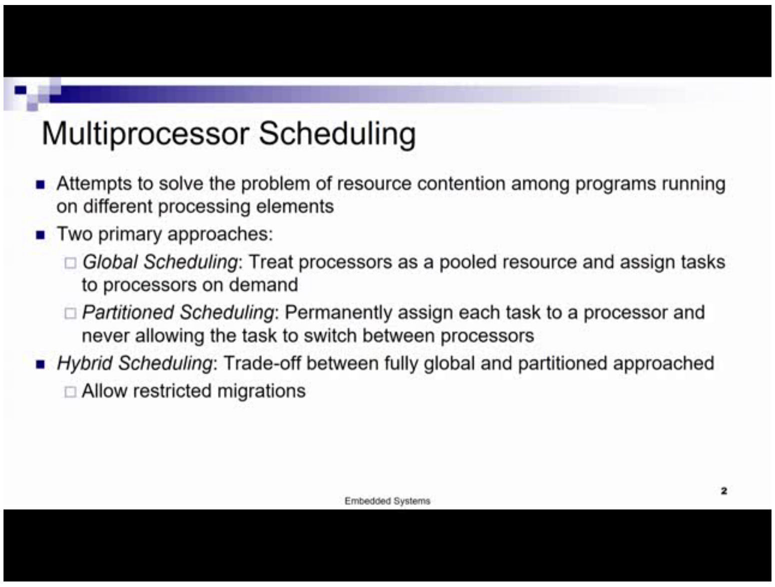
Embedded Systems – Design Verification and Test

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

Lecture – 17
Real-Time Multiprocessor Scheduling

Hello. In the last lecture, we looked at real time uniprocessor scheduling strategies used in embedded systems. In this lecture, we will look at Real-Time Multiprocessor Scheduling approaches.

(Refer Slide Time: 00:43)



Multiprocessor Scheduling

- Attempts to solve the problem of resource contention among programs running on different processing elements
- Two primary approaches:
  - *Global Scheduling*: Treat processors as a pooled resource and assign tasks to processors on demand
  - *Partitioned Scheduling*: Permanently assign each task to a processor and never allowing the task to switch between processors
- *Hybrid Scheduling*: Trade-off between fully global and partitioned approached
  - Allow restricted migrations

Embedded Systems 2

So, multiprocessor scheduling attempts to solve the problem of resource contention among programs running on different processing elements. So, we used uniprocessor scheduling to solve the problem of contention among tasks running on the same processor.

Now, if you have multiple processors and there are tasks that must be executed on a set of multiple more than one processor, then you have to use multiprocessor scheduling strategies. So, therefore, it attempts to solve the problem of resource contention among programs or tasks running on a set of more than one processing elements. And there are 2 primary approaches towards multiprocessor scheduling. The first is global scheduling which treats processor as a pooled resource. So, we do not consider separate processors.

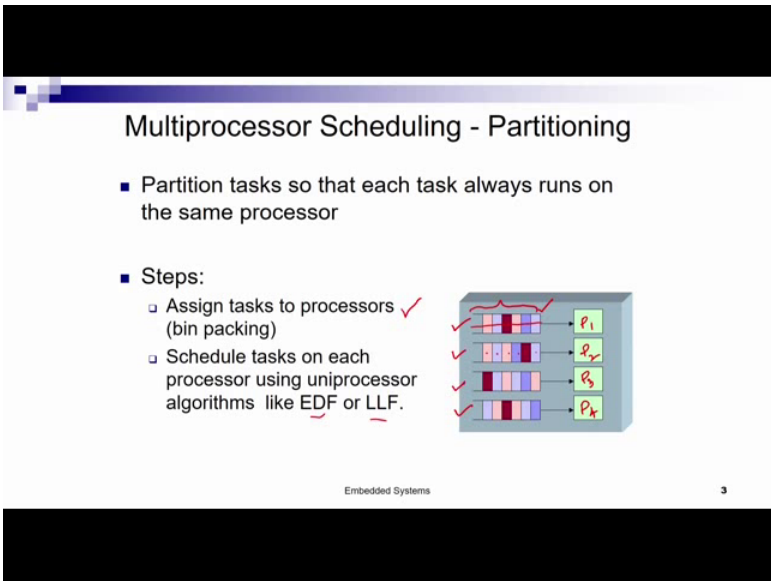
We consider all processors together as a single pooled unified resource and assign tasks to processors on demand. So, the same task can be assigned to different processors as and when required based on demand, with migrations being required when we have to transfer a task from one processor to another but we will see the all the processors together as a single resource against partition scheduling which is the other type; where we permanently assign each task to a single processor and which are never allowed to switch between processors. So, the tasks are not allowed to switch between processors, ok.

So, once there is an allocation phase in partition scheduling; there is an allocation phase where you allocate each task to this to a distinct processor, and you run that task on that processor and never migrate it for the entire duration of it is lifetime. Between these 2 approaches you also have something called hybrid scheduling; which is a tradeoff between fully global and partitioned approaches, this allows restricted migrations.

Meaning that for example, you can have each job of a task to execute on a single processor, but different jobs can execute on different processors. For example, let us say you have a task consisting of a sequence of jobs j 1, j 2, j 3, j 4 one can run in processor 1, j 2 can run in processor 2 j 3 can again run in processor 1, j 4 can run in processor 3, likewise.

So, we have allowed restricted migration. So, you are here you are allowing job level migrations, and therefore, it is not part fully global scheduling. And it is not fully partition scheduling. It is not fully partition scheduling, because you are allowing migrations, but it is not fully global scheduling, because you are restricting migration within a job. So, the job could be arbitrarily long and for the entire lifetime of the job over the entire execution time of the job it has to execute on a single processor. So, therefore, it is not absolutely global scheduling not absolutely partition scheduling something in between, hence it is called hybrid scheduling.

(Refer Slide Time: 04:04)



Multiprocessor Scheduling - Partitioning

- Partition tasks so that each task always runs on the same processor
- Steps:
  - Assign tasks to processors (bin packing)
  - Schedule tasks on each processor using uniprocessor algorithms like EDF or LLF.

Now, we look at partition scheduling in a bit more detail. So, partition scheduling it partitions tasks so that each task always runs on the same processor as we said. And so, what we do is, we assign tasks to processors first using a strategy called using a strategy called bin packing. So, here bin means processors so, each processor is a separate bin of a certain capacity, and if you have homogeneous processors, you can always assume that each processor is of unit capacity 1. And you assign tasks which are individually of size equal to it is utilization.

So, we are saying that let us say you have a task of period 4 and to be executed for and with an execution time of 2. It is period is 4 and it is execution time is 2. So, the utilization of the task E by P is 2 by 4 which is half. So, we know that if the capacity of the processor is 1, I am assigning a task of size half on to this processor. So, here this processor will be called bin, and we have to pack tasks on to each of these bins and hence it is called bin packing.

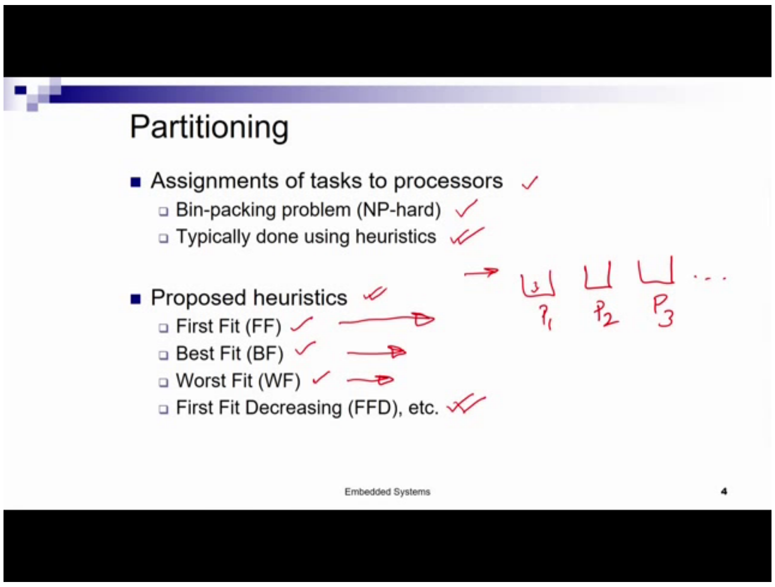
Now, we have separate run queues. So, this one is processor 1, this one is processor 2, processor 3 and processor 4. Your 4 processors and you have 4 separate run queues. For each of them 4 separate ran queues. And you have assigned the tasks to be allocated to each run queue on each processor. So, these are the tasks which needs to be which are assigned to processor 1.

Similarly, these are the set of tasks each one of them being a separate task being assigned to processor 2, likewise. So, what is the advantage of partitioning? The one big advantage of partitioning is that once you have allocated the task 2 separate processors, then you can use well known simple uniprocessor scheduling algorithms like EDF, RMA or LLF. LLF is mixed laxity first which we have not studied, but it is a derivative of EDF where you allow where you allow priorities of the jobs within the jobs to change.

So, the jobs again will be divided into subtasks; where each sub task can be executed a within the time slots assigned by the operating system. And different sub tasks within a job can have different priorities. That is LLF we did not discuss it; however, it is also an algorithm that is used in uniprocessor systems.

So, well known uniprocessor algorithms like RMA, EDF, LLF can be used within each processor, once you divide the task into the individual processors. This is why partitioning this partitioning approach is many a times used. And also the overheads are overheads are also less because once you assign the task, you do not need to migrate tasks from one processor to another. Migration is a high overhead process as we will see in time in a in a few slides from now.

(Refer Slide Time: 07:29)



Partitioning

- Assignments of tasks to processors ✓
  - Bin-packing problem (NP-hard) ✓
  - Typically done using heuristics ✓✓
- Proposed heuristics ✓✓
  - First Fit (FF) ✓
  - Best Fit (BF) ✓
  - Worst Fit (WF) ✓
  - First Fit Decreasing (FFD), etc. ✓✓

Embedded Systems 4

Now, partitioning as we told is the assignment of tasks to processors, but this bin packing strategy, the optimal strategy for bin packing is an NP hard problem. So, it takes exponential amount of time with respect to the number of tasks. And therefore, when the number of tasks and processors in the system increase the partitioning process becomes very complex and time consuming. So, therefore, people typically use heuristics to solve the bin packing problem.

And there are a few very well-known heuristics. Like first fit, best fit, worst fit and first fit decreasing. In the first fit what do you do is, suppose you are given a set of processors, you put the tasks in the first processor, and you name the processors say P 1, P 2, P 3 likewise. And you start assigning tasks. So, you have a set of tasks you start assigning tasks from P 1, P 2, P 3 and you and what do you do is, you take the next task and put it in the first processor which has enough capacity to hold it, hence first fit.

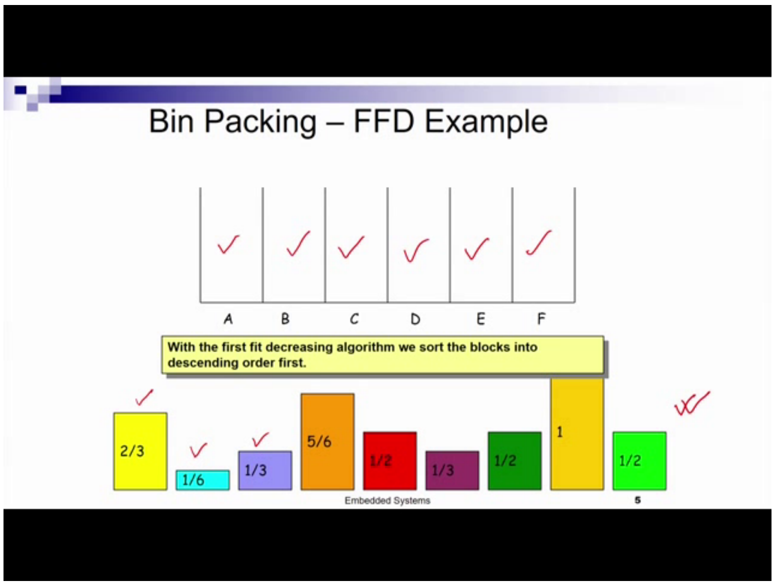
So, you start allocation from an assigned processor P 1 then P 2 then P 3, you maintain this order for consideration of the task for allocation you main always maintain the same order. And where do you put the tasks onto the first processor when searching in this order, that has enough capacity to hold the next task.

Best fit on the other hand; what does it do? It takes the processors and finds out that processor among the set of all processors which has the least capacity, but can hold this job which has the least remaining capacity, but that capacity is sufficient to hold the next job. So, it is the best fit it is called best fit, because it does not allocate to the first processor that can hold it.

The first processor can have a large capacity remaining or may not have any capacity remaining, but what does best fit do? It tries to find that processor which just has sufficient capacity to hold the task. Any other processor which has more it this processor this best fit processor that processor will have a higher priority for this job, than any other processor which has higher remaining capacity, and no other processor with lower remaining capacity can hold this job, right.

So, this is what is best fit; worst fit on the other hand what does it do? It places the next task onto the processor which has the highest remaining capacity so, hence worst fit. All these heuristics are used. First fit decreasing; in first fit decreasing what do we do? We take the task and first sort them from the highest size to the lowest size. And then we apply first fit, hence first fit decreasing. Now we will look at first fit decreasing in a bit more detail with an example. Let us now understand the first fit decreasing algorithm with an example.

(Refer Slide Time: 10:53)



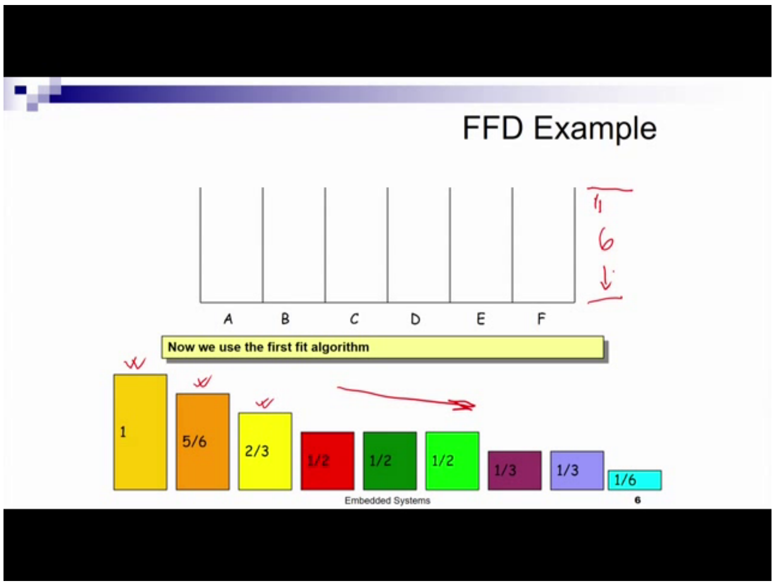
Bin Packing – FFD Example

With the first fit decreasing algorithm we sort the blocks into descending order first.

Let us assume that we have a set of these 9 tasks. The first task having an utilization 2 by 3 second task having a neutralization 1 by 6, third task having an utilization one third and so on, to be scheduled on a set of these 6 processors. These are all unit capacity 6-unit capacity processors, and we now must partition the task set first before scheduling because we are doing a partition scheduling, and we are using bin packing for the partitioning process. So now, we have to partition these 9 tasks all to this 6 processors, ok using first feed decreasing.

Now, first with decreasing says that we need to first sort the tasks based on their utilizations.

(Refer Slide Time: 11:41)

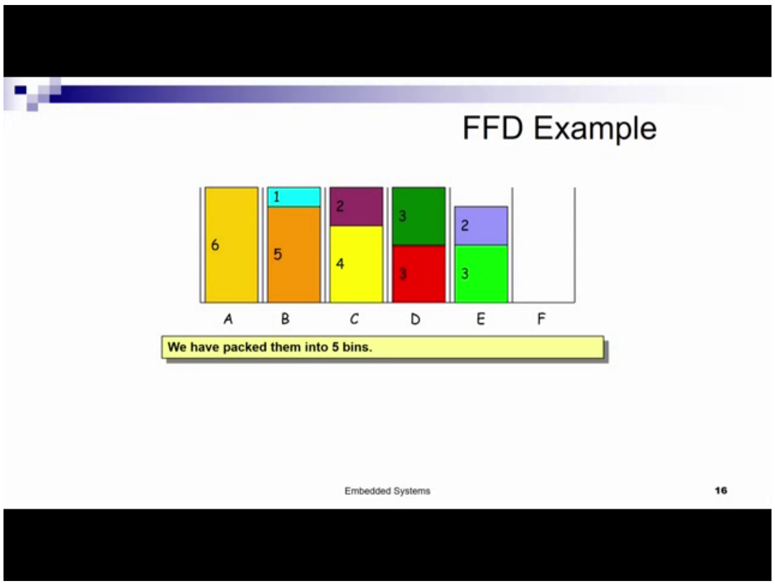


FFD Example

So, after sorting, we get this sorter sequence of tasks from the largest task to the smallest task. And after sorting we need to apply the first fit algorithm. Thus, we take the first task and put on the first processor that can hold it. But before going to do that, we will just for convenience change these tasks this fraction and utilizations into integers.

We will scale these tasks so that each of this becomes an integer which essentially means that what we are going to do is that, we will multiply each of these utilizations by 6. And then the capacity of the processors instead of one will now become 6 after we scaled these tasks. So, after scaling what happens? The weights we have now scaled and therefore, the task 1, the weight which was one has become 6, the weight which was 5 by 6 have become 5, the weight which was two-third has become 4 and likewise. So, we have changed all the task weights to integer values. And therefore, the capacity of the processor have now become 6, each processor is now 6.

(Refer Slide Time: 13:00)



FFD Example

We have packed them into 5 bins.

So now we will apply first fit on this. So, therefore, the capacity of the first processor is 6 and the task size is also 6. So, the first processor has been completely filled. So now, we have the second task we try it on the first processor, but we do not have enough capacity on the first processor. So, we go on to the next processor. And therefore, the next processor has enough space and we put the task the next task on to processor B.

We take the third task we try the first processor, we try the second processor, cannot hold it while the second processor also cannot hold it, because the processor has a capacity of 6, and 5 plus 4 equals to 9. So, the second processor does not have enough capacity to hold both the task and therefore, it will be allocated to the third processor. Similarly, the third one we will try to go on try to put it on to the processors in the order one by one and then finally, put it in it is final place, ok.

So, we see that the processor D can hold both these 3 together. Why because, the capacity of this processor D is 6 all the processors have capacity 6 and the summation of the weights of these 2 tasks is 3 plus 3 which is 6. And hence we can put both the tasks onto the same processor. The next 3 cannot be put here and hence it will go to E. The next two ok, this can hold it, why because C is sufficient to hold this 2, because previously 4 was previously the task with wait for was in C, and it still has 2 of spare capacity and that within that spare capacity we can put the next task whose size is 2.

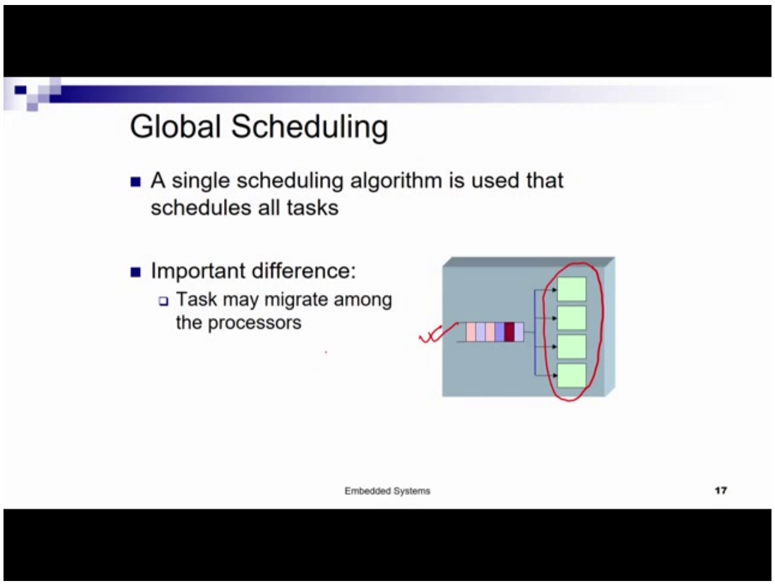
And therefore, this for the next task we can again we will see that it cannot be put in any of the processors and it has to go to E. Similarly, lastly for the last task it must be put on to the second processor.

Now, we have packed all the tasks into 5 bins. So, this is first fit decreasing similarly, we can have best fit decreasing in which again we will first sort the tasks. And then put the next task on to the processor which has the least remaining capacity, but that capacity is sufficient to hold the next task, ok.

So, the in best fit decreasing, we will first sort the task and put this put that in we will consider the task in order of from the largest to the smallest size, and we will put the next task into the processor that has just the sufficient capacity to hold the task. That will be best fit decreasing. In worst fit decreasing we will always we will again sort the task and take the high large from largest to smallest, and take the next task and put the next task into the processor which has the highest remaining capacity. That is worst fit decreasing, ok.

So, all these types of packing strategies are used depending on the situations. Now once such a packing is done we have packed. So, we have packed into 5 processors, and the one processor still is there which is spare and if further tasks come in we may need to put it into that processor, but currently this processor will be idle or will be kept off. And then now on each of these processors we will apply uniprocessor scheduling strategies like EDF, RMA to schedule the tasks on to; on in each of these processors.

(Refer Slide Time: 16:43)



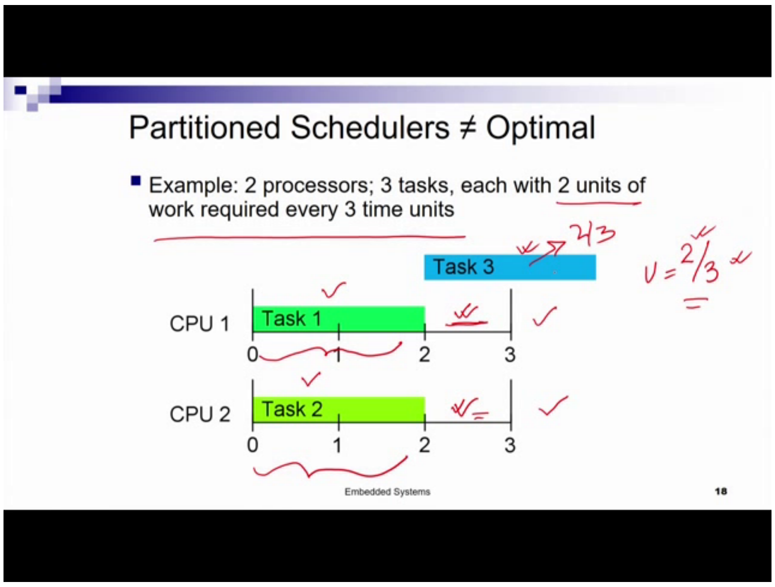
Global Scheduling

- A single scheduling algorithm is used that schedules all tasks
- Important difference:
  - Task may migrate among the processors

Embedded Systems
17

Now, we come to global scheduling. In global scheduling as we said we consider the processors as a single resource pool. So, therefore, we have a single queue. So, this is the single resource pool consisting of all the processors. And instead of separate run queues we now have one single run queue. And then there has to be a dispatcher which will find out to which processor each task will be will each task or job of the task, now the next sub task of the task given will be put, ok. So, the important difference here is that the tasks can migrate among processors.

(Refer Slide Time: 17:22)



Partitioned Schedulers ≠ Optimal

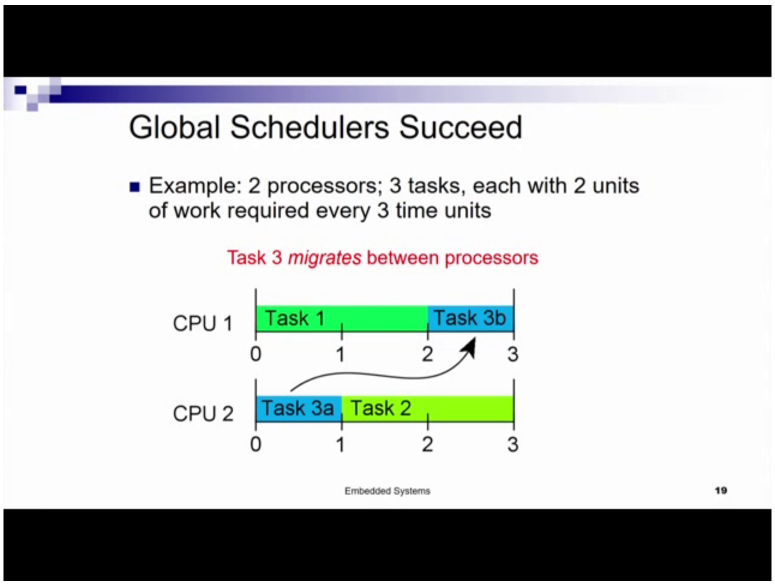
- Example: 2 processors; 3 tasks, each with 2 units of work required every 3 time units

Now, partition scheduling although is sometimes used, because we can use well known uniprocessor scheduling algorithms and that migration overheads are also low. But partition scheduling are often produce very poor resource utilizations. For example, we will take this case where we have 2 processors CPU 1 and CPU 2. And we have 3 tasks, each of who each of whose utilization is 2 by 3, ok.

So, you 2 units of work are to be done every 3 time units. So, it is period is 3, and it is a execution requirement within this period 3 is 2. So, it is utilization u is equal to 2 by 3. So, we have 3 tasks like this, we will when we are partitioning, we put the first task on to CPU 1, we put the second task on to CPU 2, but we cannot take the, but we cannot put the third task into any of these processors.

Because we only have this task has a weight of 2 by 3, and here you have 1 by 3 remaining. And here also you have 1 by 3 remaining. 2 by 3 has been used by task to 1 CPU 2, 2 by 3 has been used by task 1 on CPU 1. So, you have 1 by 3 spare capacity on processor 1, 1 by 3 spare capacity of on processor 2. So, the total remaining processor capacity though is 2 by 3. You cannot use this spare capacity to hold task 3 because none of the individual processors have enough capacity to hold this task. So, how what can you do?

(Refer Slide Time: 19:10)



Global Schedulers Succeed

- Example: 2 processors; 3 tasks, each with 2 units of work required every 3 time units

Task 3 *migrates* between processors

Embedded Systems

19

You can just break this task into 2 sub tasks, into 2 chunks, into 2 separate chunks of size 1 by 3 each.

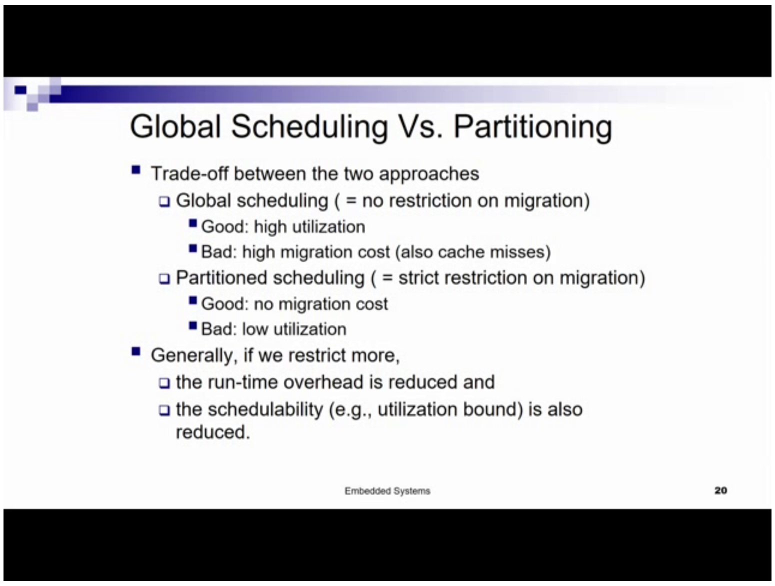
So, the first chunk is task 3 by so, the first chunk is 3 a which will be scheduled between 0 and 1 here. And the second chunk is 3 b which will be scheduled here. So, why have we done this? So, that if you do not do this, we have to if we schedule task, we cannot schedule task 3 b and task 3 a here right. We cannot schedule 3 a here. This is not possible. Why? Because other otherwise these are 2 separate, means otherwise this is the sequential process right. So, the first part has to end and only can the second part start the 2 parts cannot execute in parallel.

I have a single program code let us say. I have divided into 2 parts, but I cannot execute this until I finish this. So, I these 2 executions cannot be overlapped, these 2 separate executions cannot be overlapped. Hence task 3 and task 3 b cannot be executed at the same time on 2 processors. And hence we have put task 3 a at the beginning, and task 3 b at the end here.

However, this scheme allows us to schedule to appropriately schedule the third task partition scheme could not do it, now in a global scheme by allowing a migration, we can allocate this 3 tasks we divide the tasks into 2 parts, we execute the first part in CPU 2, then you then we migrate the task from CPU 2 to CPU 1.

So, the code if required along with the intermediate data of the task is migrated from CPU 2 to CPU 1, and then the next part of this task is executed on CPU 1, after it is completion on CPU 2. So, by incurring a migration and by dividing the task themselves, we have we will be able to successfully execute this task. And hence global schedulers succeed on multi processors.

(Refer Slide Time: 21:23)



Global Scheduling Vs. Partitioning

- Trade-off between the two approaches
  - Global scheduling (= no restriction on migration)
    - Good: high utilization
    - Bad: high migration cost (also cache misses)
  - Partitioned scheduling (= strict restriction on migration)
    - Good: no migration cost
    - Bad: low utilization
- Generally, if we restrict more,
  - the run-time overhead is reduced and
  - the schedulability (e.g., utilization bound) is also reduced.

Embedded Systems 20

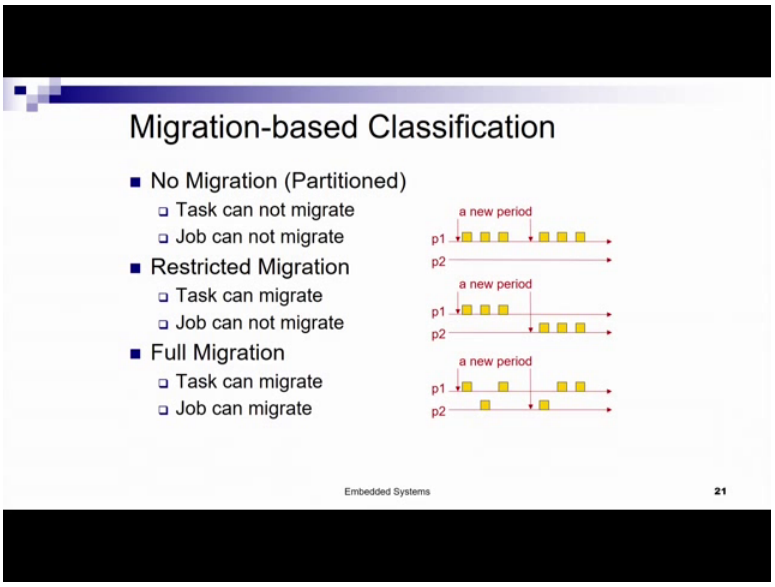
So, we if just see the pros and cons of global scheduling versus partition scheduling, we see that global scheduling has no restriction on migrations, and it allows good resource utilization as we saw. And in the last case, in the last slide we saw that it allows 100 percent utilization of the processors. It is bad in terms of high migration cost, and also cache misses. Why because, once you execute a task when we once you allow migrations, what happens is that the cache corresponding to what was executing before it is flushed.

So, you when you bring in suppose you break a task and you bring in you break a task and into 2 parts, and you take the first part execute on some processor, and you take and then you migrate it to a second processor and you execute over there. So, what happens is that in the second processor the cash is cold with respect to this task. The data corresponding to this task does not reside in the second t a second processors cache. And therefore, initially there will be a lot of cache misses. And in general when migrations are allowed, such cold misses will occur again and again. And therefore, this will incur a high cost, high migration cost.

Partition scheduling on the other hand are strict. In terms of not allowing migrations and therefore, it is good because it has no migration cost and therefore, it is high in terms of cache efficiency, cache misses are low but it is bad in terms of low resource utilization. As we saw in the last case, to one-third of the processor on off each of the processor could not be used. So, the processor utilization for that particular case was 66 percent. Why? Because the total was 2 and we could only execute 2 tasks of way 2 by 3. The third task of way 2 by 3 could not be executed. So, one-third of the total system capacity was wasted. And hence the utilization was around 66.6 percent.

In general, if we restrict more runtime overhead is reduced. So, if we are more and more partition, if we go towards partitioning, runtime overheads are reduced. And however, the schedule ability or the resource utilization bound is also reduced. So, based on this classification on migration, we can have different classes of scheduling algorithms.

(Refer Slide Time: 24:09)



Migration-based Classification

- No Migration (Partitioned)
  - Task can not migrate
  - Job can not migrate
- Restricted Migration
  - Task can migrate
  - Job can not migrate
- Full Migration
  - Task can migrate
  - Job can migrate

a new period
p1
p2
a new period
p1
p2
a new period
p1
p2

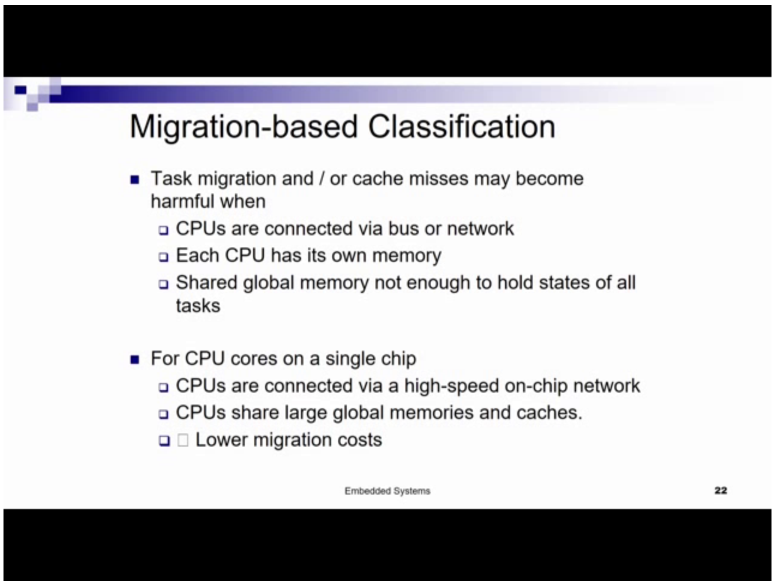
Embedded Systems 21

Firstly, we have no migration, but the task cannot migrates and the jobs cannot migrate. We can have restrict the; this is the case which shows that so, these are 2 distinct jobs and the jobs have distinct subtasks. Each of these boxes are subtasks of the same job. So, the job consists here of 3 subtasks. And all the subtasks of all the jobs of this task must execute on the same processor P 1.

And therefore, it is a completely partitioned approach. In the restricted migration case, we the tasks and migrate, but the jobs themselves cannot migrate. So, all subtasks of the same job must remain on the same processor; however, 2 separate jobs can be executed on 2 separate processors. And we can also have full migration where the tasks and migrate the jobs can also migrate.

So, the first subtask of job of the first job executed executes in P 1, the second sub task of job one executes in P 2, the third sub task of job 1 executes in P 1 again. The first subtask of job 2 executes in P 2. The second sub task of job to executes in P 1. Third sub task of job to executes again in P 1. So, this is a case for full migration.

(Refer Slide Time: 25:35)



Migration-based Classification

- Task migration and / or cache misses may become harmful when
  - CPUs are connected via bus or network
  - Each CPU has its own memory
  - Shared global memory not enough to hold states of all tasks
- For CPU cores on a single chip
  - CPUs are connected via a high-speed on-chip network
  - CPUs share large global memories and caches.
  - □ Lower migration costs

Embedded Systems 22

So, when we are doing a global scheduling, it allows a lot of tasks migrations, which leads to cache misses. And this cache misses can become harmful, especially when the CPU's are connected via bus or network. That is it is a distributed system. And therefore, taking the task from one processor to the other is costly. In a distributed system you do not have a shared cache. So, so hence, the code of the task as well as the data cannot be stored in the shared cache so that it can be easily made available to another processor. And hence the entire code and data must be must be transferred over bus to another processor.

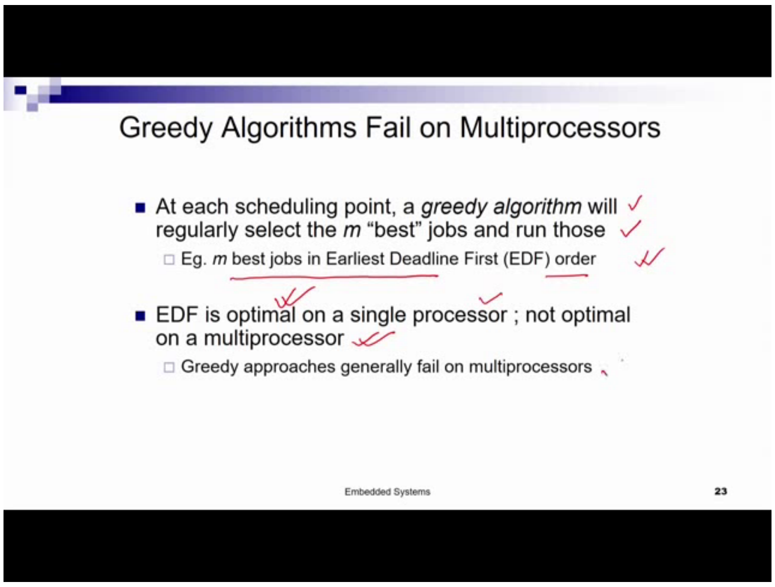
So, therefore, these such task migrations are harmful when CPU's are connected via bus or network. When each CPU has it is own memory, it still is much more harmful if the CPU if the memory itself not even cache, but the memory itself is separated. So, therefore, you have to transfer the whole. It is not even in memory. So, you have to transfer the whole task from the memory of one processor to the memory of the other processor. In the first case if the memory share you at least do not need to do a bus transfer.

It is there at least there in the shared memory. But if the memory is also separate, then you need to do a complete bath bus transfer of both the code as well as the data of the of the task from one processor to another. So, and if the shared global memory, even if you have a global memory, if the shared global memory is not sufficient to hold all states of the task, then also such tasks migrations or cache misses become harmful in terms of the overheads it has.

And it is not that harmful for CPU cores on the symbol single chip. For CPU cores on a single chip, even if you have task migrations, it is not that harmful because CPU's are connected via via may be connected via high speed interconnection on chip network.

Therefore, the task misses are not that time consuming. So, the transfers or in the transfer of both code and data is not that time consuming. And hence the overheads are less. CPU share a large global memories and caches. And hence the transfer has less cost. Therefore, whenever we have lower migration costs. We can use a global approach we can go towards the global approach when the migration costs are less; however, when the migration costs are more as in a loosely coupled distributed setting. Then we must go towards more partitioned approaches.

(Refer Slide Time: 28:22)



Greedy Algorithms Fail on Multiprocessors

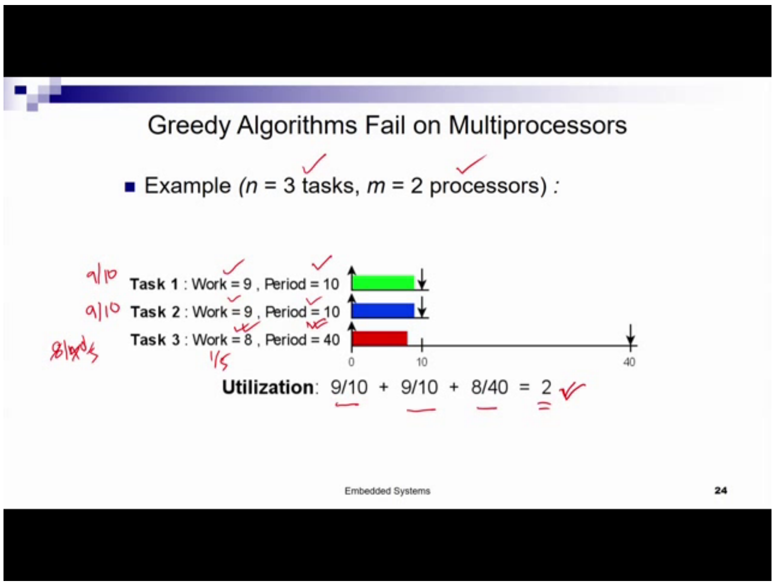
- At each scheduling point, a *greedy algorithm* will regularly select the *m* "best" jobs and run those ✓ ✓
  - Eg. *m* best jobs in Earliest Deadline First (EDF) order ✓
- EDF is optimal on a single processor ; not optimal on a multiprocessor ✓
  - Greedy approaches generally fail on multiprocessors

Embedded Systems 23

Greedy algorithms generally fail on multi processors. Why? At each scheduling point a greedy algorithm will regularly select the m based jobs to run. For example, we will select the m best jobs in the earliest deadline first order. So, we have a multiprocessor, have we have a set of m processors we have a multiprocessor system consisting of m processors. A greedy algorithm, even if it is a global algorithm what will it do; the greedy scheduling algorithm will choose the m based tasks and allocate on to m processors.

For example, if we have an earliest deadline first, we will take the m best jobs in term of their earliest deadlines. These stars will have the earliest deadlines and allocate on to m processors in order, ok. But this strategy fails on multiprocessors and we will see why. So, that is why although EDF is an optimal algorithm as we have seen on a single processor it is not optimal on a multiprocessor system. So, greedy approaches generally failed on multi processors.

(Refer Slide Time: 28:34)



Greedy Algorithms Fail on Multiprocessors

- Example (n = 3 tasks, m = 2 processors) :

9/10 **Task 1** : Work = 9 , Period = 10

9/10 **Task 2** : Work = 9 , Period = 10

8/40 5 **Task 3** : Work = 8 , Period = 40

1/5

Utilization: $\underline{9/10} + \underline{9/10} + \underline{8/40} = \underline{\underline{2}}$ ✓

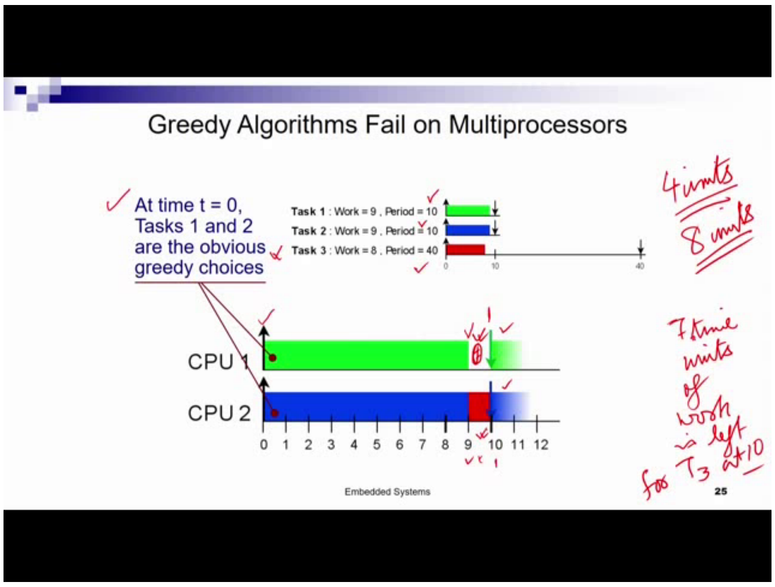
Embedded Systems 24

Now, let us take an example to see why this happens. Let us say we have 3 tasks to be scheduled on 2 processors. The 3 tasks have task one has an execution requirement of 9 to be scheduled within 10 within a period of 10. So, it has an utilization of 9 by 10. The second tasks also has an utilization of 9 by 10. So, task 2 has a work of 9 or execution requirement of 9 to be completed within a period of 10. And the third task has an utilization of 8 by 40, ok or 1 by 5. So, the third task has a neutralization of 1 by 5, it has an execution requirement of 8 to be completed within a period of 40.

If we add these utilizations, we see that 9 by 10 plus 9 by 10 plus 2 by 10 which is equals to 2. So, the total utilization is 2. And therefore, it the total the total capacity of the system is also 2, 2-unit capacity processors. So, in the ideal case we should be able to schedule on these 2 processors. We should be able to schedule these 3 tasks feasibly on to these 2 processors.

However, we will see that if we use greedy strategies like EDF it is not possible to do so.

(Refer Slide Time: 31:09)



Greedy Algorithms Fail on Multiprocessors

At time t = 0, Tasks 1 and 2 are the obvious greedy choices

Task 1 : Work = 9 , Period = 10

Task 2 : Work = 9 , Period = 10

Task 3 : Work = 8 , Period = 40

Embedded Systems

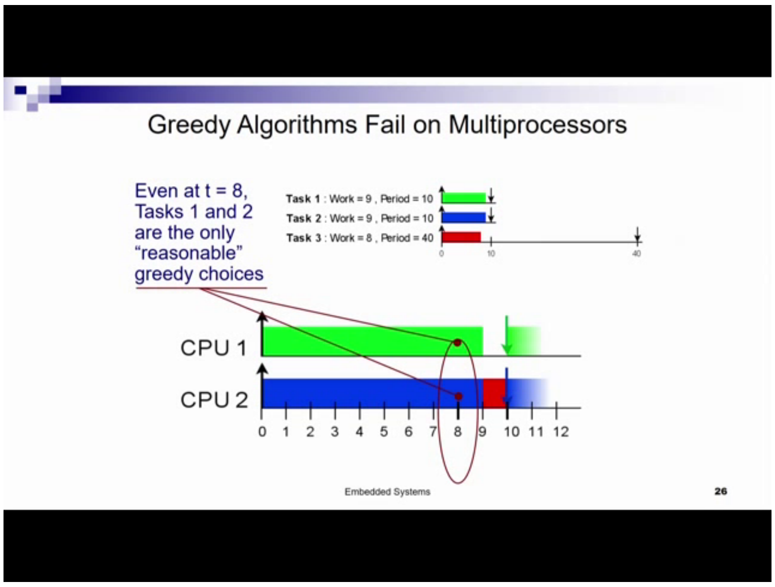
25

So, what happens at time 0? At time 0 tasks t 1 and t 2 are the obvious greedy choices, why? Task t 1 has a deadline of 10 tasks t 2 also has a deadline of 10, tasks t 3 has a deadline of 40. So, these 2 have task 1 and task 2 are the ones with the earliest deadlines and are scheduled at time 0. And if we see that you know their jobs complete at time 9. Their jobs complete at time 9, at time 9 only the third task is left, and it is executed let us say on CPU 2 for one-time unit. And therefore, say at 10 7 time units of work for task 3 is still left, 7 time units of work, 7 time units of work is left for 3, for left for t 3 at 10, ok. And then at 10 the next jobs of t 1 and t 2 arrive again.

So, again t 1 and t 2 are the greedy choices. The best greedy choices and t 3 does not get a chance. T 3 again gets a chance at 19, because at 19 both t 1 and t 2 will finish at 19, and then between 19 to 20 in that time slot t 1 will execute for one slot again. And then again it will again get another chance at 29 and another chance at 39, but then by 40. We understand that t 3 will only complete 4 units of work. But it needs to complete 8 units of work. If you see why this happened we see that one freeze every 10 time units, we have to keep the CPU idle for one-time unit.

We cannot schedule anything here in this in this place, why? Because we have nothing to execute T 3 has it is executing on the other processor, and it cannot execute at the same time on 2 processors. We cannot have overlapped executions. And therefore, due to these free slots these idle slots of the CPU, everyone every 10 time units one idle slot of the CPU this makes, this makes this task 3 miss it is deadline at 40.

(Refer Slide Time: 33:42)



Greedy Algorithms Fail on Multiprocessors

Even at t = 8, Tasks 1 and 2 are the only "reasonable" greedy choices

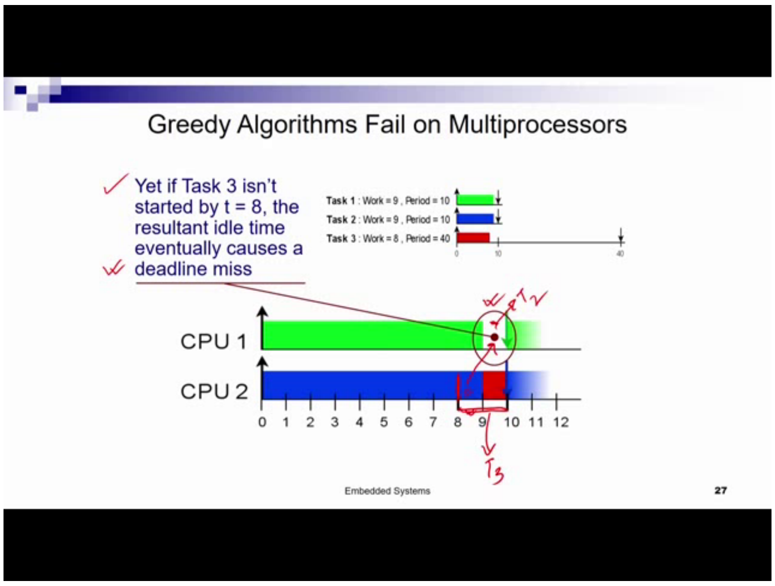
Task 1 : Work = 9 , Period = 10
Task 2 : Work = 9 , Period = 10
Task 3 : Work = 8 , Period = 40
0
10
40
CPU 1
CPU 2
0 1 2 3 4 5 6 7 8 9 10 11 12

Embedded Systems

26

So, however, what happens there is something important that happens at 8 we which we must be missing here. So, even at 8 what happens? The task t 1 and t 2 are the only reasonable greedy choices, why? Because at 8 we see that it is if you see even at 8, task t 1 has a deadline of 10, tasks t 2 has a deadline of 10, while task t 3 has a deadline of 40. And therefore, at 8 also we schedule both t 1 and t 2 and we do not we do not execute t 3 up to this unit, this 9.

(Refer Slide Time: 34:22)



Greedy Algorithms Fail on Multiprocessors

- Yet if Task 3 isn't started by t = 8, the resultant idle time eventually causes a deadline miss

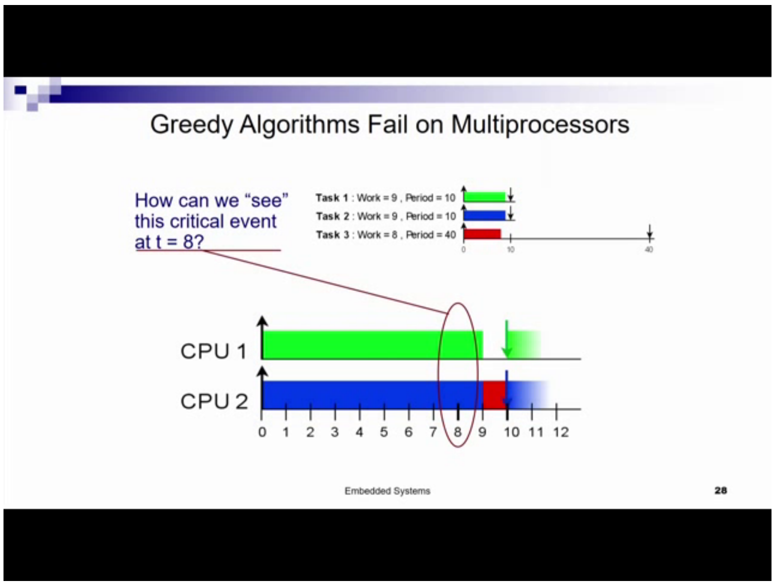
Task 1 : Work = 9 , Period = 10
Task 2 : Work = 9 , Period = 10
Task 3 : Work = 8 , Period = 40
CPU 1
CPU 2
0 1 2 3 4 5 6 7 8 9 10 11 12
T_3

Embedded Systems 27

Yet if task t 3 yet, if t task t 3 isn't started by 8 so, why did we talk about this 8 why was 8? So, important if task t 3 is not started by 8 the resultant idle time here eventually causes a deadline miss.

So, if task 3 was started at 8 what would happen? Suppose, we executed this blue task which is t 2 up to 8, and then we executed task t 3 for these 2 time units here, then what would happen? We could migrate this task from here to here; this task t 2 and one remaining slot of execution of t 2 could be executed here, ok. So, this place t 2 could have executed. But then for that we need to understand that at 8 we have something important. At 8 we need to you we need to allocate t 3, otherwise ultimately at 14th will miss deadlines that has to be understood. So, how do we make the system understand this.

(Refer Slide Time: 35:31)



Greedy Algorithms Fail on Multiprocessors

How can we "see" this critical event at t = 8?

Task 1 : Work = 9 , Period = 10

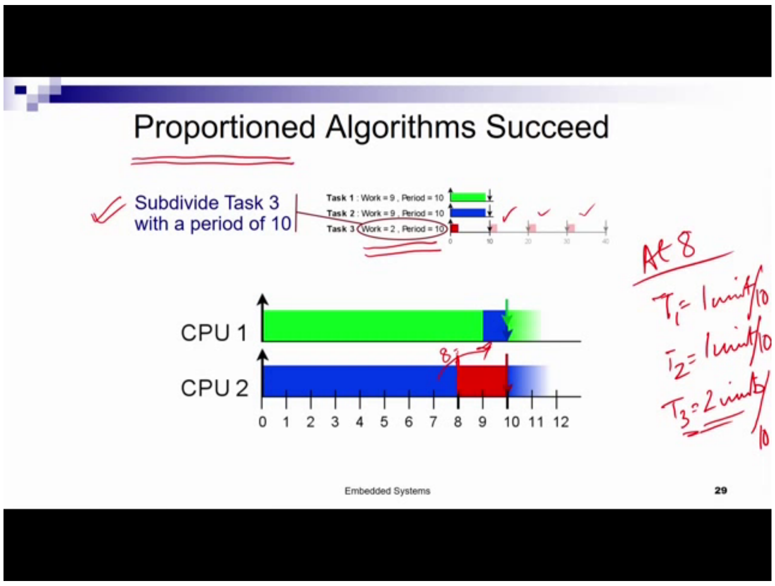
Task 2 : Work = 9 , Period = 10

Task 3 : Work = 8 , Period = 40

Embedded Systems 28

So, how can we see this critical event at 8 is the question.

(Refer Slide Time: 35:34)



Proportioned Algorithms Succeed

Subdivide Task 3 with a period of 10

Task 1 : Work = 9 , Period = 10

Task 2 : Work = 9 , Period = 10

Task 3 : Work = 2 , Period = 10

At 8

$T_1 = 1 \text{ unit}/10$

$T_2 = 1 \text{ unit}/10$

$T_3 = 2 \text{ units}/10$

Embedded Systems 29

So, what can we do? We can subdivide task 3 with a period of 10. Now at we if we subdivide task 3 with a period of 10, we see that every period of 10, it needs to do it needs to have do a work of 2 units. So, within that within 10, if it will do to you it must do 2 units of work, within 20 it must do 2 units of work within 30 it must do 2 minutes of work and within 30 to 40 also it must do 2 units of work, in order to complete all, it is 8 units of required execution.

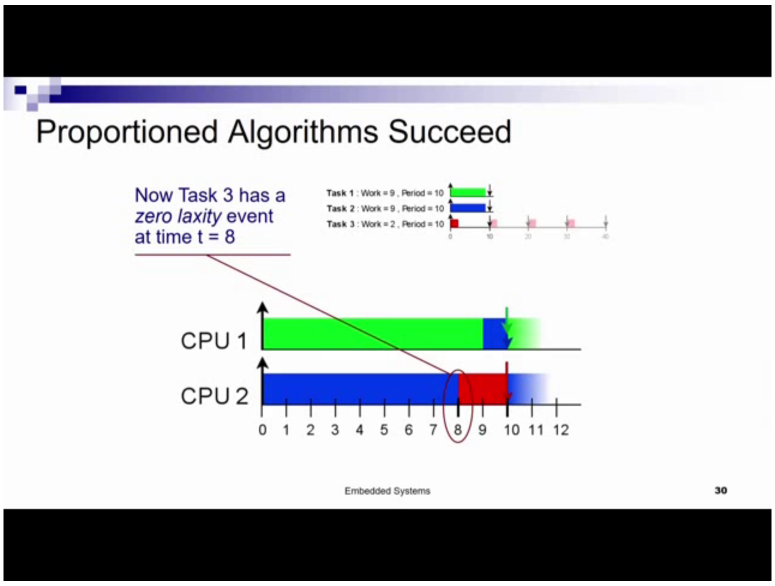
Then what happens? We see that then we can if we do this, if we do this, with this if we do this proportional division of the workload of task t 3 into 4 chunks of 2 units each for every 10 time units of work. This proportioned this proportion division of work for T 3 makes it possible to view this important event at 8. We see that at 8 tasks 1 has 1 unit of work remaining, task 1 has 1 unit of work remaining, task 2 also has 1 unit of work remaining, but task 3 has 2 units of work remaining.

So, at suddenly at 8 T 3 becomes more important than T 1 and T 2, because it must it has 2 units of task remaining to be completed before 10. T 1 has 1 unit of come work remaining to be completed before 10. T 2 also has 1 unit off this happens at 8. At 8 T 1 has 1 unit of work remaining to be completed within 10 T 2 also has 1 unit of work remaining to be completed within 10. But T 3 has 2 units of work remaining to be completed within 10. So, in this sense t 3 becomes more important than T 1 and T 2 and T 2 at 8.

And hence T 2 for in this case T 2 is preempted and T 3 is allowed to execute on to CPU 2, right. And then you have one unit of work remaining which can be executed in CPU 1 after T 1 completes his execution at 9 by incurring this migration.

Hence proportioned algorithms which does what which proportionately divides the workload with respect to deadlines of tasks succeed in multiprocessor systems. So, if we can determine which if we can find what are the deadlines for different tasks in the system, and by all these deadlines we proportionately execute the workloads of tasks we will succeed on multi processors.

(Refer Slide Time: 38:41)



Proportioned Algorithms Succeed

Now Task 3 has a *zero laxity* event at time t = 8

Task 1 : Work = 9 , Period = 10
Task 2 : Work = 9 , Period = 10
Task 3 : Work = 2 , Period = 10
CPU 1
CPU 2
0 1 2 3 4 5 6 7 8 9 10 11 12
Embedded Systems
30

So, this is the takeaway here. And so, now t 3 has a 0 laxity event at 8. So, 0 laxity event meaning, that it has 2 units of work to be completed within 10. So, amount of work remaining is 2, amount of time remaining is also 2. So, laxity is 0. So now, task 3 has a 0 laxity event at 8. And because the laxity is 0, tasks t 3 becomes more important than task t 1 and t 2, because still they still have a laxity of 1 at 8. And hence we you we allocate t 3. And therefore, all the 3 tasks can meet their deadlines using this approach.