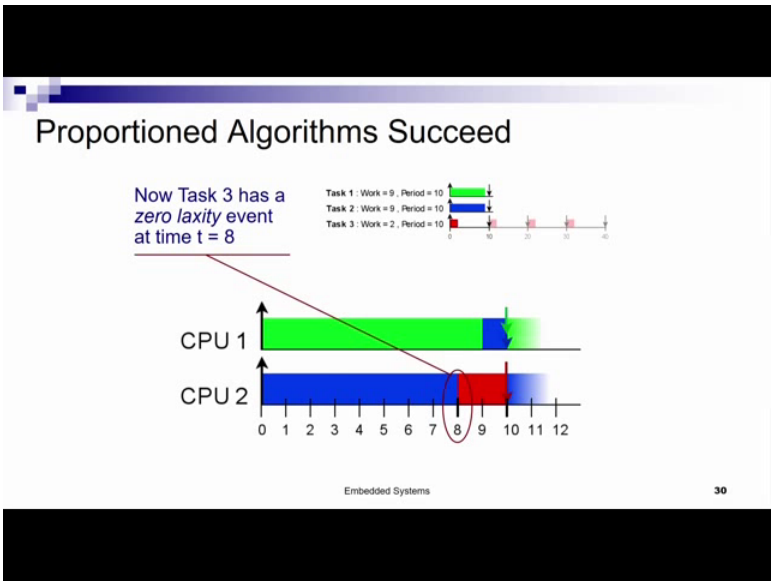
Embedded Systems – Design Verification and Test

Dr. Santosh Biswas
Prof. Jatindra Kumar Deka
Dr. Arnab Sarkar
Department of Computer Science and Engineering
Indian Institute of Technology, Guwahati

Lecture – 18
Introduction and Basic Operators of Temporal Logic

Hello. In this lecture, we continue with our discussion on multiprocessor scheduling algorithms for real-time periodic independent task sets, those are used in embedded systems. And we saw in the last example that algorithms which allow proportionate execution progress of tasks in the system succeed on multi processors in providing high resource utilization.

(Refer Slide Time: 01:00)

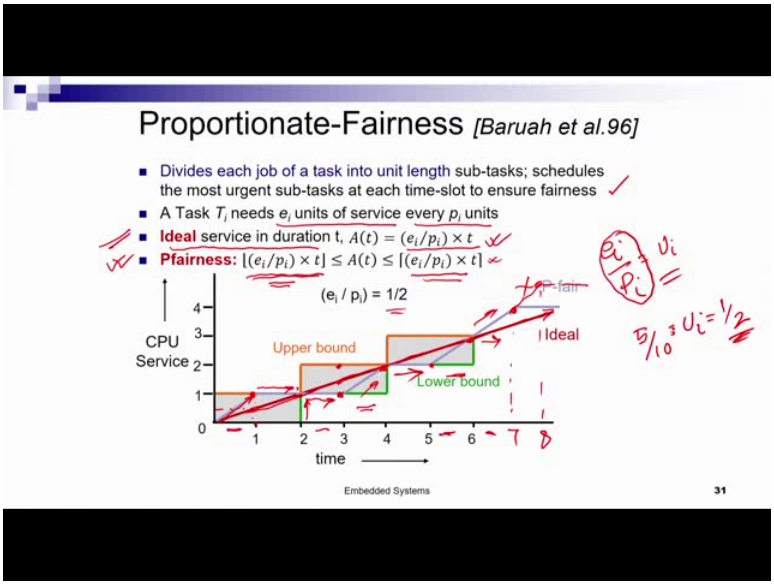


We saw how greedy algorithms such as EDF which allow the tasks with the earliest deadline at in scheduling point to execute, or LLF which allow the tasks with the least laxity to execute at any scheduling point fail on multi processors, but tasks which allow proportionate progress of execution. For example, if in the last example we saw that every 10 times lots, task 3 is in task 3 must execute 2 parts of it is overall 8 units of execution requirement in order to this such a proportionate progress allowed it to complete it is deadline by time 40 and thus as we saw that probe algorithms allowing

proportionate progress proportional it execution progress of tasks to succeed on multi processors.

The first algorithm that was designed on this idea of a proportionate execution progress was the p fair algorithm designed by Sanjay Borough of a back in 1996. And it was the first multiprocessor scheduling algorithm for real time periodic tasks which allowed full resource utilization.

(Refer Slide Time: 02:13)



We will now have a deeper look into this algorithm. So, this algorithm worked works by dividing each job of a task into unit length subtasks where a sub task is that quantum of work that can be completed within a timeslot of execution provided by the operating system. So, a timeslot of execution is the least amount of time that can be shared that that is given by the operating system. So, within a time slice only one task can execute. Across time slice at the boundary of time slice, we can choose which task should be executed next. So, sub task is that amount of work that can be completed within the duration of one time slice, or sorry one-time slot, ok.

Now, now P fair this P fair algorithm progresses by scheduling the most urgent subtasks at each time slot to ensure fairness. So, p fair algorithms how does it proceed it finds out among all tasks at a given time which of the subtasks of the task has the most urgent has the most urgency, ok. Based on that it defines which it finds out which sub task scheduled next at any given time slot boundary.

So, therefore, what we are saying we have jobs of tasks, and the jobs of tasks are divided into sub tasks the sub tasks are sequentially numbered from 1, 2, 3, 4 likewise. And we find out few sub tasks of each task at any given time of few sub tasks of each task has already been executed. Now we have the next sub tasks of each task to execute. Among those next sub tasks, the p fair algorithm finds that sub task or that set of sub tasks which are the most urgent, and schedules them on the m available processors in the system, ok.

And p fair scheduling or disproportionate fair share scheduling proposed by Borough as it is called P fair it, this P fair algorithm says that each task must executes you must complete e i units of service every p i time units. In fact, it says that each task should progress based on it is based on the ratio e i by p i. So, in proportion to the ratio e i by p i which is also called it is utilization u i

So, the execution should progress in proportion to this value e i by p i. Therefore, ideally the ideal rate at which the service should be provided to any task t i is given by e i by p i into t. So, let us say that this e i equals to 5 and this p i equals to 10. And therefore, e i by p i so, therefore, in this case u i equals to 1 by 2. So, the ideal rate at which execution should progress is given by e i by pi into t. So, after 1 unit of time the ideal progress says that the task t i should complete half of it is sub task.

By time unit 2, it should complete one part of it is sub task, one sub task. So, by time unit 3 it should complete one and a half sub task. By time unit 4 it should complete 2 sub task. By time unit 5 it should complete 2.5 sub task. And by time unit 6 it should complete 3 sub task, why? Because it is utilization is half. So, the rate at which it should be provided service is at the rate of e i by p i into t.

However, we said at the beginning that within each time slot only ones 1 task can execute. So, therefore, either I will give the processor for the entire duration of one-time slot to a given task, or I will not allocate that processor to that task. So, a sub task will either fully execute, 1 unit of the sub test will execute or the sub task will not execute at a given time slot. Therefore, this ideally execution rate provided by this p fair this ideal execution rate cannot be followed practically in practice. So, p fair told that you fall you find you cannot follow this execution rate ideally, but what can you do is, you can approximate this approximate this ideal service rate as closely as possible.

So, what it said is that, you execute at least you cannot execute these many subtasks at every time slot fine. But you can ensure that you will at least execute e i by p i into t, floor of e i by pi into t sub tasks which is an integer and our ceiling of e i by p i into t which is also an integer. So, the amount of allocation the amount of work for t i that should be completed at time t should be upper bounded by this much, this value and should be lower bounded by this much, ok.

So, what happened is that, at the end of time one therefore, at the end of time one we can at most complete 1 unit of execution or we can complete we may not give this we may not give this processor at this time slot to the to time i in that case it will only have received 0 units of execution. Why this is possible? Because at the end of 1, I can the floor of e i by p i into t is 0 because it is half. So, e i by p i into t is half at the end of 1. So, floor of e i by p i into t is 0. So, you are allowed to complete 0 units of execution to ensure P fairness.

And you can add the most complete 1 unit of execution because ceiling of e i by p i into 1 is 1. So, you can complete at most one and you must complete at least 0; however, at time 2 what happens at time 2 this floor this floor is half into 2 which is one. So, the lower bound is 1 here and the upper bound is also 1. So, at the end of 2, you can exactly you must exactly complete 2 units of execution. At the end of 3 what happens at the end by the end of 3, you see that the see the floor of e i by p i into t is given by 1.5. So, sorry e i by p i into t it is so, e i by p i is half into 3 is 1.5 so, a floor of 1.5 is 1.

So, you must complete 1 unit of execution, and you can at most complete 2 units of execution, because ceiling of 1.5 is 2. By the end of 4 you must complete 2 units of execution and likewise. This is what is ensured for the p fair algorithm. So, what am I allowed to do? So, it suppose I the actual allocation which satisfies an actual schedule which satisfies P fair must guarantee this for all tasks in the system. And sub this is a perfectly this blue line is a perfectly valid p fair schedule. So, this has said that it allow it allocated it allocated the processor to t i at time between 0 to 1 and completed 1 unit of execution at by 1. It did not allocate the processor between 1 to 2.

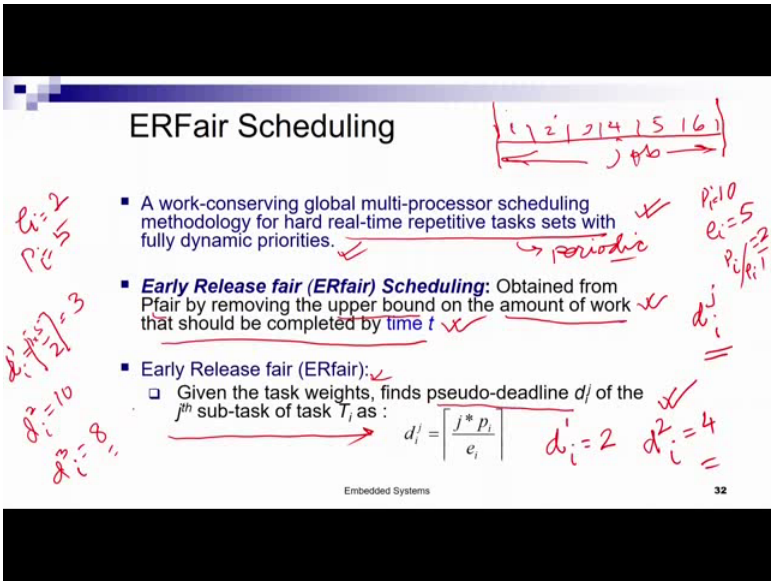
So, the amount of work completed remained at 1 between 1 to 2 and then it did not allocate the processor between 2 and 3. And so, at 3 the amount of work completed was 1. This is also perfectly valid because it satisfies the lower and upper bounds. Between 3

and 4 it again allocates the processor. So, by 4 it completes 2 units of execution, lower and upper bound is satisfied absolutely valid likewise, it does not allocate the processor between 4 and 5.

So, at 5 it has just completed 2 units of execution. It is still valid lower and upper bound satisfied, between 5 and 6 the processor is allocated to t i. So, at 6 it completes 3 units of execution, then it again allocates the processor at 7 to this processor. Let us say, and it completes it completes 4 units of execution by 7.

And this is also perfectly valid, because both the lower and upper bounds are satisfied. So, between 5 and 6 and between 6 and 7 the processor is allocated 2 consecutive times to this to this task which has an utilization of half. And the lower and upper bounds are satisfied and this blue line is a perfectly valid p fair schedule; however, let us say if this if the processor is again allocated to this task at this time slot 8, it will complete it will complete, how many? It will complete 3 4 5 units of execution by 8 which is invalid. And this will make it and invalid p fair schedule, ok.

(Refer Slide Time: 12:35)



With this understanding we proceed to ER fair scheduling which is a derivative of the P fair scheduling. In fact, it is a work conserving version of the P fair scheduling meaning that it does not allow a processor to idle if we have ready tasks to execute, ok.

So, in the previous in the previous algorithm in the P fair algorithm, we saw that the processor can remain idle if this upper bound is reached. Let us say you have no other tasks in the system and you have completed 1 unit of execution by time 1. You cannot allocate the processor to you cannot allocate the processor to t i, because it has already completed 1 unit of execution by time 1. And between 1 and 2, you cannot allocate the processor 2 t i to meet this because this upper bound will be violated.

Now, if there are no time load tasks to execute in the no other task to execute processor, the processor just remains idle between this time at this time, ok. Now if you have ready task if the sub tasks have arrived in the system and the tasks and the sub tasks are all there in the system while a task is ready to execute not allowing the task to execute is not good. Because the processor just remains idle doing nothing.

Your fair removes this constraint. So a work conserving global multi-processor scheduling methodology for hard real-time repetitive tasks in fact, for periodic tasks in fact, for periodic task sets with fully dynamic priorities. Meaning that the subtasks the priority of the; relative priority of the subtasks of a task change over time, ok.

Because the urgencies of the subtasks change so, the jobs have not only the jobs have has dynamic changing priorities, but even the priorities within a job of a task can change each sub tasks can have a distinct relative priority with respect to other tasks. And hence, it is an algorithm with fully dynamic priorities. So, ER fair scheduling is obtained from P fair by removing the upper bound on the amount of work that should be completed by time t, ok.

So, this part still remains, but this one this upper bound is removed. So, it is just says that it must complete this amount of execution. So, it must complete 2 amounts of it must complete at least 0 units of execution by time 1. It must complete at least 2 units of execution by time 2 it must complete at least 2 units of execution by time 3. It must complete at least 3 units of sorry, it must complete at least 1 unit is 0 unit of execution by time one it must complete at least 1 unit of execution by time 2. It must complete at least 1 unit of execution by time 3. It must complete 2 units of execution by time 4 and likewise.

However, the upper bound can be anything. It can complete say the processor can be continuously allocated to task I for this 4 consecutive time slots say and the task is

perfectly eligible to complete 4 units of work by time 4, if the processor has no other tasks to do other than allocating itself to this task, ok. So, ER fair allows this P fair does not

Now, how does P fair guarantee this that that how does P fair find out the urgency we said that how P fair finds out the urgency among all other tasks is something that, we have not looked at we say if, we found out how what P fair guarantees, but how does it guarantee we have not seen so, this is what we will see now. So, given a given the task weights P fair finds pseudo deadline d j i of the jth sub task for task I, I said that the sub task of a task are numbered. So, if these are the sub task of a task 1, 2, 3, 4, 5, 6. And let us say these 6 sub task conseque complies the job of the first one job of this task.

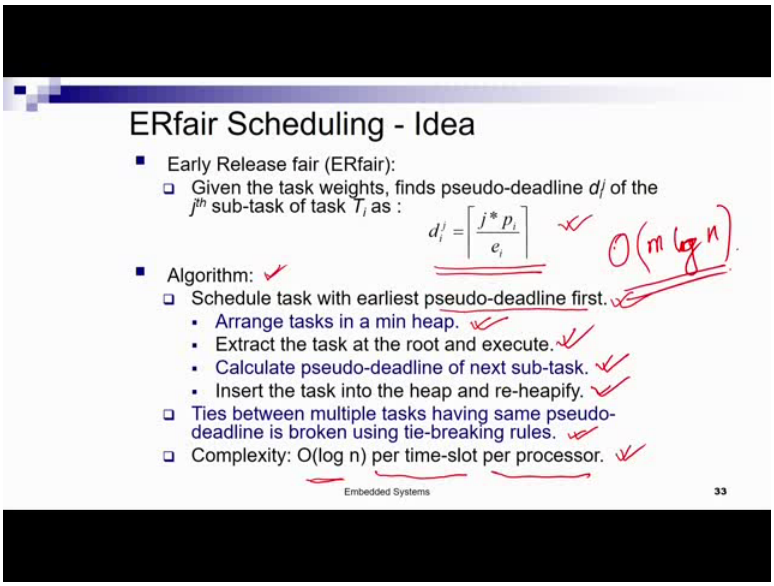
And these jobs repeat, ok. So, then so, this one is subtask 1, this one is sub task 2 1, this one is sub task 3 off task i. So, the jth sub task of tasks i, the deadline for the jth sub task of task i is denoted d j i so, d j i is the jth sub task of task i.

Now, this pseudo dead line says that you must do this. You must complete the first job sorry, the first sub tasks by what time? P i by e i with respect to the previous algorithm we had p i equal with respect to the previous example, we saw that for that task t i we had p i equals to 10, p i was 10 and e i was e i was 5. So, p i by e i equals to 2 by 1, ok.

So, it says that the first job should complete by 2. So, j is the sub task number. So, d of 1 i equals to 2. Why? Because 1 into 2 into 10 by 5 or 1 into 2 by 1 which is 2. Now the second job must complete by 2 into 2 by 1 which is 4. So, d 2 i equals to 4 likewise. So, if we say if you have another task let us say we have e i equals 2 let us say 2 and we have p i equals to, let us say 5. So, it is 2 by 5 so, the first job should complete. So, for this task d 1 i equals to what one into 5 by 2. So, 5 by 2 1 into 5 by 2 ceiling, which is equals to which is equals to 3, ok.

So, the first job should complete by 3. The second job should complete by what 10 by 2 which is 5. So, d 2 i equals to 10 5 into 2 by 2 which is 10. The third job should complete by 3 into 5 15 by 2. So, the third job should complete by 3 into 5 15 by 2 which is 8. So, likewise it progresses this is what ER fairness wants.

(Refer Slide Time: 19:53)



Now, how does the scheduling actually progress? The algorithm is as follows. So, this is what we just studied. For each subtask of each task we provide a pseudo deadline like this. Now some of the pseudo some of the sub task at any given time a few subtasks let us say of each task has already been scheduled. And we have to determine which is the next sub task to execute. So, how does it do this? So, schedule sub tasks with the earliest pseudo deadlines first this is the overall idea.

How does it do? It arranges sub tasks in a min heap. And in this min heap the key is the pseudo deadline of the next sub task of this task. So, it arranges the tasks in the min heap and how what how does it arrange the key value for this min heap is the pseudo deadline for the for the next sub task of each subtask of the next sub task of each task. So, therefore, at the root of the min heap we have that task whose pseudo deadline is least or earliest.

So, extract the task at the root and execute because it is the most urgent. Now if you have m processors, you extract the m most urgent subtasks or you are you call you sequentially extract m tasks from the root with the least pseudo deadline. And then you execute it, after execution you recalculate thus the pseudo deadline of the next sub task and insert that back into the heap, ok. So, this is how execution is going to progress. Ties between multiple sub task having the same pseudo deadlines is broken using a set of tiebreaking rules. So, you can have multiple say sub tasks of tasks whose utilizations are

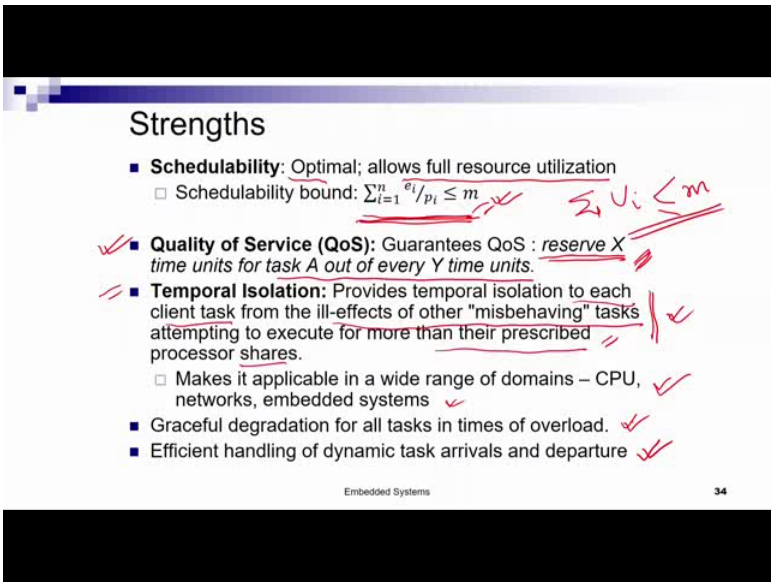
different, but multiple sub tasks can have the same next pseudo deadline for this next sub task.

So, the pseudo deadline is same for multiple say subtasks. Then what do you do? You have a set of tiebreaking rules which we will not discuss as part of this course there are there are actually 2 tiebreaking rules one is called the successor with the other is called group deadlines which will not go into in this course. For this course we would say that if such if such a clash happens of multiple subtasks having the same pseudo deadline break it arbitrarily.

Now, what is the complexity of this algorithm? The complexity of this algorithm is big O of n big O of log n per timeslot per processor. So, why do we have this? Suppose why do we have this we have to build a heap which is a onetime procedure big O of in which we neglect at the beginning. And then what happens? We take out the least we extract the task with the least pseudo deadline which is O 1 fine, but then we have to reheat if I and rearrange the heap which is a log n operation big O of log n operation.

And for m sub tasks if we have an m processor system, then the total overhead for extracting the m sub tasks at each time slot and re heap we find them is of the order of the big O of m l g n,. So, this is the complexity for this algorithm.

(Refer Slide Time: 23:42)



So, what are the strengths of this P fair algorithm? The first is that it is optimal. It allows full resource utilization, the scheduling bound being summation e i by p i is must be less than equals to m the number of processors. We saw for EDF that summation e i by p i or u i has to be less than 1; therefore, EDF we said was an optimal algorithm which allowed full resource utilization in uniprocessor systems. And in this case for the ER fair algorithm. In fact, for the p fair algorithm as well. It says that if you have a task set whose total utilization is less than m you will be able to feasibly schedule that on this multiprocessor system, ok.

So, this is what it says. So, therefore, summation over u i less than equal to m is sufficient is the sufficient condition for schedulability on multiprocessor systems using ER fair. It allows it allows guaranteed quality of service. In terms of saying that it allows you to reserve x units of time or rather e i units of time for task a out of every y time units. So, we said that such kind of execution progress guarantees or quality of service guarantees can be provided when we use this. And this property can be useful in many embedded systems which execute a lot of tasks together let us say streaming task along with other best a fair downloads together. So, if a set of tasks have a such a quality of service requirement like that for streaming tasks which must execute at a certain rate p fair algorithm is a good alternative. Although, or the derivative ER fair is a good alternative; however, there are other constraints which we will look why that which is why P ER fair is not always practically used.

Another good property of ER fair is that it allows temporal isolation of a of a from as we will see it provides temporal isolation to each task each client task, from the ill effects of other misbehaving tasks attempting to execute for more than their prescribed processor shares. So, it so, we have said that each task is guaranteed a processor share, it reserves the x units of time for task e out of every y time units.

Because it always finds the earliest pseudo deadlines among all sub tasks and execute it first. So, if a task takes more execution time than it is stipulated to take it is that task which only suffers. Other tasks whose execution urgency becomes a higher will get the processor, and it is only this task which will be delayed in its completion.

So, this is somewhat this is a very good temporal isolation and that is provided and it is often important in embedded systems, safety critical embedded systems, running critical

tasks. And this property to some extent we was also provided by the rate monotonic algorithm how it the execution progress of a task can only be affected by is higher priority task. It cannot be affected by lower priority tasks.

Here we are saying it is a much stronger it is a much stronger condition which says that it will never it does not matter high priority low priority, because the priorities are dynamic and we have priorities associated with each subtask. We can say that the task will never be affected by other misbehaving tasks which take more time to complete than it was stipulated to take.

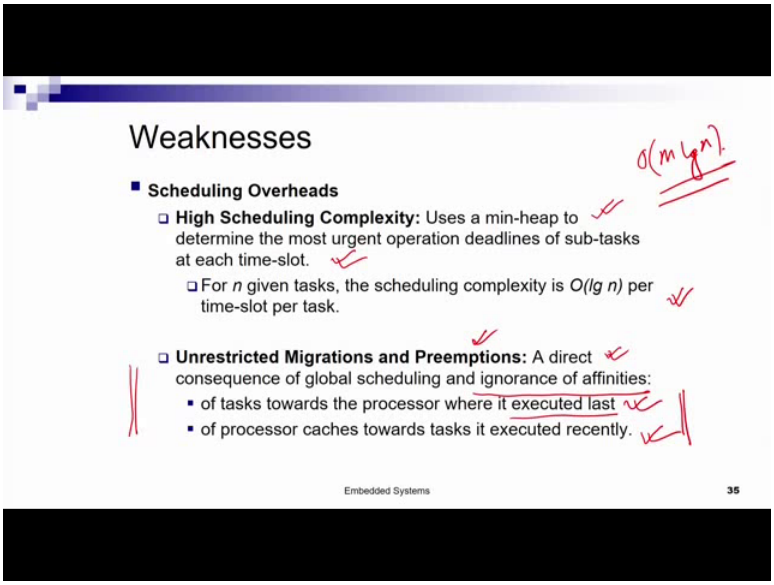
Suppose a task was allocated e i amount of time to complete, but it takes more than that to complete; that means, behavior will not affect other tasks. This is what ER fair guarantees. And this makes it applicable to a large range of domains in CPU networks embedded systems etcetera. It allows graceful degradation in times of overload meaning that, suppose this condition is not met. So, what happens is that this summation becomes greater than m. Then everybody is affected by the same degree, why? If you see the way ER fair algorithm is implemented at each time slot, it finds out the task with the highest urgency. This relative urgency is going to remain same, even if summation e i by p i becomes higher than m.

So, everybody we will be affected by the same degree. Everybody will get delayed by the same amount. And by the same proportionate degree and this is why ER fair provides graceful degradation for all tasks in times of overload. And it allows efficient handling of dynamic task arrivals and departure. So, at any point in time, handling dynamic arrival so, a new task arrives and an existing task departs this handling is a very simple for the ER fair algorithm. We just need to check if after the arrival of the new task is this condition still satisfied.

So, if I accept the new task will summation e i by p i still remain less than m, if this is satisfied the new task can be allowed into the system. If an existing task departs it just departs it does not matter because ER fair the upper bound on execution there is no bound. So, it will just happen that other tasks will execute at a higher rate in a proportionate manner, and it will consume the CPU. It will it will take the CPU that is the left idle by to some amount by the task that has departed. That space that additional space will be proportionately shared by other tasks and this will naturally happen we do

not need to do anything a special for that and hence task arrival and departures are easily handled in ER fair systems.

(Refer Slide Time: 30:03)



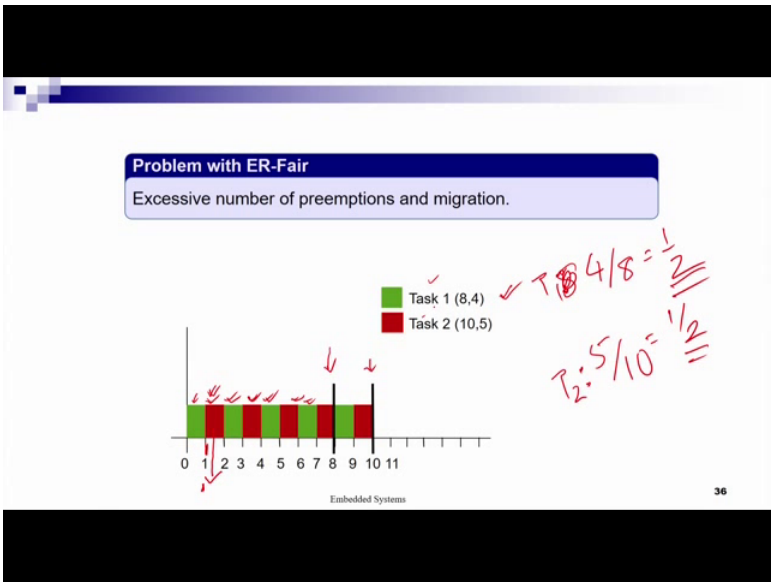
However, the ER fair is schedulers also have some significant weaknesses. So, the first is a high shading complexity. It uses a min heap to determine the most urgent operation deadlines of sub task at each time slot this is what we saw. And this incurs big O of m l g n at each time slot and this is significantly high, if the number of processors and especially the number of tasks in the system becomes very high, ok.

And this is one big problem, and thus then the and the second big problem is unrestricted migrations and preemptions. It is a direct consequence of fully global scheduling and we are saying that at the end of each time slot we decide which sub task is the best to schedule, and we schedule in that order. And therefore, it so there will be a lot of preemptions and there will be a lot of inter-processor task migrations. And due to this, there will be ignorance of affinities of tasks towards the processor where it last executed. And of processor caches towards the task it executed recently.

So, there could be large number of cache misses. And such cache misses could be costly. In even in let us say in closely coupled systems, where you do not have shared cache at a close enough level, such cache misses could be costly because that the transfer rate the, you have to transfer the code and code and data of a task from one processor to the other from the local cache of one processor to the local cache of another processor.

And this could be quite significant. And if it is the completely distributed system such migrations could be very huge the overheads could be very huge, and this algorithm can become non unusable. So, ER fair is a good algorithm for say closely coupled synchronous multicore, systems let us say with moderate number of tasks not very high. So, for cases for systems in which these overheads can be controlled ER fair is a good algorithm to use.

(Refer Slide Time: 32:28)



So, with this we highlight this migration problem of ER fair let us. So, we have 2 tasks one is 8 by 4, the other is 10 one has an utilization, one has an utilization 4 by 8 which is half again. And the other has an utilization 5 by 10 which is half again. So, you have 2 tasks T 1 this 1, and T 2 this 1, ok. These are the 2 tasks we have here and therefore, both have a utilization of half. So, if we see that that both have the first subtask of both will have a pseudo deadline of 2. And hence one is going to execute, ok.

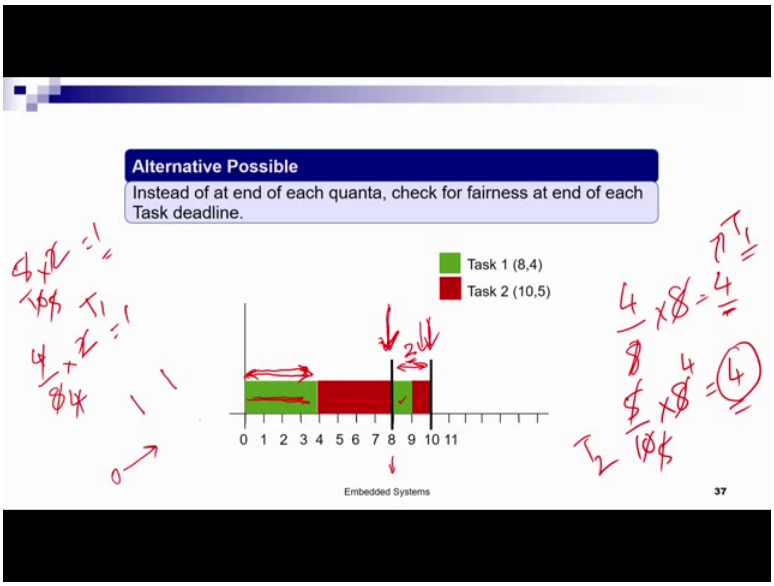
So, they will get they will take alternate turns to execute. Each of them will take alternate turns to execute, because after let us say sub task 1 has executed for task 1. The second sub task has a pseudo deadline of 4. The first sub tasks of task 2 has still not executed. So, it is pseudo deadline is 2.

So, therefore, this becomes more urgent than the second. So, for tasks to the first subtask of task to becomes more urgent than the second sub task of task 1 at this position at this time instant one at the boundary of 1. After task 1 the first subtask of task 1 has already

executed. Then at this time point one the first subtask of task 2 becomes more urgent the pseudo deadline is lower than the second sub task of task 1. So, the first subtask of task 2 executes here then again this alternative arrangement proceeds one by one. But the deadlines by which actually we need to finish tasks is 8 and 10 for these 2 jobs for the first jobs of task 1 and task 2, right.

So, the deadlines are much later, but we are doing that, but we can do better than that.

(Refer Slide Time: 34:43)



As we saw that, if we have, if you had done deadline partitioning, if you had done deadline partitioning what you could have said is that, within time 8, within time 8 which is the first deadline starting from 0. So, at from if you look from 0 the first deadline is at 8 the second deadline is at 10. The first deadline is for task 1 the second deadline is for task 2. And we can say that by 8 it is sufficient to complete 4 units of work. So, 4 by 8 into 8 which is 4 units of work for task 1. And also 4 units of work for task 2 ok so, this is 4, this is 5.

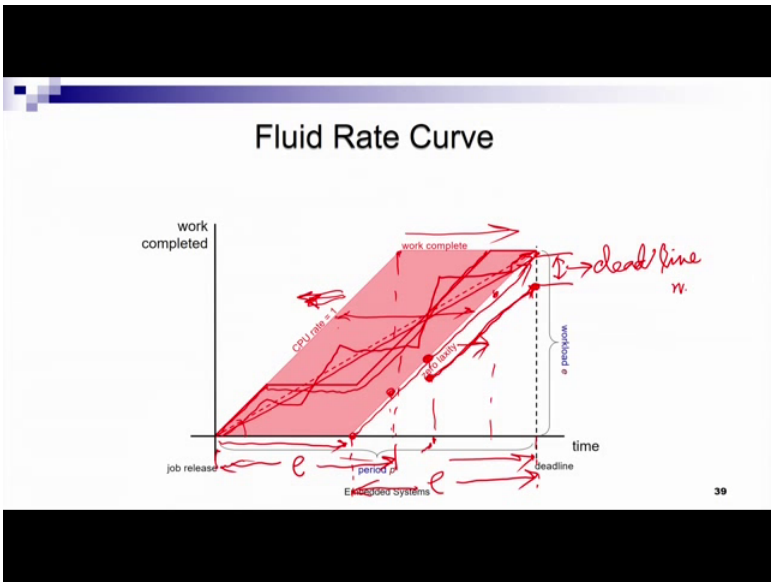
So, it is sufficient to complete 4 units of work for task 2, and 4 units of work for task 1. So, this is what has happened and you need not have incurred mi preemptions and migrations in between. You can consecutively execute 4 subtasks, and you can safely do that because we know that the first deadline that is going to come is at 8. And the proportionate progress a deadline boundaries is sufficient to meet all deadlines as we saw in the example previously.

So, it completes 4 units of execution and then task 2 completes 4 units of execution. And then between 8 and 10 I have a slice size of 2. And within this 2 I know that if I have 4 and 8 and within this 2. I therefore, must complete 1 unit of execution for task 1. And again 5 by 10 into 2 which is 5 so, 1 unit of execution for task 2 so, I must complete 1 unit of execution for task 1 and 1 unit of execution for task 2 between 8 and 10.

If I can ensure this I am I can ensure that at all deadline boundaries, at which I check at which I actually check whether I meet or miss deadlines. I will be able to guarantee all deadlines. Because all task progress fairly by there ER fair degree. I can say that the task had ER fair at 8 the task at are ER fair at 10 all tasks.

So, if they are ER fair they will also always meet deadlines. So, we are saying that because we are checking deadlines only at actual deadline boundaries, it is sufficient to violate the exact ER fair requirement within consecutive deadlines. This is what has happened here. We have violated individual subtasks pseudo deadlines, but we can still ensure the dead the progress that the sufficient progress has been made at actual deadline points, ok. So, by this we can save migrations and preemption, but ensure full resource utilization. And this is the essence of the DP fair or algorithm which we will discuss next.

(Refer Slide Time: 37:43)



So, DP fair has come within around the 2010. So, around 2010 it was first published suppose in real time system symposium and then subsequent journal version of this work has come in 2011. So, it also basically is based on the fluid schedule. So, this is the fluid

scheduling rate which we just saw. And we said that this one is a CPU rate of 1. So, this angle is 45 degree is meaning that if you allocate so, this position this duration is essentially e, this workload e.

So, what happens is that, if you allocate the processor continuously. If you allocate the processor continuously is from release, it will complete at time e if it has and execution requirement of e. And you allocate the processor continuously only to this task. Then it will complete the work at e. And therefore, this one we will have a slope of 45 degrees. So, slope of one sorry this angle is 45 degree and this is a slope of 1.

So, it will exactly complete it is work at time e and this part it will remain idle here. Now how long can it wait? It can wait it is saying this is the 0 laxity curve this is again 1 it that which means that this 1, this is again e. So, this is that this is the point of 0 laxity; that means, you have e amounts of work remaining, you did not allocate the processor by this whole amount or by this whole time.

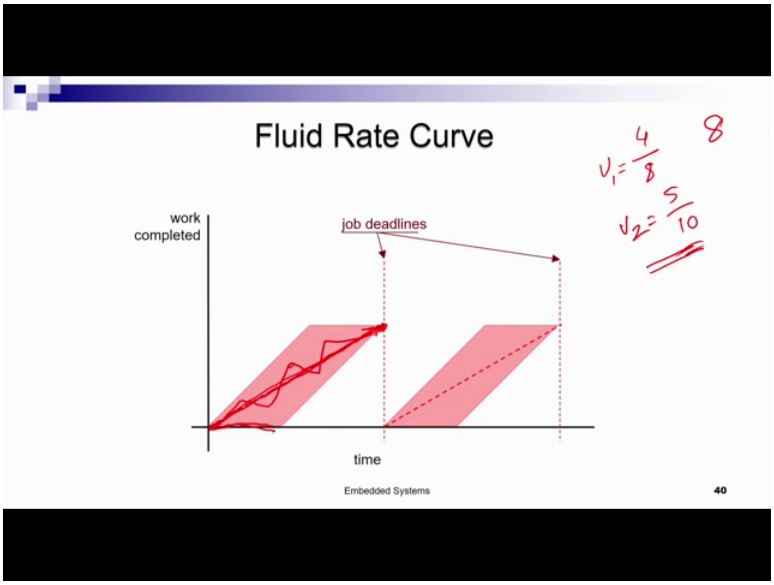
And then at this amounting you have exactly u e e amount of work remaining to be completed within this term with within this time of e, within this time e you must complete the e amounts of work. And then after this time from this time to complete your work by the deadline, you must continuously allocate the processor and complete your work here, this is what it says.

Now, in between the actual schedule can proceed. So, here EDF any algorithm if you look at it will progress something if the schedule is valid and does not violate deadline, then what will happen? The actual execution can proceed in any curve like this; however, it has to execute something in between. It can be anything else as well, it can be anything else as well, but it has to complete it has to remain within this region this red region, this red region it has to remain.

If it violates this region, it cannot go on this side because this means that you are executing at a rate more than 1 which is not possible, because we know that at most this much amount of work can be done if the whole processor is allocated to the task for this entire time. So, it cannot progress on this side, it cannot progress on this side because deadlines will be missed.

So, it must complete at least this much of execution at this time, it must complete at this much of execution by this time from the beginning. It must complete at least this much of execution from this time, it must complete at least this much of execution by it is deadlines to ensure that deadline will be met. Suppose it does not it is somewhere here and it does not complete this much amount of execution at this point from the beginning. It means that even if you allocate the full processor to this, it cannot complete it is full execution it will only complete about this much amount. So, this much amount of execution this much amount of execution will still be left to be completed and hence it will miss deadline; so, this is what this figure says.

(Refer Slide Time: 41:59)



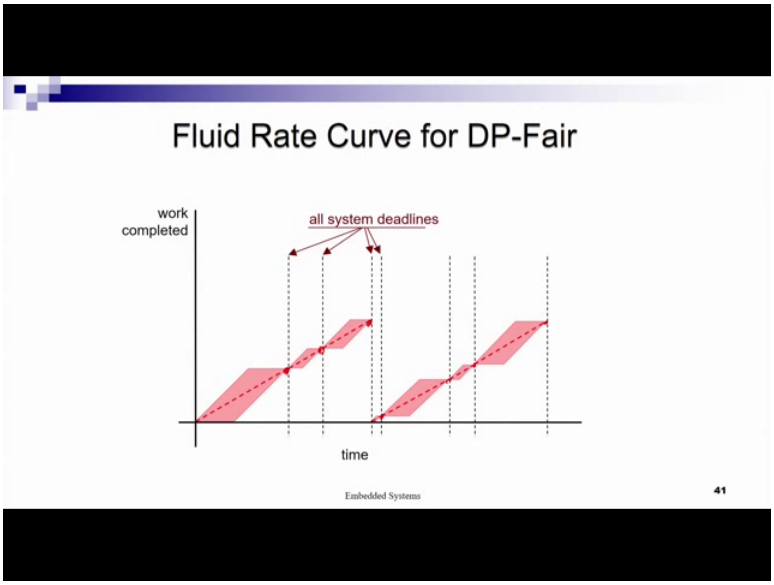
Now, our deadline fairness essentially deadline partitioning fairness said that, you should complete your fair amount of execution progress by deadlines, ok. At we said that for that example in which we had these 2 tasks of 4 by 8, you 4 by 8, you one 4 by 8 and you to 5 by 10; for these 2 tasks we found that by 8 we should complete 4 units of execution both for task 1 and task 2 by because 4 units of execution was the feared execution progress at deadline boundaries. So, this one says it this one says that this is the amount of work that you can you must do by this deadline for a given task and within this.

What DP fairness says that within this it does not check fair execution progress. So, you need not you need not absolutely follow this follow this ideal execution rate. You need not exactly follow this ideal execution rate within this time. You are allowed to be within

this at some point in time. At any point in time you are allowed to be anywhere within this, but the end of this time what DP fairness wants is by the end of this time you must complete this much amount of work.

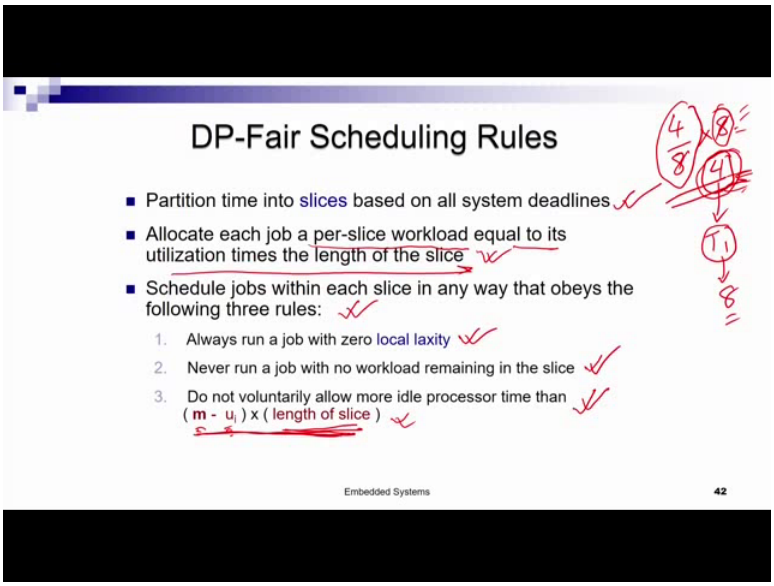
And again in the next deadline by the next deadline you should complete this much amount of this much amount of work. This is what it says.

(Refer Slide Time: 43:36)



So, if the system has various deadlines as we saw 8 10 and then we will again have a deadline at 16, 20, likewise if you go on having the system has numerous deadlines, the DP fairness prescribes that for each task, for each task you should come you should have completed your fair amount of execution at each deadline boundary. And that should be guaranteed for each task. So, this is the amount of work that should be completed for task 1. For the first job of task 1 and then for the second job of task 1 again. It should progress in a it should come in a in a deadline partitioning fair manner DP fair manner at each deadline boundary. This is what it says, ok.

(Refer Slide Time: 44:24)



So, what are the rules for DP fair scheduling? You partition time into slices based on system deadlines. So, what are the system deadlines? These are the period boundaries of all subtasks. So, you have multiple job boundaries or rather the job boundaries the job deadline or period task period boundaries of all tasks over time. And then at any point in time it finds the next earliest deadline boundary, it allocates shares for example, we said that in the in the first time slice 0 to 8 it must complete 4 by 8 into 8 which is 4. So, this 4 is the share execution share of that task T 1 for in the time slice of 8 duration 8, ok.

So, it allocate each job a per slice over workload equal to it is utilization times the length of the slice. So, this was is utilization, this was the length of the slice. And therefore, it allocates each job of each task a personalized workload equal to its utilization times the length of the slice and then scheduled jobs within each slice in a way that obeys the following rules. So, we just saw the rules and we just track them down again, always run a job with 0 laxity. So, you cannot delay a job with 0 laxity within that you can you can you can be anywhere you did not you need not. For example, obey by the exact ER fair say execution progress guideline. You can violate, and you never run a job with no workload remaining in the slice.

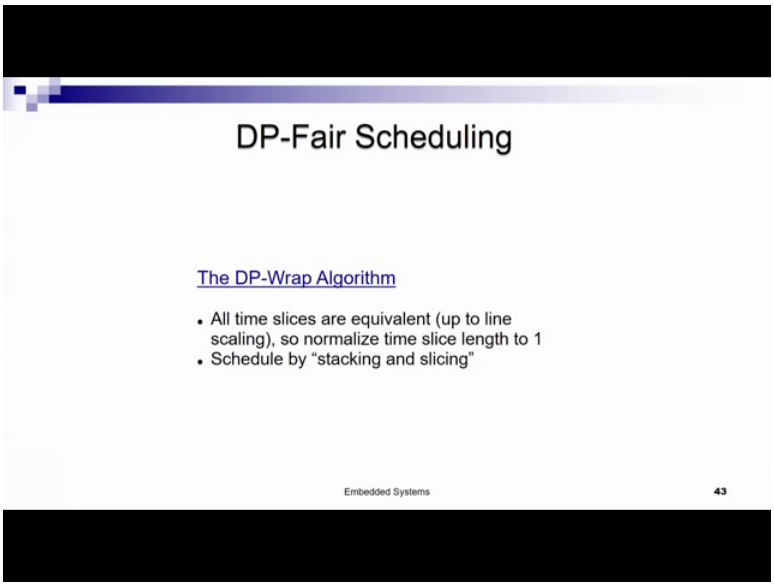
So, then you are you are saying that if you do not execute more than your workload. So, it means that, this version of the DP fair algorithm can allow a bit of non-work conservation. Meaning that, if your amount of work as completed in a particular time

slice, and you do not have any other tasks to execute you still do not execute more than what you are prescribed to execute in the total time slice. So more than 4 times time slides that time units of execution you do not execute within this time 8. So, this is what DP fair suggest so, that no deadlines will be missed, but we can have mechanisms to bypass this as well, ok. And do not voluntarily allow more idle time than m minus u i into length of the slice.

So, for any task the u i is its utilization, now what does this particular expression tell me; if you see m into length of the slice, m into length of the slice is the total number of time units available within the time. Size m number of processors and on each processor you can execute for the length of the time slice. So, let us say 8 is the length of the time slice, and you have 4 processors, then 8 4's are 32 total time units of execution is possible in this time slice, of 8 time units. And what is u i; u i is the amount of work that that is that is needed for a given task ok.

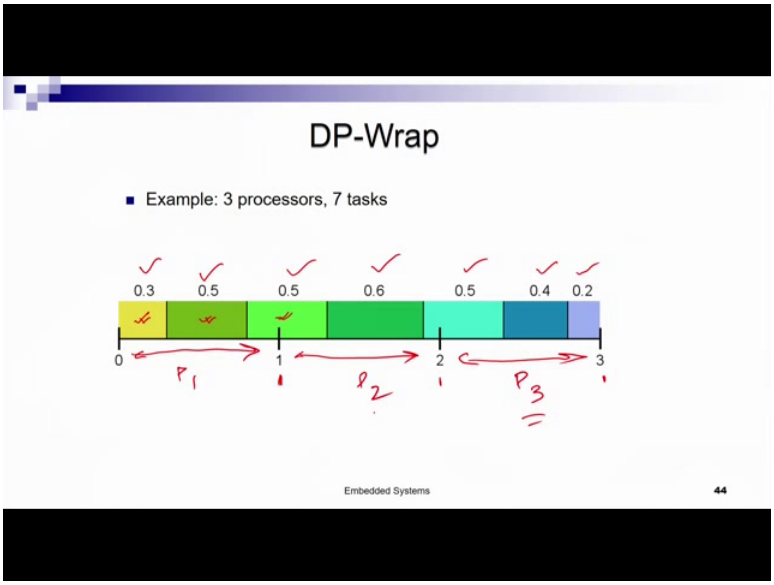
So, u i is the amount of work that is needed to be completed by a given task. Within let us say this 8 time units I need to complete force. So, u i into length of the time slice is 4 for say task 1 here for this case ok. So, this one says that you cannot do you cannot voluntarily allow more idle processor time than this. Because if you allow more than that then you cannot complete the set of jobs that you in the set of total set to total workload that you have within the time slice.

(Refer Slide Time: 48:48)



Now, we provide a strategy for execution within each time slice. How does execution within each time slice progress. And that happens using the DP wrap algorithm. Now all time slices are equivalent up to line scaling. So, normalized time slice by 1 what does this say? It says that for all time slices you normalize any time slice into a time slice of length 1, ok. And then you do stacking and slicing as follows, we see the procedure here.

(Refer Slide Time: 49:28)



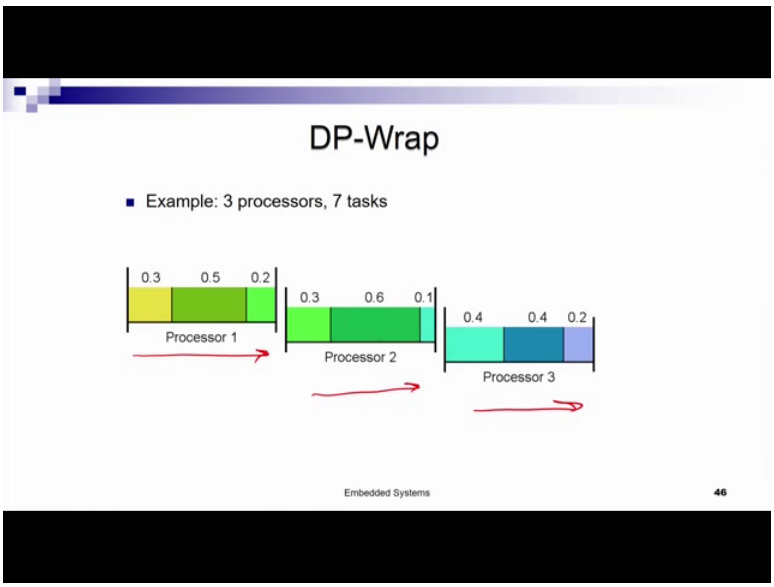
Suppose, you have a set of 3 processors and you have 7 tasks. And these 7 tasks are have utilization 0.3, 0.5, 0.5, 0.6, 0.5, 0.4, 0.2. So, these are the 7 tasks. And we just tap the 7th as one after another onto this line. And we divide this whole set, also this tasks are stacked one after another onto this line, and we divide this line into at the at the junction at the integer points 1, 2, 3, ok.

We said that the; what is the idea? The idea is that we said that the processors are each of unit capacity. So, this is the total workload that can be handled by 1 processor. So, this can be allocated to say processor 1 this can be allocated to processor 2, this can be allocated to processor 3, and we can execute them.

(Refer Slide Time: 50:40)

So, therefore, what we are doing? We are allocating this amount so, 0.3 plus 0.5 so, this 0 point 0.3 plus 0.5 0.8. So, we allocate 0.2 here, and we allocate 0.3 here. We allocate 0.3 plus 0.6 0.9. So, we I allocate 0.1 here and 0.4 of this total 0.5 here. And we complete by 3. So, if you see this total workload, if you sum up the utilization some u i equals to 3 in this case. So, it is schedulable through because this algorithm DP fair is optimal, ok.

(Refer Slide Time: 51:23)

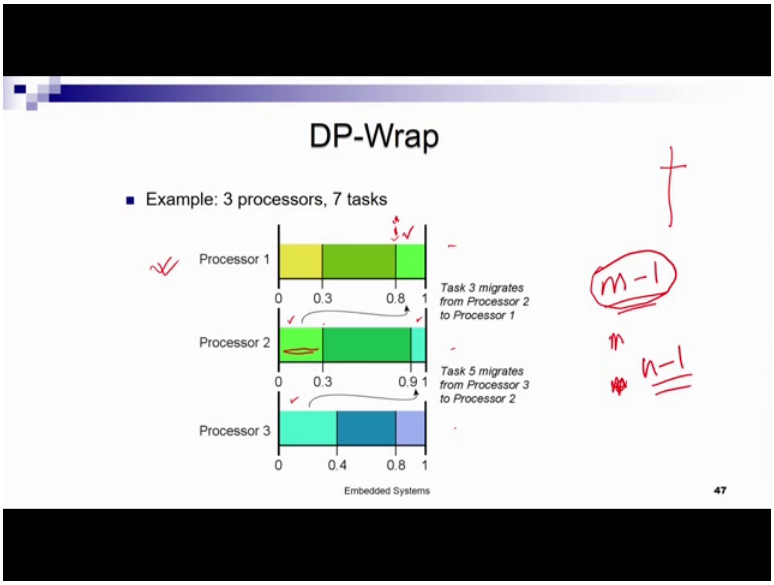


So, then we slice we had stacked we divided into processors and we sliced. So, this is the workload for processor one this is the workload for processor 2 and this is the workload

for processor 3. And then for this is for any we said that this is for a unit length time slice.

Now, if your time slice is 8, you just multiply the amount of work that can that must be done for task 1. So, 0.3 into 8 is the amount of work for task 1, 0.5 into 8 is the amount for work for task 2. 0.2 into 8 is the amount of work for task 3 on processor 1, 0.3 into 8 is the amount of work for task 3 on processor 2. 0.6 into 8 is the amount of work for task 4 on processor 2, likewise it proceeds.

(Refer Slide Time: 52:18)



So, these tasks which are partially execute on one processor and partially on another processor must incur a migration. And these tasks cannot overlap over time, because this is a single piece of code and it means that the first part of the work will be done on processor 2 and it will complete by this amount of time. And at this time within the time slice then task 3 you will be migrated from processor 2 to processor 1, and start execution at this time after it completes on processor 2.

And we can guarantee that it will never overlap if the total work we can have an allocation, like this like through this slicing through this tacking and slicing which and if the processors are homogeneous we can guarantee that if the individual utilized of the task is less than 1, then we can have an allocation where a task will never overlap on 2 processors.

So, we can have like if the task will complete on processor 2. And then it will be migrated to processor 1 and a complete execution. Similarly task 5 will first execute on processor 3. And then after it completes on processor 2, it will be migrated to processor 2. It will execute on processor 3 and then after it completes on processor 3 it will be migrated to processor 2, where it will complete it is remaining execution by the end of it is time slice, ok. This is how DP wrap algorithm within DP fair progresses.

So, we discussed 2 important algorithms, the e r the 3 important algorithms, rather the P fair algorithm the ER fair algorithm and the DP fair algorithm which uses DP wrapped within it to conduct scheduling, we said that P fair is the first optimal multiprocessor scheduling algorithm, but P fair and ER fair are the first optimal multiprocessor scheduling algorithms. But they put severe constraints on the rate of execution of the tasks which is not necessarily just to meet deadlines.

And in doing that the algorithms become costly in terms of scheduling overheads, and also in terms of migration and preemption overheads DP wrap is a more recent algorithm which allows much better much which fair much better in terms of its schedulings as well as migration and preemption overheads.

In fact, DP fair can guarantee that within every time slice there will be at most m minus 1 task migrations where m is the number of processors. So, DP fair can guarantee that the number of migrations within each task sorry within each time slice will be at most m minus 1. For example, if we see in this case we see that we have 3 processors and the number of migrations are 2; one from processor 2 to processor one for task 3 and the other from processor 3 to processor 2 for task 1. It also guarantees that the number of preemptions are also limited. So, number of preemptions is upper bounded by n minus 1.

So, where n is the number of tasks so, within each time slice. So, DP fair guarantees that the number of preemptions. So will be at most n minus 1 and the number of migrations will be at most m minus 1 within every time slice. And hence it is overheads are much lower than p fair and ER fair, with this we come to the end of this lecture.