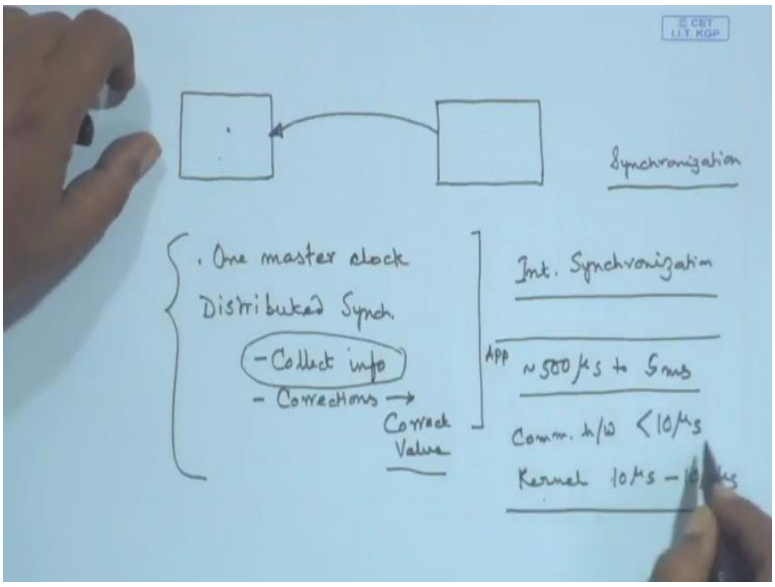
Embedded Systems Design
Prof. Anupam Basu
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 27
Real Time O.S – II

So, in the last class we had looked at time and we have seen two types of representations one is TAI and another is UTC.

(Refer Slide Time: 00:32)



Now in many situations when we have got multiple computers working for solving some problem which is also very realistic situation in the present embedded systems when there are multiple processors working for the overall objective of meeting some constraints or delivering certain things. For example, in the automobiles there are different dedicated processors for different tasks. So, there is a necessity of synchronization between the clocks, whenever I have got more than one clock I need to have synchronization among them, so that they count together.

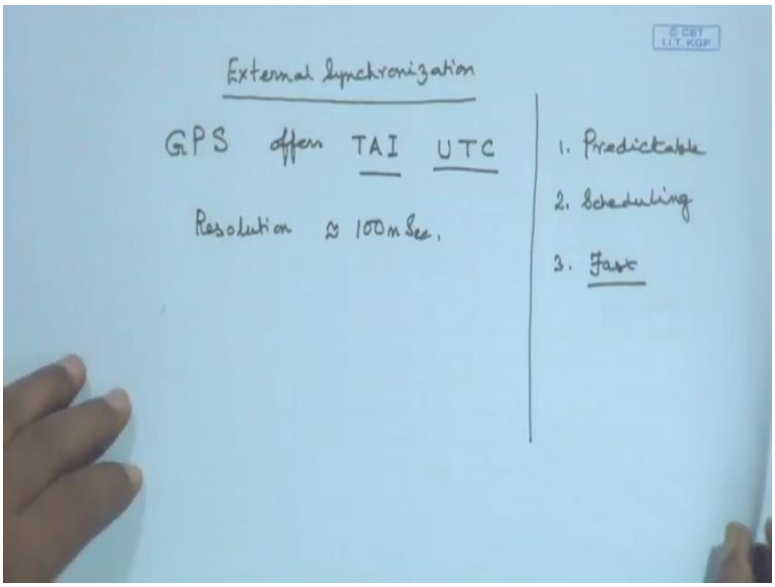
So, if I have one master clock one thing is that I have got one master clock and everybody is adhering to that master clock then that is fine for internal synchronization. In other situations when there are distributed synchronization, there is also internal synchronization then we collect the information from the neighbors; say for example, I get its time and I first collect the information from the neighbor and then apply the

corrections and get the correct value so; that means, as if I want to check my time I may ask you what is the time? And accordingly I adjust my time like that and we set the correct value; that is how we do for internal synchronization.

For example this process will seek the time from here and will correct it itself. Now the precision of this that is collecting the information from the neighbors; how frequently that will be done that varies from the application level. For example, if it be the application layer then it might be at the level of microsecond; for example, around 500 microsecond to 5 millisecond; at this interval I want to check, whereas if I come to the communication hardware, it becomes much more faster.

For example, whenever I come to communication hardware then it is less than 10 microsecond, I communicate much faster. At the kernel level for example, when I come to the kernel level; it is somewhere in between like 10 microsecond to 100 microsecond. So, the precision of step whenever I mean these are the levels of precision that I want that the match between this one and this one, at the application layer; let me write it down if the application layer, it should be within this, the error should not be more than this whereas, for the communication hardware; it should be much more accurate.

(Refer Slide Time: 04:15)

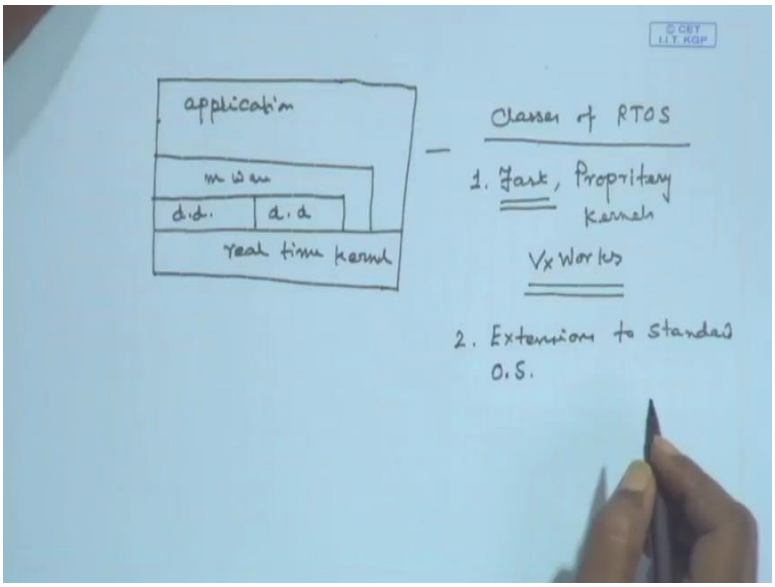


That is for internal synchronization, but sometimes we also do external synchronization and external synchronization what we do is we usually use a GPS. Nowadays the trend is to use the GPS, the GPS actually offers both TAI and UTC both timing information are

provided and we take the GPS and synchronize with respect to that and in this case the resolution is of about 100 nanosecond resolution, so that is how we deal with this.

Now what are the things we have done, we have seen what are the characteristics of the operating system or real time operating system; one is it must be predictable; the requirements number 2; it must be doing the task of scheduling, we look at scheduling in much more detail later and third it must be very fast, the operating system must be fast because of meeting real time constraints, it should not take too much time to take the decisions. Then we saw that how we carry out the synchronization and all those.

(Refer Slide Time: 06:23)



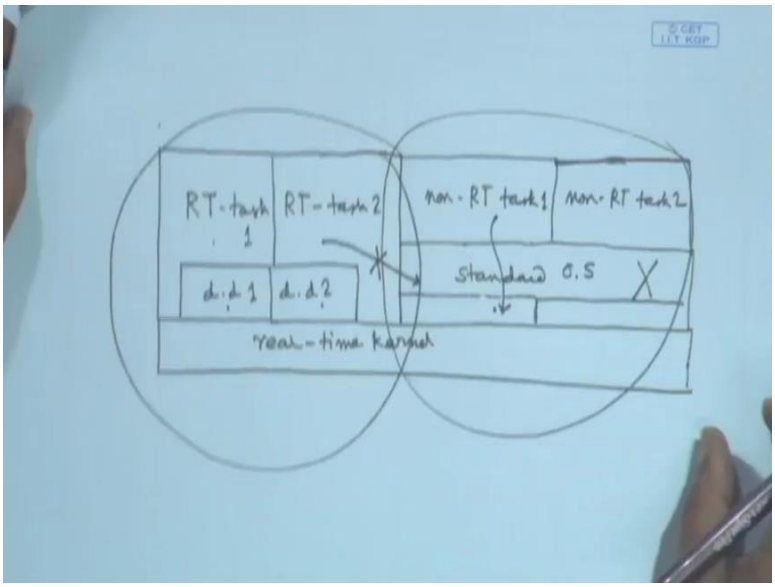
Next we move to the structure of the real time kernels. So, we have already seen in the last class that we had a layer like this that the application layer was there at the top and the real time kernel was at the bottom and the device drivers on the stack, device drivers were there and then there was some middleware and the application software was running on this; that was the scenario compared to the operating system coming into in between these layers in the case of a standard operating system.

Now, this can be thought of our general RTOS; Real Time Operating System or there can be RTOS for specific domains, I can have specially tuned real time operating systems as well. So, the classes of real time operating systems are one is; fast and proprietary; there is a not open kernels. Typical example of this is the V x works, now

what is the advantage of these which are proprietary made for a particular domain that they are fast, but they are not predictable.

Typically it has been found that they are not predictable for all situations, they have been designed specifically for a particular application tuning it so that it is fast for that and that is a proprietary kernel, but it has been found that it has not been tested for all the situations; therefore, the predictability is still remains an issue in this case. The second category is the RT extension to standard OS, where extending the standard OS to generate RT.

(Refer Slide Time: 09:31)



Now, I will draw a diagram which will illustrate something interesting, some of the advantages of this; see I have got a real time kernel at the base. Now I am creating two compartments here on this side, on this I have got my device drivers. So, device driver 1, device driver 2 and on this I have got real time tasks say real time task 1, say RT task 2.

On the other side, I have got the standard OS running on this kernel, running on the same kernel and on this side I have got non RT task. Now what is the advantage of this? If I you know often in your PC or in your windows machine, the system crashes. Even if there be a crash of the standard OS, the real time tasks will not be affected. The disadvantage is that the standard, there are some facilities or some functionalities given in the standard OS, but the real time tasks; these cannot access those functionalities, they cannot access those functions that is the downside of this.

But this is fine, but like one typical example of this is the RT Linux which is build on the Linux kernel, but these are the problems that occur in that case and of course, because of this standard OS; it is a little heavy and it is not that fast as the proprietary kernels are. It may not always be the case; in that case what can happen is that the question is that why is it the device drivers are not accepted by this.

Now, in my diagram here under this standard OS there are device drivers also, there are device drivers in this zone also, but these device drivers are under the control of the standard OS. Therefore, whenever a task once to access this device drivers that request has to be routed through the standard OS, which can establish mutual OS, which can make you wait all those things, but that is not a desirable situation and neither is it a required situation for many real time tasks.

So, the dedicated devices I can keep out of the purview of the standard OS.

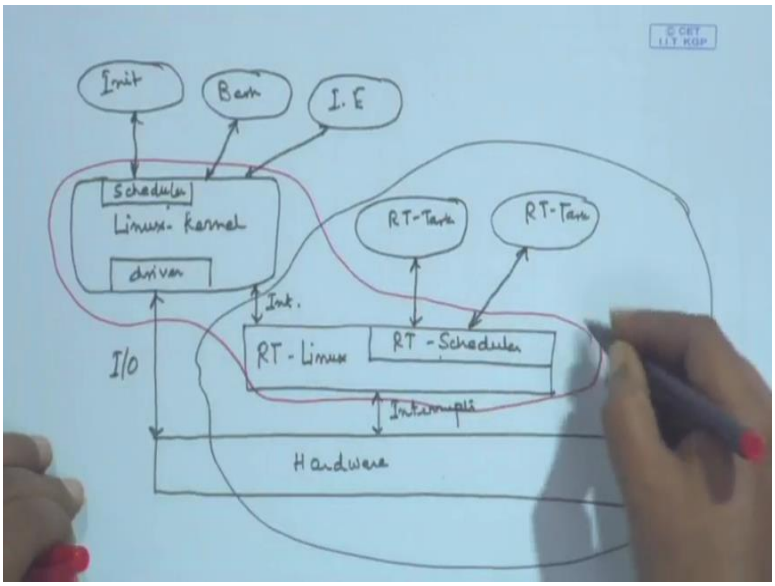
Student: (Refer Time: 13:17).

Now that is the problem that you cannot do.

Student: So how we can (Refer Time: 13:26).

No that is it is not a general OS, whenever there is a general OS part; only up to this part the general OS part and this part is a real time thing, but they are using same kernel. So, I had the real time; I had this standard OS initially, I had the kernel; I extended this and I have connected my devices on this so that my real time tasks run on this, I will show you another example for this.

(Refer Slide Time: 14:12)



See for example, I have got the hardware here and on this I have got the say real time Linux in which there is a real time scheduler; a scheduler is there and I have got real time tasks, these are communicating with this scheduler. The scheduling policy of the standard OS will be different from the real time scheduling policies also we will see that soon and so this is communicating through some interrupts and on this side, I have got the Linux kernel where I have got a scheduler and the drivers here.

The drivers are connected; I mean communicating via I operations with the hardware through this kernel and this they can also connect through interrupts here and on this are my very well known processes like say Init, bash or may be internet explorer whatever or Google chrome they are running on this. Now, if there be any crash on this; this part will not be affected all, so here I have got a separate scheduler for both of them. The reason for having a separate scheduler will be evidence soon when we will see that may be next couple of lectures, when we will see that the scheduling policies that we are familiar with regarding in standard OS are always not applicable for real time operating system.

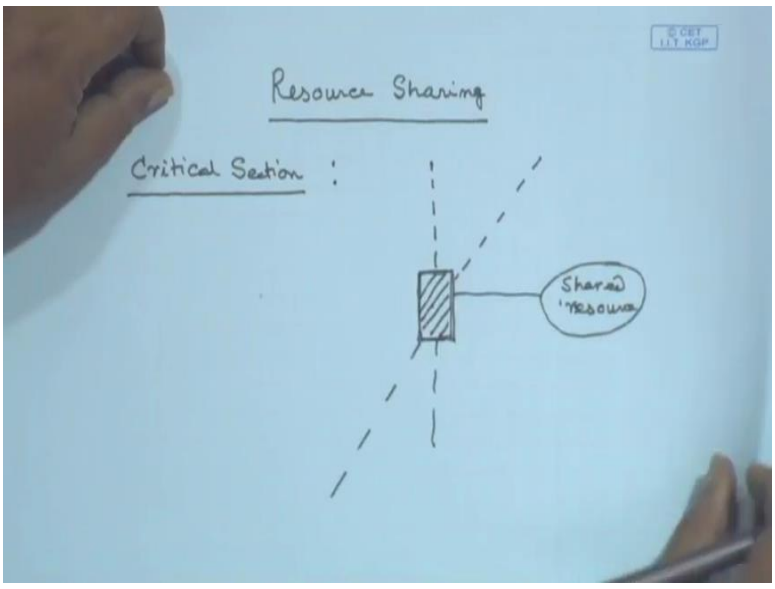
So, yes needs to sit in the same system I mean; so for example, if you want; no no that is just an idea for example, the question is that as this diagram is showing, it is the same hardware is being shared by both of them is not the case like that. The case is that forget about the hardware, I can hack separate out the hardware there can be an embedded system which is on this, we are right now discussing about the operating system part.

Now, this operating system that has been defined to co-exist along with the general operating system, so this part let me put some other color; these two are as extensions, the red ones; these ones are extensions as was shown in my earlier diagram, where did it go? So in this diagram, so the operating systems are sitting on the kernel and, but the standard operating system is using some kernel function, the real time tasks are since some other kernel function. The device drivers are separate and the non real tasks are accessing the kernel through the standard operating system and this one has got the option of not going to the standard operating system that is all.

Now, whether that the type of diagram that I have drawn here that as if they are on the same hardware that is not necessary because for an embedded system, I do not expect that the same hardware will loaded by a non real time task as well as real time task. So, this hardware can be different actually will be different, but the point is that how the operating system interacts with the hardware, there are two separate parts for doing that is what is been proposed. So, with this we have looked at the requirements of real time operating systems in general.

Now we will look into some other aspects of real time system, gradually we will move to the scheduling part, but before that as this has been said the timing and predictability are two very important issues for real time operating systems.

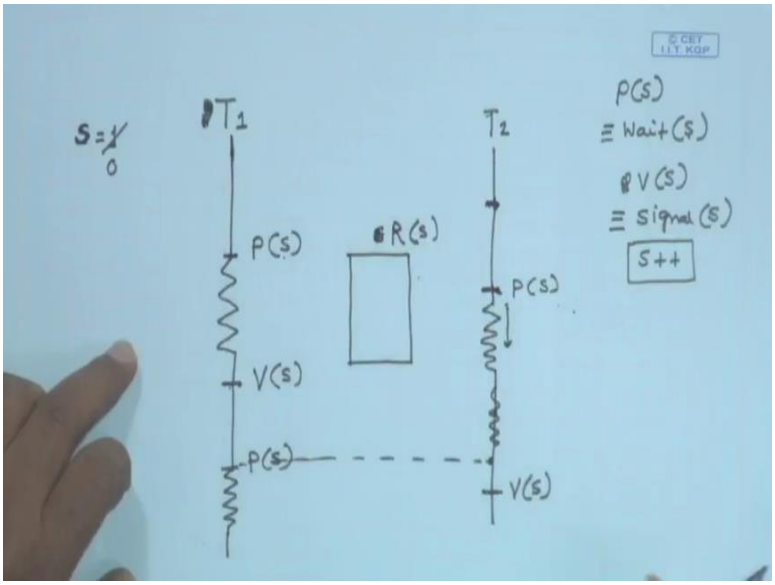
(Refer Slide Time: 20:13)



One of the major impediments to predictability is resource sharing, for example if you want the resource and you may not get that resource, so how do you handle that? Or so this problem of resource sharing, we have also encountered in the normal operating systems course where we let us quickly revised what we mean by critical section, what is critical section? Critical section is that part say here is my code, the piece of code that interacts with the shared resource that used to access some shared resource.

For example, a driver; that driver is accessing this shared resource and there is another state of the program which also wants to access the same shared resource. Now for that it will use the same driver, now are we going to allow both of them access this piece of code at the same time? That implies that will be accessing this shared resource simultaneously which often may not be admissible because if there be a two rights or one read or one write then the results can be unpredictable and the results can be also erroneous.

(Refer Slide Time: 22:21)



So, we need to employ some sort of mutual exclusion over that and we know that in order to ensure mutual exclusion for example there is let me call it task 1 and task 2. So, when task 1 executes before it access a critical section; it performs a P operation; P means the wait operation P S; where S is the semaphore is equivalent to wait on S and you know that this semaphore is nothing, but an integer on which only two atomic operations are can are applicable only two and they are single and wait.

Now, the critical point here is that each of them must be atomic; that means, when a process is performing wait S, then another process cannot perform wait S or single S on this particular S; S is a particular semaphore variable and what does wait is do; wait is simply decrements the value of the semaphore by 1. So, suppose process T 1 is executing and here it wants to enter the critical section here, it performs a wait. Suppose just for those whom might not have done an operating systems course, I am just repeating this, but I am sure most of you have done that.

Suppose this semaphore S was initialized to 1 and semaphore S is a shared variable; now when I do P S or wait S; that means, this value of S gets decremented to 0 and suppose now T 2 also starts and at this point, it will starting a little late may be T 2 has started little late and sorry not here may be somewhere here; T 2 wants to enter the same critical section, same driver it wants to access, same resource it wants to access and it tries to do a P S because this piece of code is protected or gated by the safety valve S semaphore.

So, what happens is now, but the resource is already occupied by T 1; therefore, value of S is 0. The function of wait is what I said was little incomplete; the function of wait is it will first check whether S is greater than 0 if so it will decrement; otherwise it will continuously wait on this value to be 1 clear? Wait is actually first checks the value of S, if S is greater than 1 then decrement S and proceed, otherwise go on checking the value of S till it becomes greater than 0.

So, since the task T 1 is inside this critical section, task T 2 is trying to enter that and is doing this P S; it cannot enter and when this one completes the task of the critical section, it performs V S; V S is equivalent to signal S; signal S means S plus plus. So, it increments S; I have skipped some of the details of wait S because this busy waiting that one process T 2 for example, was continuously checking the value of S that overhead can be avoided by putting them in a queue and those details implementation details I am not discussing, those details are basically operating system issues and there are different ways or by which interrupt can be utilized in order to get rid of this thing called busy waiting.

Now so when this one comes out of this V S, then this one will succeed. Now suppose here again T 1 wants to enter the critical section, so it will perform a P S, but this is still going on. So, at that point this is locked and since this one has got done V S; this P S has

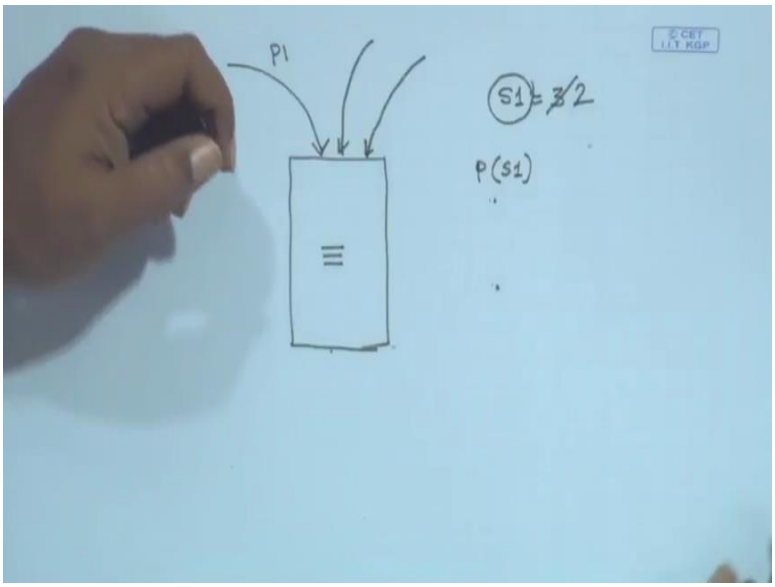
succeeded. Therefore, T 2 is now somewhere is here; now when T 2 exits this point, say here T 2 does V S only after that this can go in. Therefore, on the shared resource on a particular shared resource that has been gated by S is being provided mutually exclusion by the semaphore S; say the resource R is being protected by the semaphore S. Therefore, so what can be the consequence of this in terms of real time performance; I do not know the apriori what resources a task actually request for.

So, we will see how the timing of tasks gets effected because of this critical section issues.

Student: Are there different (Refer Time: 29:09).

Are there different access for different S because these are the different typically that should be the case, the question is whether we should keep separate semaphores for protecting the separate resources; yes that is the actual practice, otherwise there will be completely less condition; I mean one resource is being freed, but another one which is waiting for some other resource will get that. So, there should be separate semaphores the semaphores may be binary may not be binary also.

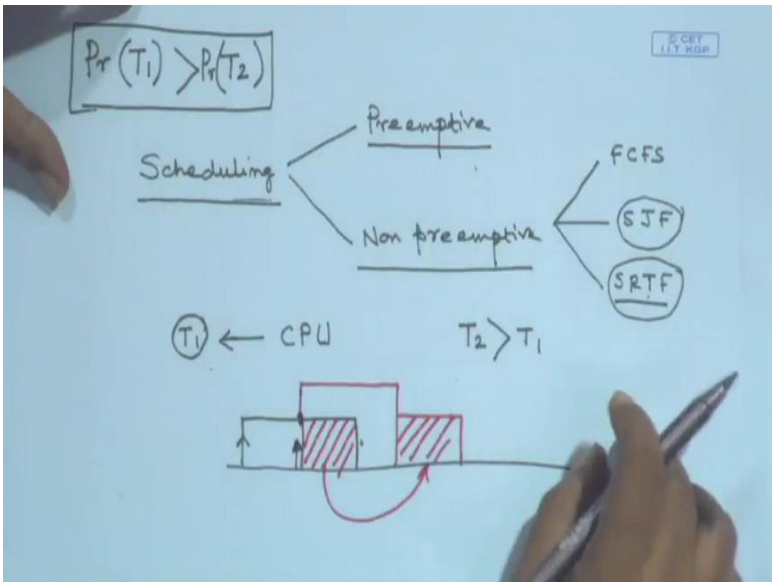
(Refer Slide Time: 29:57)



For example, may be in a particular application I may like to allow say some application; it really does not matter if three processes can go in simultaneously. I can allow three processes to go in, but not more than that, so in that case say I make it another

semaphore S 1, I can initialize it to 3. So, whenever the first process P 1 wants to get in it does P S 1 and S 1 becomes 2; that means, 2 more can still go in. Thus this value of this variable can also be utilized to know at any particular point of time how many processes are there inside the critical section.

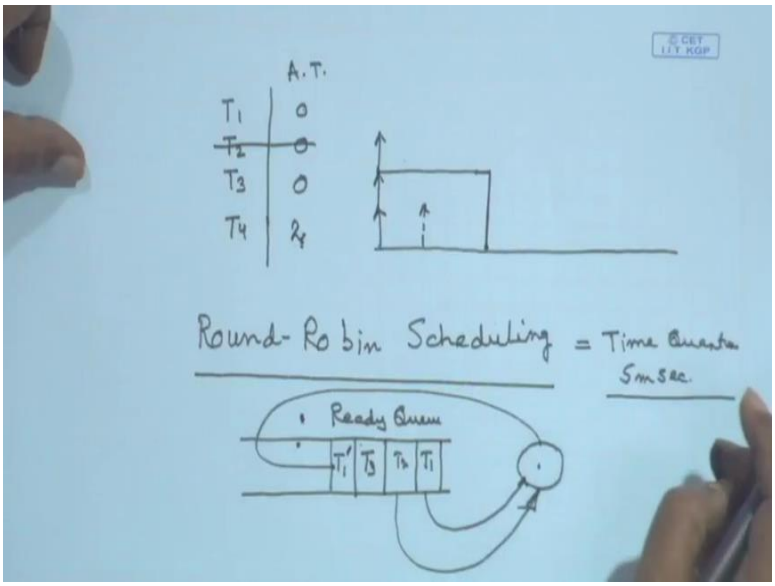
(Refer Slide Time: 30:59)



Now, let us take an example; let us right now consider two tasks T 1 and T 2 and we assume that the priority of T 1 is greater than the priority of T 2. So, this means let me write down priority of T 1 is greater than the priority of T 2. Now here I am talking of priority scheduling, but before going into that let me talk about; I will come back to this in a minute, before that let us talk about scheduling. Scheduling can be preemptive or non preemptive; what is the meaning of that? Even before that when we schedule a task what do we really do? What we schedule? We schedule the processor to that task.

Now, when I do non-preemptive scheduling; suppose the task T 1 is now has got the processor the CPU has been allocated to T 1. Now if another process and the task T 2 which has got higher priority than T 1 arrives now alright, then in a non preemptive thing I cannot touch T 1; I cannot do anything. So, the scenario is this T 1 came here and was started; at this point T 2 arrived, T 1 arrived here and it was started, T 2 arrived and T 2 has got the higher priority than T 1, but still since T 1 has been allocated the CPU; it will go on continuing; T 2 can only start after T 1 completes.

(Refer Slide Time: 33:34)



Now, if it be the case that initially I have got; it is non preemptive, now T 1, T 2, T 3 all of them arrived at the same say, the arrival time for both of them was 0, both of them arrived at the same time. So, in my timeline I have got all the tasks T 1, T 2 and T 3 all arrived here; which one should I allocate?

Now, that can be done in different ways for example, in a non preemptive scheduling we are aware of the operating system scheduling algorithms. For example, in standard one we have got the first come first serve, now here in this case the first come first serve does not apply because all of them have come together. Then we have got the shortest job first, we look at we have to have an priory idea of what is the runtime of T 1, T 2 or T 3 accordingly we schedule any one of them, for example, I schedule T 2.

Now once I schedule T 2, now suppose T 4 has come, which is shorter in length T 1, T 2, T 3, but it came at a time say T 2 here; here it arrived. Now if my shortest job first algorithm is non-preemptive then I cannot give any priority to this until this finishes. Once it is finished then T 2 has finished then I look at my remaining ones and see which one will have the highest priority, then this one will have the highest priority because it is shortest job, so this one will get that CPU time at that time.

But in the case of preemptive scheduling, if a higher priority task arrives for example, T 1 was running here and T 2 has arrived here; now if T 2 is of higher priority than T 1 then T 1 will be stopped here and T 2 will be started from this point and after T 2

finishes; this remaining part of T 1 will be carried out; this will be done here, so this will push back here.

Now, the question is how do we decide on the priorities? The priorities can be defined either by the processes themselves, we can assign apriory priorities that this task will have higher priority over this; we can create a hierarchy of priorities or by the scheduling policies for example, shortest job first gives a priority to the shortest jobs, shortest remaining time first; there is another shortest remaining time first, so that one will be again giving a priority dynamically. Now this can be this and this can be either preemptive or non preemptive, now before we will take up this blocking due to mutual exclusion in the next lecture, but let us conclude with a note on our standard scheduling algorithm that is a round robin scheduling algorithm.

Now, what happens in the case of Round-Robin scheduling algorithm; is it preemptive or non preemptive Round-Robin scheduling algorithm means now I can again have it in both ways; one is that I take, but round robin essentially what happens is there are tasks in my ready queue, there is a queue which holds all the tasks which are ready to be executed, waiting for getting the CPU. Now I can select from this queue based on some priority, I can select from this queue also based on first come first serve, but please note that first come first served is also a priority; what sort of priority? The priority is being determined based on your time of arrival.

Suppose I take this task and give it to the CPU then along with this in a round robin; I usually associate a time quantum say for example typically 5 millisecond, so this job will run for 5 millisecond, it will either complete if its execution time is less than 5 millisecond, otherwise it will join back the queue; T 1 second installment comes back here and then T 2 will get it, so this is also preemptive. The preemption is being done not because of the arrival of some other job, but the preemption is being done because the time quantum being elapsed. So, everybody has been given some time quantum that time quantum is elapsing, so that is being taken back.

So, we know now what is priority and the standard operating system scheduling algorithms, they can be either preemptive scheduling algorithms or non preemptive scheduling algorithms. In real time environment from the next lecture onwards, we will

be looking at preemptive scheduling algorithms only initially and then we will come up with other real time scheduling algorithms.