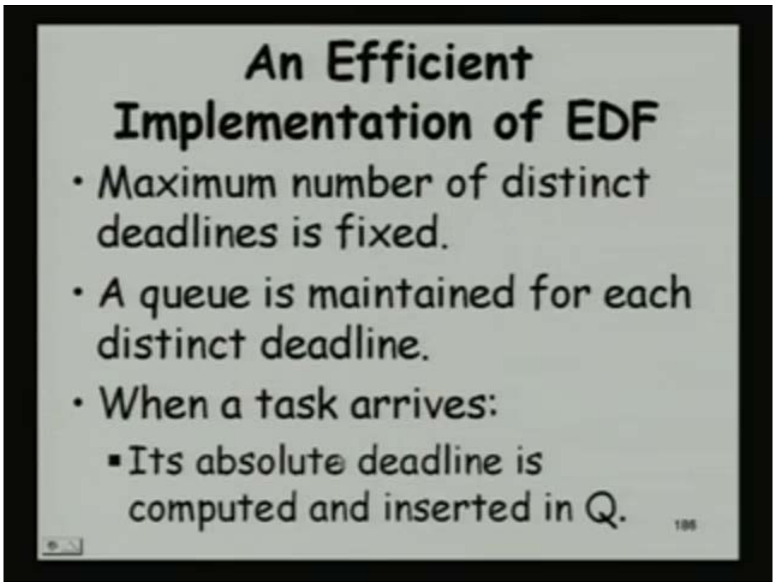
- Simple FIFO queue:
  - A freshly arriving task is inserted at the end of the queue.
- Sorted queue:
  - Priority queue

185

So, this we had seen that not a very efficient implementation. We will have to just to maintain a queue, first in first out queue, but here each time selecting the smallest task, the task be the smallest deadline. We will have to scan the queue, once that is log(n) that is n is not it each time, we have to look at everyone every task n this can do it in log(n) you need to maintain the queue.

Here, you do not maintain the queue you just have to look up time takes more.

(Refer Slide Time: 47:12)



An Efficient Implementation of EDF

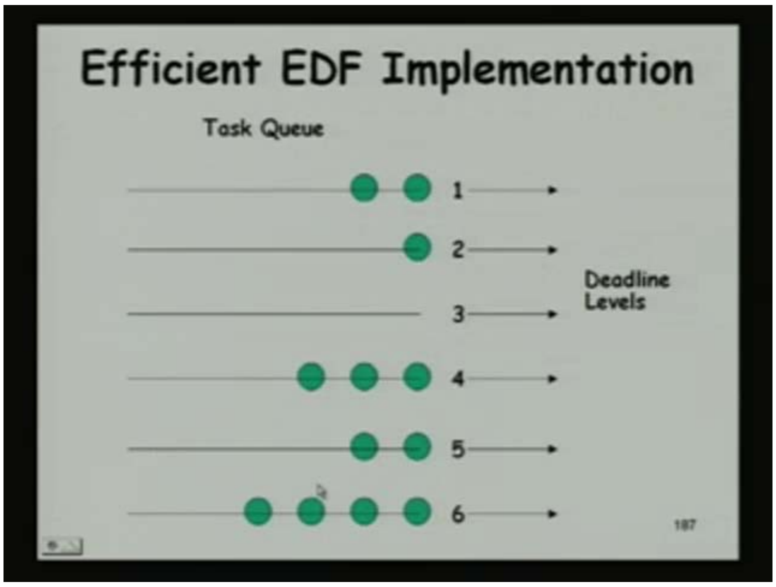
- Maximum number of distinct deadlines is fixed.
- A queue is maintained for each distinct deadline.
- When a task arrives:
  - Its absolute deadline is computed and inserted in Q.

186

But, still a more efficient implementation of EDF can be possible where you fix the maximum number of distinct deadlines, whether it is a 5 or 7 distinct deadlines. You can give let us say 0 to 10 and then 10 to 20 and so on.

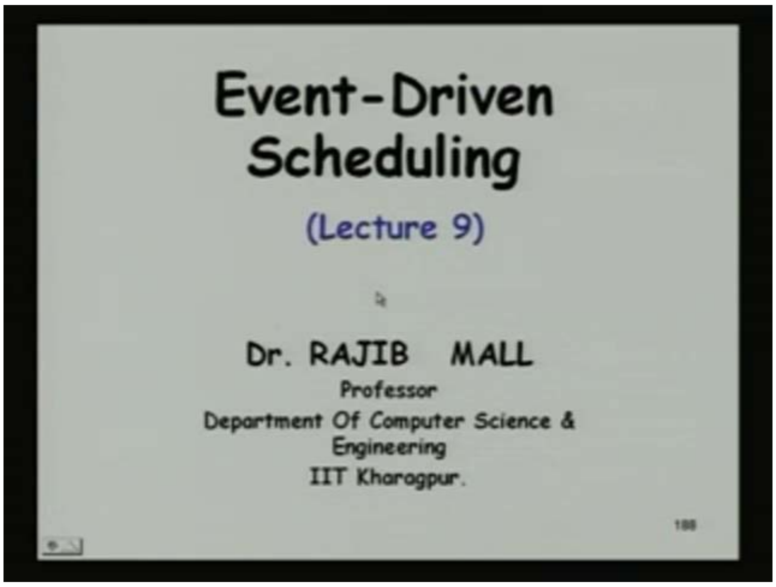
And then a queue is maintained for each distinct deadline and the top of the queue has the highest. I mean the deadline for that band lowest deadline so when a task arrives its absolute deadline is computed and inserted into queue. So, you just have distinct set of deadlines and each time you just need to manipulate only the first few tasks.

(Refer Slide Time: 47:53)


Efficient EDF Implementation

Check it the deadline. So, to run it you just need to take this one, the highest on, the top here and then you need to look at the top one here, and just put them in the appropriate levels. So, this will be a much more efficient implementation and of course each of them will be a priority queue.

(Refer Slide Time: 48:33)


Event-Driven Scheduling

(Lecture 9)

Dr. RAJIB MALL

Professor

Department Of Computer Science & Engineering

IIT Kharagpur.

188

So, we will stop here. If you have any questions we can just discuss about that, otherwise we will stop here. So, any question so far we looked at the dynamic priority scheduler that is the EDF and we said that EDF is the optimal algorithm and but it is not really used very much practically and then we identified the reasons. So, what were the reasons anybody remembers.

Transient overloading.

Yes, it does not if there is a transient overload then EDF will have problem.

Inefficient planning datas.

Yes, it is inefficient implementation is inefficient if the number of tasks is large then the time the scheduler will take to look through the task and select the one. That is need to run is high and also to arrange the queues each time and any other.

Sharing of designing there.

Yeah. So we had said that, resource sharing is a occurs in every practical situation and in EDF if we try to extend it see. So, far we have been looking at independent tasks, but when we try to extend this to handle situations where resource is shared we will see that it will become extremely inefficient and tasks can miss their deadline.

So, we will stop today and in the next class we will discuss about the static priority algorithm, that is the rate monotonic algorithm the optimal algorithm heavily used simple algorithm again and we will spent more time on that see EDF. We really did not spend more time, because after all it is more for our reference and it is the optimal scheduler if EDF cannot schedule nothing else can schedule.

But RMA is the one which is used extensively every commercial operating system. As I means to support at least RMA if nothing else or various of RMA. So, we will see the RMA and how resource sharing is done, how task precedence can be handled and so on. We will spend at least 4, 5 hours discussing RMA. So, we will stop here, thank you.