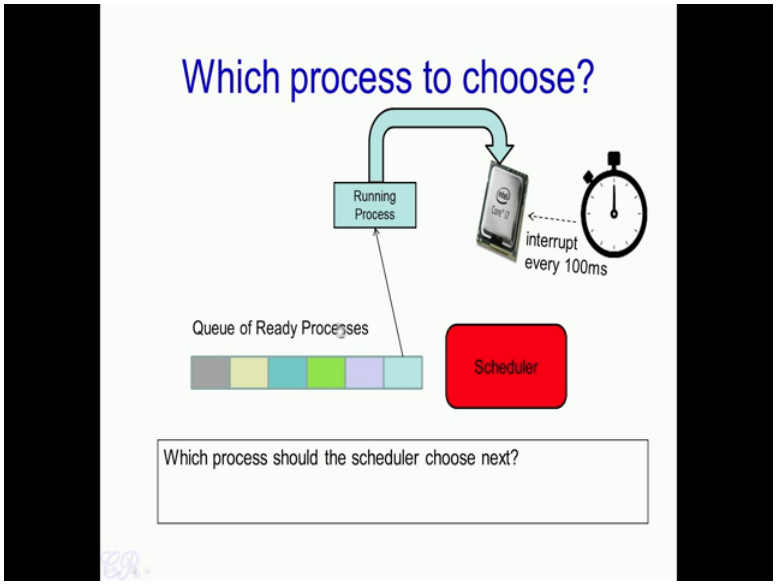
Introduction to Operating Systems
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week – 05
Lecture – 18
CPU Scheduling

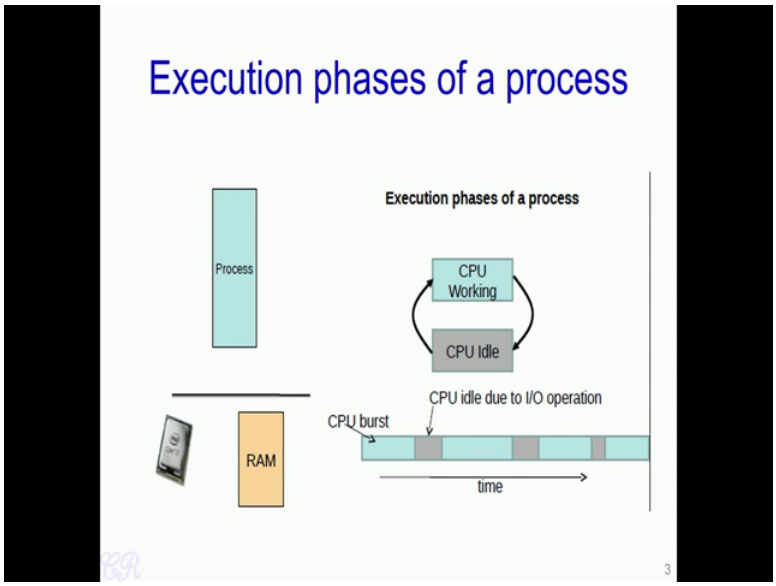
Hello. In this lecture, we will look at CPU Scheduling Algorithms.

(Refer Slide Time: 00:22)



We had seen in operating systems that a scheduler present would choose a particular process from the ready queue and that process is assigned to run in the processor. Now the question that we were going to analyze now is how should this scheduler (mentioned in above image) choose the next process? Essentially, how should the scheduler choose a process to run on the CPU from the existing queue of ready processes?

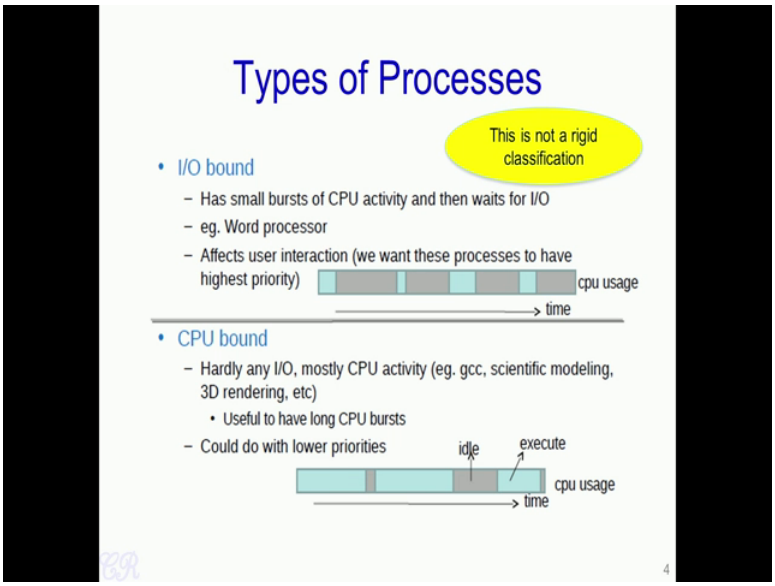
(Refer Slide Time: 00:53)



Now to analyze this, we first look at execution phases of a process. Now any program has been known to have two phases of execution; one is when it is actually executing instructions which is known as a CPU burst (mentioned in above image as blue boxes), while the other is when it is blocked on an I/O or not doing any operations, so in this particular case, the CPU is idle (mentioned in above image as grey boxes).

Thus, as we look with over time, a particular process would have some amount of CPU burst in which it would execute instructions on the processor, then it would have an some idle time in which it is waiting for a I/O operation. Then there would be a burst of CPU time and so on. Thus, there is always this inter-leaving between CPU burst and idle time that is a waiting for an operation.

(Refer Slide Time: 01:51)



So, based on this cases of execution (CPU working and CPU idle case) one could classify processes into two types. One is the I/O bound processes, while the other is the CPU bound process. Why do you make this distinction between I/O bound and CPU bound processes? Essentially, this is from a scheduling perspective, we would like to give I/O bound processes a higher priority with which they are allocated the CPU. Essentially, we want that I/O bound processes wait lesser time for the CPU compared to CPU bound processes. So, why is this required, we can take with an example.

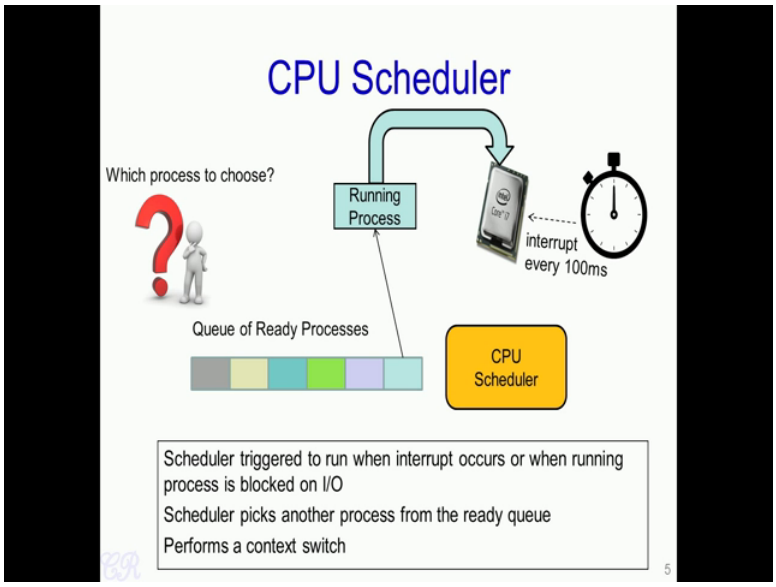
So suppose we are using a word processor such as a notepad or Vim and we are giving this I/O bound process a very low priority. Now, suppose a user presses a key, because it has a low priority it does not get the CPU very often and therefore, it would take some time before that key pressed by the user appears onto the screen. So, this may be quite uncomfortable for the user, therefore we would like to give the I/O bound process such as the word processor higher priority with which it will get the CPU, so that the user interaction with the CPU becomes more comfortable.

On the other hand, if you look at CPU bound processes, we could give it a lower priority. Now for instance, if you take one of these applications, the CPU bound applications like for instance say let us say ‘gcc’ that is compiling a program and let us say you are compiling a large program which takes 5 minutes. Now it will not effect this user much if the time taken to compile that particular program increases from 5 minutes to say 5.5

minutes. Thus, a CPU bound processes could work with a lower priority. This classification between I/O bound and CPU bound is not a rigid classification, that is a process could be an I/O bound process at one time, and after some time it could behave like a CPU bound process.

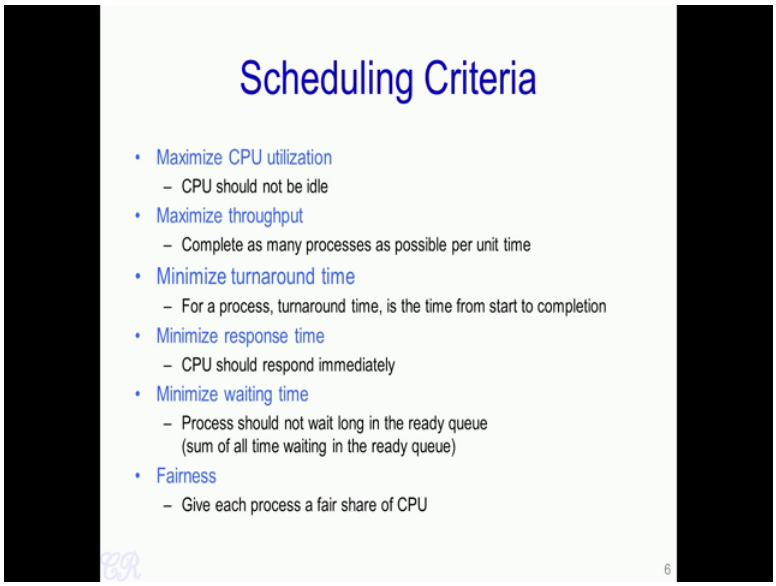
So, to take an example of a process which behaves both like an I/O bound as well as CPU bound, you could take for instance Microsoft excel. When we are actually entering data into the various cells in excel, it acts as I/O bound process. So, it behaves like an I/O bound process with small CPU burst and large times of I/O cycles. While on the other hand when you are actually computing some statistic on the data entered, Excel will behave like a CPU bound process, where there is a large portion of CPU activity or the time taken to actually operate on that particular data.

(Refer Slide Time: 04:38)



Now, let us come back to the question about how the CPU scheduler should choose from the queue of ready processes, the next process to execute in the CPU. There could be several ways in which the scheduler could makes this choice; essentially, there could be several CPU scheduling algorithms which would look into the queue and make a particular decision.

(Refer Slide Time: 05:05)



So in order to compare these various scheduling algorithms, operating systems text books or operating systems research defines several scheduling criteria. So, these criteria could be used to actually to compare various scheduling algorithms to see the advantages and disadvantages of each of them. So let us go through each of these scheduling criteria one by one. The first scheduling criteria is the CPU utilization. The scheduling algorithm should be designed in such a way so as to maximize CPU utilization. In other words, the CPU should be idle as minimum time as possible.

The next criteria, we will look at is the throughput. Essentially, scheduling algorithms would try to complete as many processes as possible per unit time. A third criteria is the turnaround time and this criteria is looked at from a single process perspective. So, turnaround time is defined as the time taken for a single process from start to completion.

The fourth criteria is response time. So, this is defined as the time taken from the point that when the process enters into the ready queue to the point when the process goes into the running state, that is the time taken from the instant the process enters the ready queue to the time the CPU begins to execute instructions corresponding to that process. Another criteria, is the waiting time. Now this criteria, is based on the time taken by a process in the ready queue. Now as we know processes ready to run are present in the ready queue and it is required that they do not wait too long in the ready queue. So,

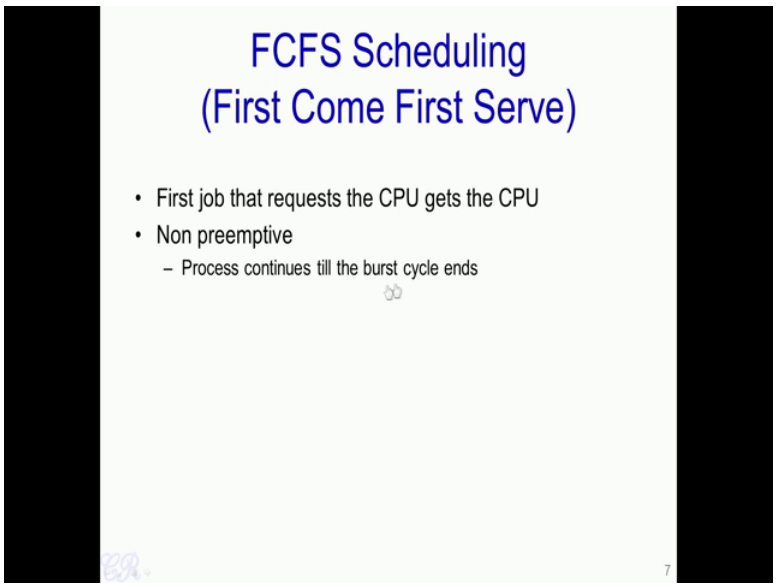
scheduling algorithms could be designed in such a way that the waiting time or the average waiting time in the ready queue is minimized.

The final criteria we will see now is fairness. The scheduler should ensure that each process is given a fair share of the CPU based on some particular policy. So, it should not be the case that some process, for instance, takes say 90 percent of the CPU while all other processes just get round 10 percent of the CPU. So, all these criteria need to be considered while designing a scheduling algorithm for an operating system.

A single scheduling algorithm will not efficiently be able to cater to all these criteria simultaneously. So, therefore, scheduling algorithms are therefore designed for to meet a subset of these criteria. For instance, if you consider real time operating system, the scheduling algorithm for that system would for instance be designed to have minimum response time; other factors such as CPU utilization and throughput may be of secondary importance.

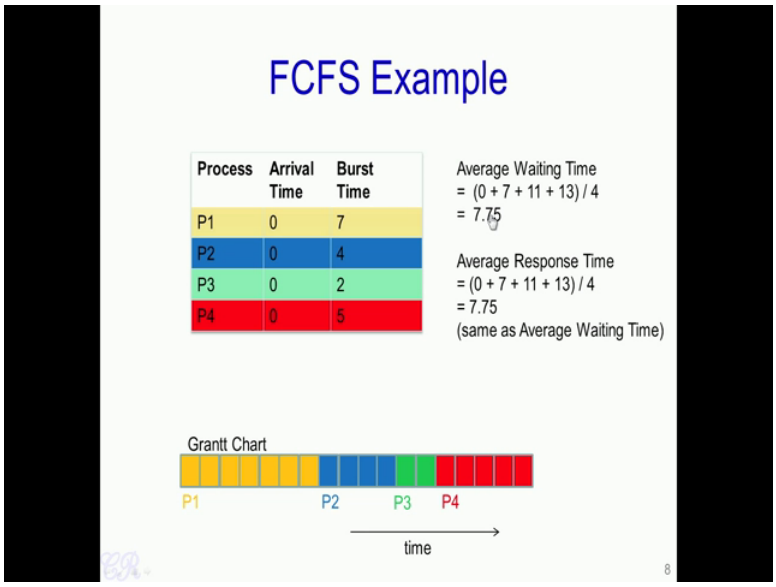
On the other hand, desktop operating system like a Linux will be designed for fairness, so that all applications running in the CPU or in the system are given a fair share of the CPU. Criteria such as response time may be less important from that perspective. So, we will now look at several scheduling algorithms starting from the simplest one that is the first come first serve scheduling algorithm and go to more complex scheduling algorithms as we proceed.

(Refer Slide Time: 08:42)



So let us look at the First Come First Serve of the FCFS scheduling algorithm. The basic scheme in this case is that the first process that requests the CPU would be allocated the CPU or in other words, the first process which enters into the ready queue would be allocated the CPU. So, this is a non preemptive scheduling algorithm which means that the process once allocated the CPU will continue to execute in the CPU until its burst cycle completes.

(Refer Slide Time: 09:19)



Let us see this with an example. Let us say we have a system with 4 processes running (1st column); the processes are label P 1, P 2, P 3, P 4 and they have an arrival time, so the arrival time is present in the 2nd column (mentioned in above image). So the arrival time is defined as the time when these processes enter into the ready queue. So, for this particular very simple example, so we will consider that all processes enter the ready queue at the same time that is at the 0th time instant. The 3rd column is the CPU burst time, so it gives the amount of CPU burst for each process.

For instance, P 1 has a CPU burst of 7 cycles; P 2 has a burst of 4 cycles. Thus, this particular table (mentioned in above image) we have like 4 processes, which all enter simultaneously into the ready queue at the time instants 0 and they have different CPU burst time, for instance P 1 has 7 cycles, P 2 - 4 cycles, P 3 - 2 cycles and P 4 - 5 cycles.

Now, we will see how these four processes get scheduled into the CPU or how these four processes get allocated the CPU. Since, all of these processes arrive at the same time, the

scheduler does not actually have a choice to make. So, he would pick randomly a particular ordering. For instance, let us say the scheduler picks P 1 to run, so P 1 runs for 7 cycles and when it completes, the scheduler picks P 2 and P 2 runs for 4 cycles. After P 2 completes its burst, and P 3 executes in the CPU for 2 cycles; and then finally, P 4 is scheduled in to execute for 5 CPU cycles (mentioned in above image). So, this is represented by a Gantt chart.

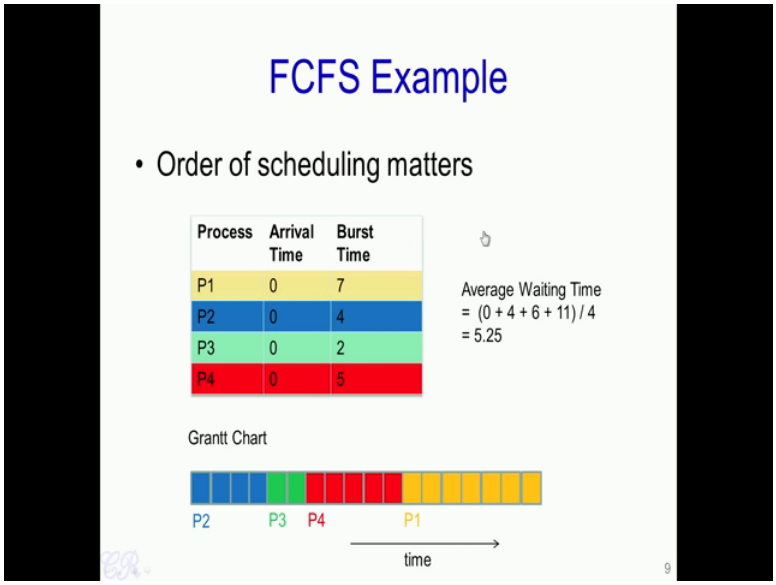
So, a Gantt chart is a horizontal bar chart developed as a production tool in 1917 by Henry L Gantt who was an American engineer and a Social scientist. So, essentially in a Gantt chart we have like several blocks over here and each block represents a cycles of execution (mentioned in above image). So, for instance, P 1 executes for 7 cycles, so it has likes 7 blocks – 1, 2, 3, 4, 5, 6, 7 (yellow blocks), P 2 then executes for 4 cycles, so it's given like 4 blocks (blue blocks). Then P 3 executes for 2 cycles, so it is given 2 blocks (green blocks) and finally, P 4 executes for 5 cycles, so it is given 5 blocks (red blocks).

So, we could compute the average waiting time for this particular case (mentioned in above FCFS example image). We see that process P 1 enters into the ready queue at the instant 0 and immediately gets to execute in the CPU. Thus, it does not have to wait at all (waiting time is 0). The second process P 2 arrives also at 0 that is also arrives at this point (instant 0) but gets to execute only after P 1 executes that is only after 7 cycles; thus, its wait time is for process P 2 is 7. Similarly, process P 3 which also enters in the 0th cycle gets to execute only after process P 1 and P 2 completes. In other words, it has to wait 11 cycles and the fourth process in a similar way needs to wait for 13 cycles. Therefore, the average waiting time in this particular case is 7.75 cycles ((0+7+11+13)/4 = 7.75).

Now, we can look at an other scheduling criteria, which is the average response time. So, average response time in this case is the same as the average waiting time. So the average response time is the time taken for a particular process to begin executing in the CPU minus the time it actually enters into the ready queue. So, for instance, P 2 enters into the ready queue at this instant, but begins to execute in the CPU only after 7 cycles. So, therefore, the response time for process P 2 is 7. Similarly, process P 3 has a response time of 11 because it has waited for 11 cycles to actually begin executing in the CPU

(mentioned in above image). So, on average, the average response time is 7.75 just like the average waiting time.

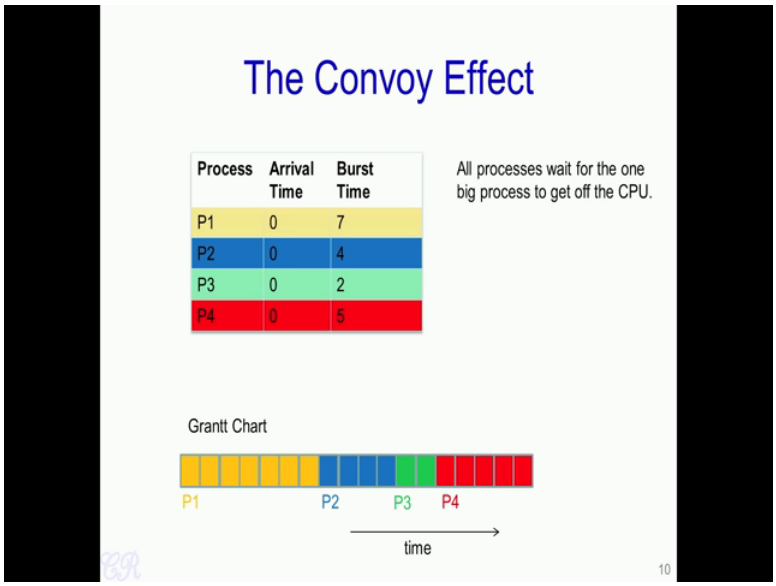
(Refer Slide Time: 14:07)



Now, one characteristic of the FCFS scheduling algorithm is that the order of scheduling matters. In the previous slide that is in this slide (slide mentioned at 14:26) we had assume that P 1 executes then P 2 executes then P 3 and then P 4, and we have got average waiting time and average response time of 7.75 cycles. Now, suppose we just change the ordering and let us say the ordering is now as follows that P 2 executes then P 3 executes then P 4 and finally P 1 (mentioned in above slide).

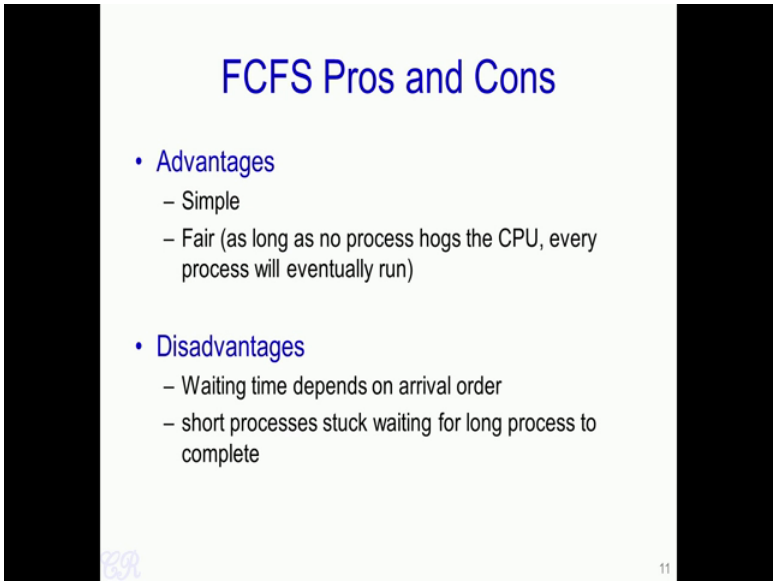
In such a case, if we compute the average waiting time we see that it gets reduce from 7.75 cycles to 5.25 cycles. Similarly, if you compute the average response time, you will see that the response time is also 5.25 cycles. In a similar way, you could compute the other criteria, which we are mentioned over here (mentioned in slide at 15:05) and you will be able to actually see the difference with the two ordering schemes.

(Refer Slide Time: 15:13)



Another characteristic of the FCFS scheduling algorithm is the Convoy effect. Essentially, in this particular case we see that all processes wait for one big process to get off the CPU. So, for instance, in this case, until or rather even though all processes enter the CPU at the same time, all processes have to wait for P 1 to complete only then they could be scheduled. So, if we have a large process over here for instance, we have a process P 1 instead of burst time of 7 takes a burst time of say 100, all other processes P 2, P 3 and P 4 would wait for 100 block cycles before they actually be able to get to execute in the CPU. So, this is a huge drawback of FCFS scheduling scheme.

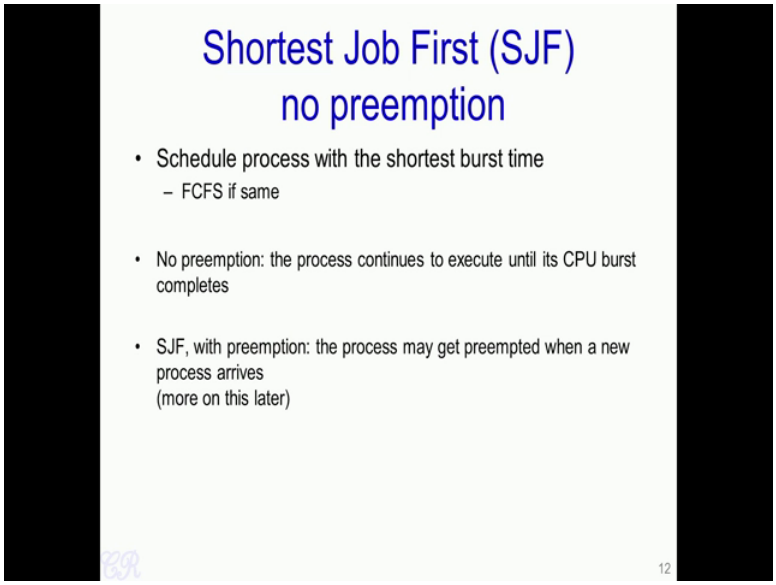
(Refer Slide Time: 16:10)



However, FCFS scheduling algorithm have several advantages. For the first thing is it is extremely simple. So the scheduling algorithm could complete very quickly and therefore, the time taken by the scheduling algorithm will be very less, and you would end up with very less context delays while changing the contexts. Another advantage of the FCFS scheduling algorithm is that it is fair. As long as no process hogs the CPU, every process will eventually run; or in other words, as long as every process terminates at some point, every other process in the ready queue will eventually get to execute in the CPU.

Now the drawback or the disadvantage of the FCFS schedulers as we have seen is that the waiting time depends on the arrival order. So, we have seen the example in the previous slides. Another disadvantage is that short processes are stuck in the ready queue waiting for long processes to complete; or rather, this is the convoy effect that we have just looked at (in previous slide).

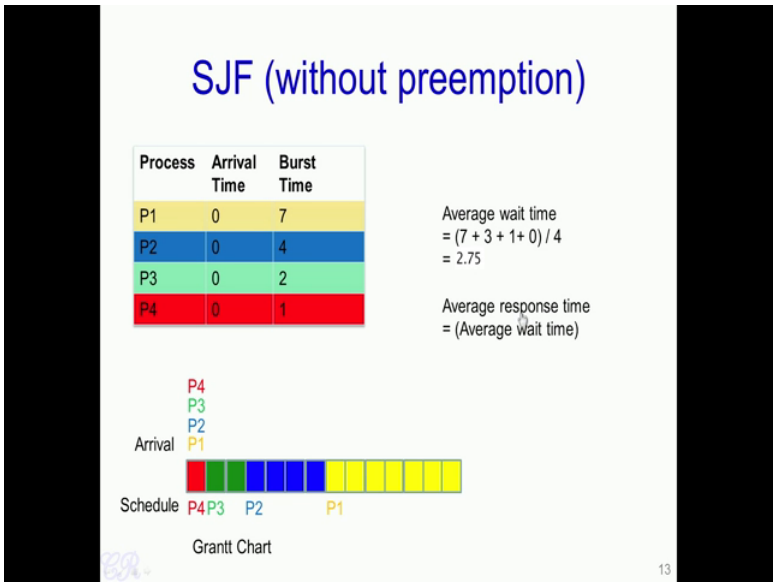
(Refer Slide Time: 17:19)



Now, let us look at another scheduling algorithm know as the Shortest Job First scheduling algorithm. In this particular scheduling algorithm, the job or the process with the shortest CPU burst time is scheduled before the others. Now if you have more than one process with the same CPU burst time then standard FCFS scheduling is used. There are two variants of the shortest job first scheduling algorithm.

The first is the no preemption variant, while the second one is the shortest job first with preemption. Now in the SJF with no preemption, the process will continue to execute in the CPU until its CPU burst completes. In the second variant with preemption, it may be possible that the process which may get preempted, when a new process arrives into the ready queue. We will see more on this later, but first we will start with the shortest job first variant with no preemption.

(Refer Slide Time: 18:21)

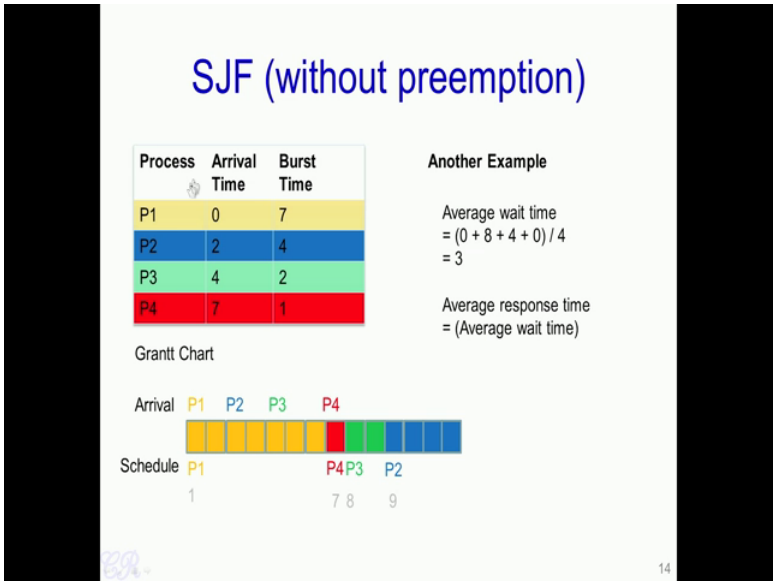


So let us take the same example that we have seen in the previous case, we had their four processes P 1 to P 4 and all of them are arriving at the instant 0, that is at instant 0 these four processes P 1 to P 4 get into the ready queue. And each of these processes have a different CPU burst time that is 7, 4, 2 and 1 respectively (mentioned in above image). Now in the first instant, the CPU scheduler will look at the various burst times and find the one which is minimum.

So, in this case (mentioned in above image) we see that P 4 has the minimum CPU burst time (burst time 1), so that were scheduled first. So, first the process P 4 gets scheduled until it completes, in this particular case it completes in 1 cycle. Then among the remaining three, we see that P 3 has the lowest CPU burst time. So, process P 3 gets scheduled and executes till it completes its burst (burst time 2). Then P 2 gets scheduled because it has a burst time of 4, while P 1 has a higher burst time of 7. And finally, P 1

gets to execute till completion. Now, the average wait time, if we compute this is 2.75, while the average response time is also 2.75 as in the wait time case.

(Refer Slide Time: 19:43)



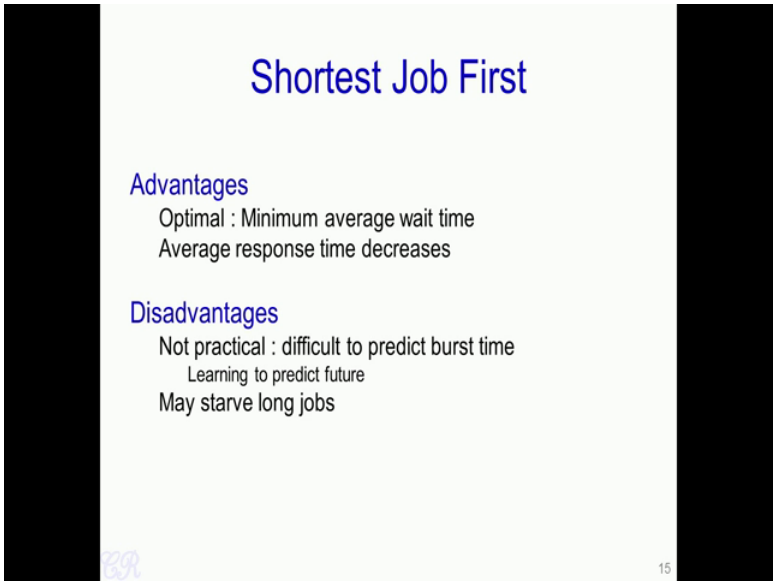
Now let us look at another example of shortest job first without preemption (mentioned in above image). So, we will take the same four processes P 1 to P 4 and each of these processes have the same burst time as before that is 7, 4, 2 and 1 respectively. However, they arrive at different instants that is the moment the instant in which they enter into the ready queue would be different. So, P 1 enters at the 0th instant, P 2 in the 2nd instant, P 3 at the 4th instant and P 4 in the 7th instant. Now, this is a slight modification in the Gantt chart, where in addition to showing which process is executing in the CPU, it also shows the order in which processes arrive. It shows that the P 1 arrives first, then P 2 in the 2nd instant, then P 3 and finally P 4 in the 7th instant (refer above slide image).

Now, when the scheduler begins to execute at this particular instant (starting point in Gantt chart), the only process that has arrived at this particular point is P 1 therefore, it schedules P 1 to execute. So, P 1 executes for its entire burst that is of 7 cycles, and then at this particular cycle (after 7 blocks), the scheduler enters again or a scheduler executes again and this time it has got three processes to choose from all P 2, P 3 as well as P 4 have arrived in the ready queue.

And out of them P 4 has the shortest burst time therefore, it is chosen for execution. Therefore, P 4 executes in the CPU, and then P 3 because P 3 has a shorter burst time

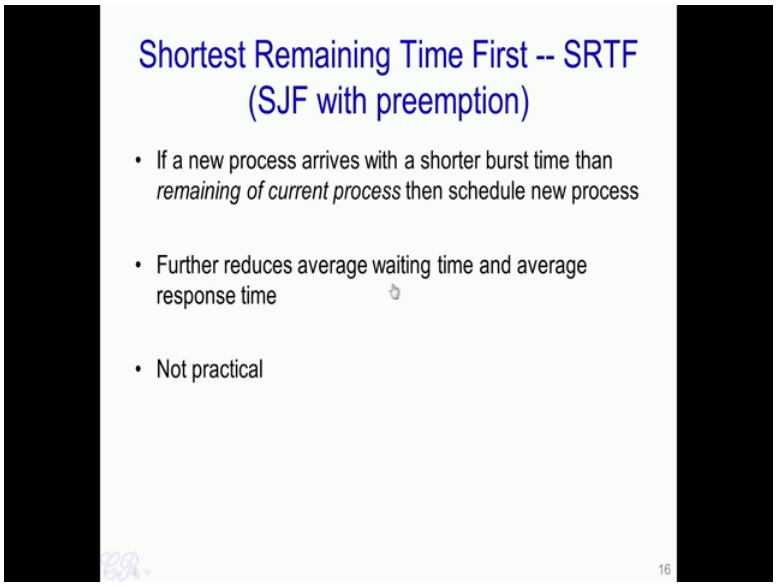
than P 2 and finally, P 2 gets executed. So, if we compute the average wait time, we see that it is 3 cycles (mentioned in above image).

(Refer Slide Time: 21:37)



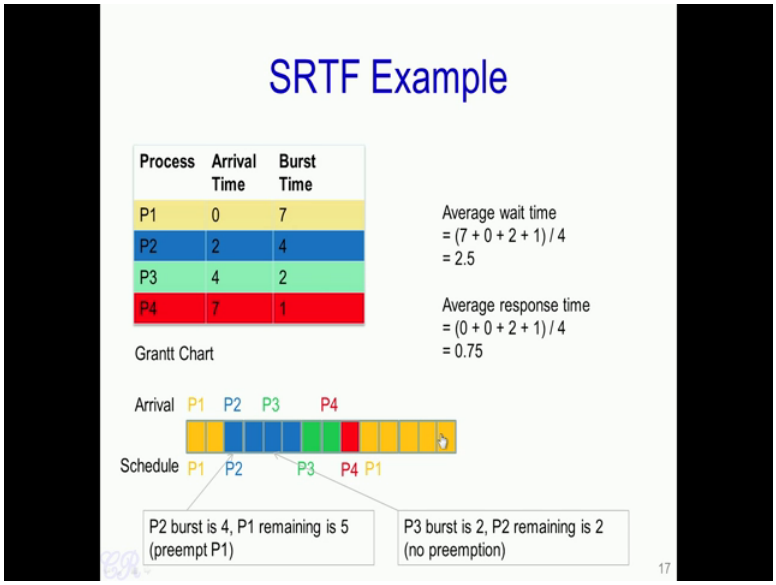
So the advantages of the shortest job first scheduling algorithm is that is Optimal. It will always give you the minimum average waiting time. And as a result of this, the average response time also decreases. The main disadvantages of the SJF scheduling algorithm is that it is not practical; essentially, it is very difficult to predict what the burst time would be. So another drawback of the SJF scheduling algorithm is that some jobs may gets starved; essentially if you have a process which has an extremely long CPU burst time, then it may never get a chance to actually execute in the CPU.

(Refer Slide Time: 22:19)



Now, we will look at the shortest job first scheduling algorithm (SJF) with preemption. So, this is also the shortest remaining time first. So the basic idea in this algorithm is that if a new process arrives into the ready queue, and this process has a shorter burst time than the remaining of the current process, then there is a context switch and the new process gets scheduled into the CPU. This further reduces the average waiting time as well as the average response time. However, as in the previous case that is the shortest job first with no preemption here also it is not practical. So let us understand more on this with an example.

(Refer Slide Time: 23:02)



So let us take the same example of 4 processes with burst time 7, 4, 2 and 1; and arrival times at 0, 2, 4 and 7, will develop the Gantt chart as the time processes. So, at the instant 0, the only process which is present is P 1; and therefore, the scheduler has no choice, but to schedule P 1 onto the CPU. Thus P 1 would execute for 2 clock cycles.

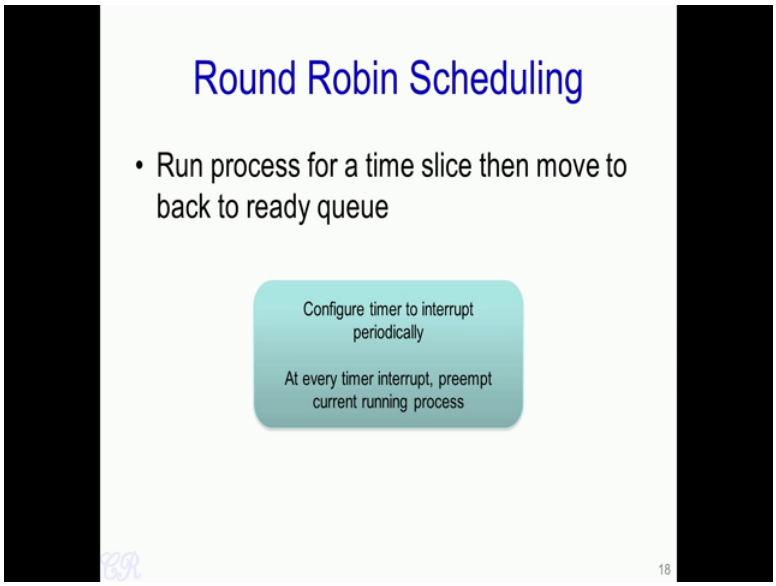
Now after the 2nd clock cycle the process P 2 has entered into the ready queue. Now you see that P 2 has a burst of 4 cycles. However, P 1 has a remaining burst of 5 cycles. So, what we mean by this is that out of the CPU burst time of 7 cycles, P 1 has completed 2 cycles. So, what remain is 5. Now the scheduler will see that P 1 has 5 cycles, which is greater than P 2, which has burst of 4 cycles. Therefore, it will do a context switch and schedule P 2 to run. So, P 2 will run for 2 clock cycles and then P 3 arrives at the 4th clock instant (mentioned in above image).

So, at this particular instant, the scheduler will find out that P 3 has a burst time of 2 cycles, while P 2 has a remaining burst time of 2 cycles; we achieve 2 because out of the 4 cycle burst time for P 2, it has completed 2. So, what remains is 2 more cycles. Since P 2 the old process which is running on the CPU has a 2 cycle remaining burst time, and P 3 the new process also has 2 cycles burst time therefore, there is no preemption and P 2 will continue to execute (refer above image).

Now after P 2 completes, in this particular case, P 3 executes for 2 cycles, and then P 4 enters now after which if you will verify will not cause any preemption. So, after P 3 completes, the scheduler decides to run either P 4 or P 1. So, we see that P 4 has a burst cycle of 1 while P 1 has a remaining burst cycle of 5, therefore the scheduler will decide to choose P 4 over P 1. So, P 4 runs on the CPU and after it completes P 1 will executes for its remaining burst time (refer above image Gantt chart).

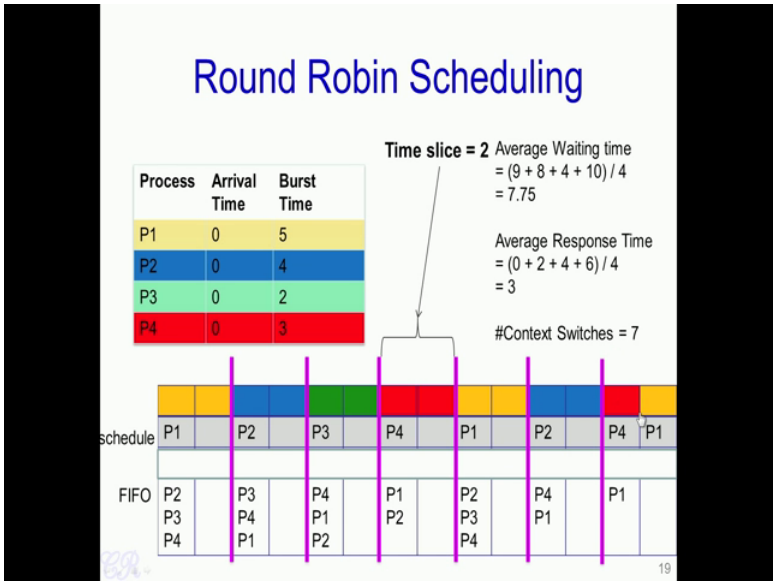
So, if you compute the average wait time, you see that it reduces to 2.5, while the average response time reduces considerably to 0.75 (mentioned in above image). However, as we mentioned before just like the shortest job first, this scheduling algorithm is also not feasible to be implemented in practice, because it is very difficult to actually identify what the burst time of a process is and even more difficult to identify what the remaining burst time of the process would be.

(Refer Slide Time: 26:08)



Now we will look at another scheduling algorithm known as the Round Robin Scheduling algorithm. So, essentially with the round robin scheduling algorithm, a process runs for a time slice that is a process executes for a time slice and when the time slice completes it is moved on to the ready queue. So, in order to achieve this round robin scheduling algorithm which is also a preemptive scheduling algorithm; now in order to achieve the round robin scheduling, we need to configure the timer in the system to interrupt the CPU periodically. At every timer interrupt, the kernel would preempt the current process and choose another process to execute in the CPU.

(Refer Slide Time: 26:50)



Let us discuss the round robin scheduling algorithm with an example. So, one difference with respect to the other scheduling algorithms that we have seen so far is the notion of time slice. So, this is the Gantt chart (refer above image). So, especially see that periodically in this case, with a period equal to 2 that is we have keeping a time slice equal to 2, there is a timer interrupt that occurs and the timer interrupt would result in the scheduler being run and potentially another process being scheduled into the CPU. So, a data structure which is very useful in implementing the round robin scheduling algorithm is the FIFO.

This particular FIFO stores the processes that need to be executed next into the CPU. For example, in this particular case (mentioned in above slide image), P 2 is at the top of the FIFO, so it is a next process which gets executed in the FIFO. So, P 2 gets executed over here (next to P1 in Gantt chart). So, for example, we will still consider the 4 processes as we have done before that is P 1 to P 4 and we will assume that all of them arrive at the instant 0 and go into the FIFO of the ready queue and they have burst times of 5, 4, 2 and 3 respectively.

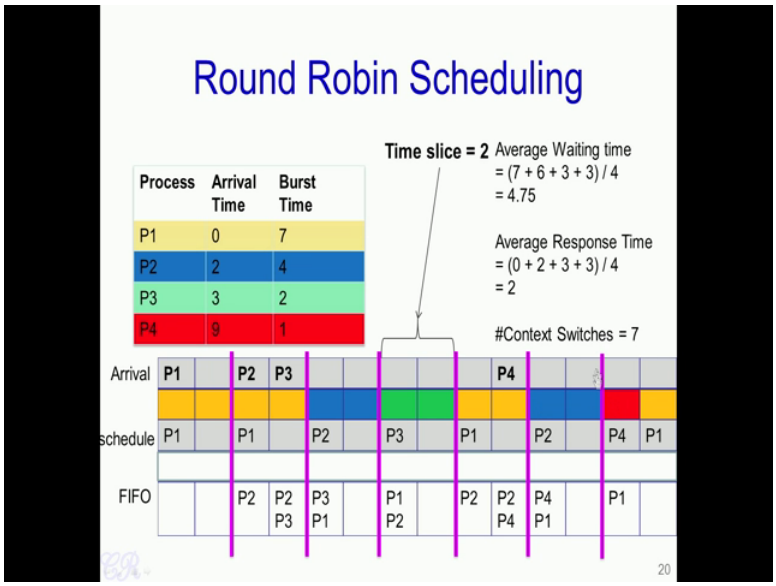
So let us say for discussion that the scheduler starts off with this particular order P 1, P 2, P 3 and P 4 and it first chooses to execute P 1. So, P 1 executes for 2 cycles and then there is a interrupt which occurs leading to a context switch and then the top of the FIFO in this particular case (mentioned in above image) P 2 get scheduled into the CPU, while P 1 gets pushed into the FIFO. So, P 2 then executes for 2 cycles until the next timer interrupt in which case the time slice of 2 cycles completes and then it gets pushed into the FIFO. So, P 2 now is at the bottom of the FIFO, while P 3 which is at the top of the FIFO get scheduled to run. So, in this way, every two cycles a new process may get scheduled into the CPU and execute.

So, if we compute the average waiting time. So, in this particular case, we will see that the average waiting time and the average response time is different. So, what is the average waiting time for P 1? So, the P 1 executes 2 cycles here (check first P1 block in Gantt chart), 2 cycles here (second P1 block) and completes execution over here (last P1 block). So, it waits in the ready queue in the remaining of the cycles. So the number of cycles it waits is 1, 2, 3, 4, 5, 6, 7, 8, 9 (P2, P3, P4 block then P2,P4 block). So, P 1 waits for 9 cycles; now P 2 waits for 8 cycles – 1, 2, 3, 4, 5, 6, 7, 8; P 3 for 4 cycles and P 4 for 10 cycles. So the average waiting time is 7.75 cycles.

Now, to compute the average response time as we have defined it before, the response time is the time the process enters into the ready queue to the time it begins to execute in the CPU, that latency would be the response time. So, P 1 for instance has a response time of 0 because it enters into the ready queue or enters into the FIFO and gets executed immediately. P 2 on the other hand enters in the 0th cycle, but gets to execute only after 2 cycles, so it has a response time of 2 (refer round robin scheduling slide).

Similarly, P 3 enters at 0 but executes only at this point. So, at rather at this instant (After P1 and P2 in Gantt chart) therefore, it has a response time of 4 and P 4 has a response time of 6 therefore, the average response time is 3. Now the number of context switches that occur is 7. So, in this case, 1, 2, 3, 4, 5, 6 and a context switch occurs over here (last block P4 and P1) because the process P 4 is existing out and P 1 gets continues to execute. So the numbers of context switches over here are 7 that is 1, 2, 3, 4, 5, 6 and the 7th context switch occurs over here (last block) then P 4 exits and P 1 gets switched into the CPU.

(Refer Slide Time: 31:42)



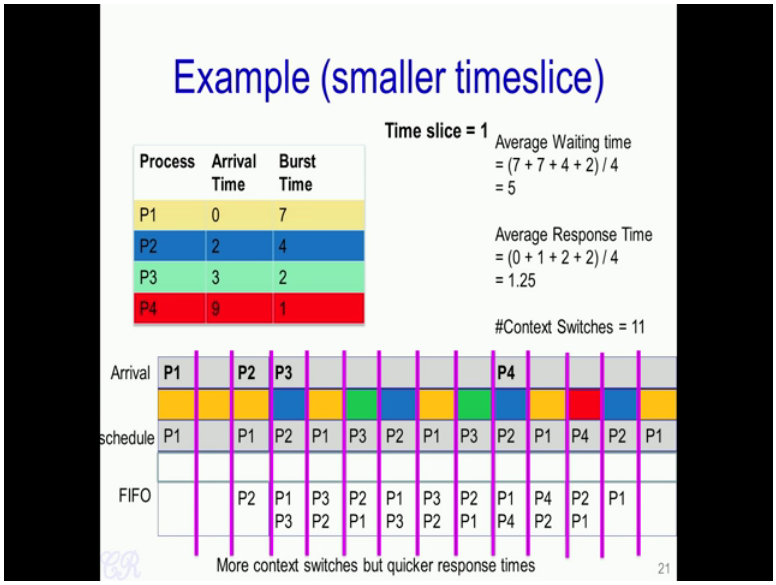
Now let us take another more complex example of round robin scheduling, where we also have arrival times which are not the same. So, P 1 is arriving at the 0th instant; P 2 at the 2nd, P 3 and P 4 at the 3rd and 9th respectively. So, we can similarly draw the Gantt chart and the states of the FIFO for this case. So to start with, in the 0 instant, the only process which has arrived is P 1 and therefore, P 1 executes for 2 cycles. And at this

particular point (first pink line in above slide Gantt chart), when the timer interrupt occurs, no other process is present as yet, therefore P 1 will continue to execute for another 2 cycles for another time slice.

However, in this time slice, we have two processes which have entered into the ready queue; these are the process P 2 and P 3. So, P 2 arrives at this interval (3rd cycle) while P 3 arrives at this interval (4th cycle) and they get added into the FIFO. So, at the second time slice completion, there is a context switch and P 2 gets scheduled into the CPU to execute while P 1 which was executing will then go into the FIFO. So, P 2 executes for 2 cycles, then P 3 executes for 2 cycles, then P 1 executes for 2 cycles and at that time P 4 has arrived and gets added into the FIFO.

Now, we have three processes P 1, P 2 and P 4 and these get schedule to run for a period of time. So the average waiting time in this case is 4.75, while the average response time is 2. So, how is the average waiting time 4.75, it means that process P 1 has waited for 7 cycles so, before it completes. So that is 1, 2, 3, 4, 5, 6 and 7. While process P 2 has waited for 6 cycles, so it is 1, 2, 3, 4, 5, 6; process 3 has waited for 3 cycles, and process 4 has waited for 3 cycles (refer above slide). The average response time can be verified to be equal to 2 and the number of context switches was as before 7.

(Refer Slide Time: 34:15)

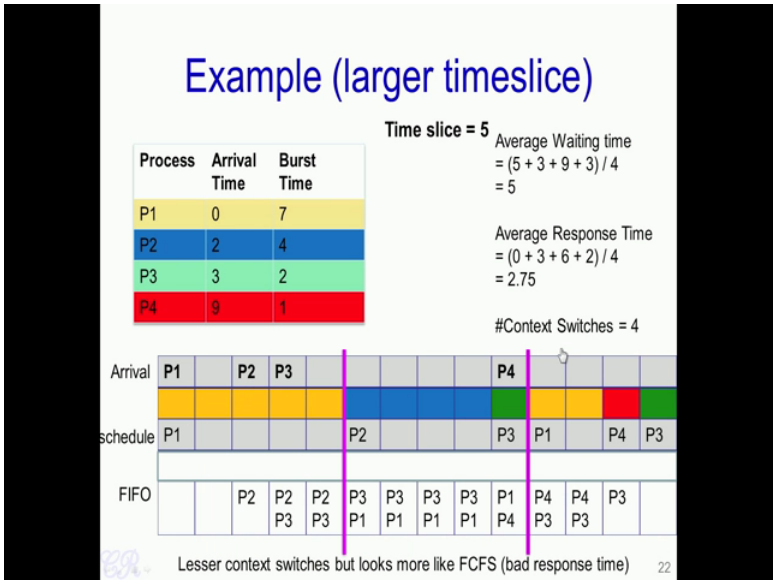


Now let us take the same example as we have done before, but with a time slice of one that is we have reduced the time slice from 2 and we have made it 1. So, we see that in

every instant we have a timer interrupt and potentially context switch that can occur. If we compute the average waiting time and average response time for this particular case we see that the average response time in particular has reduced. Why has this occurred is because once a process has entered into the queue it has to wait for lesser cycles before it gets scheduled into the CPU.

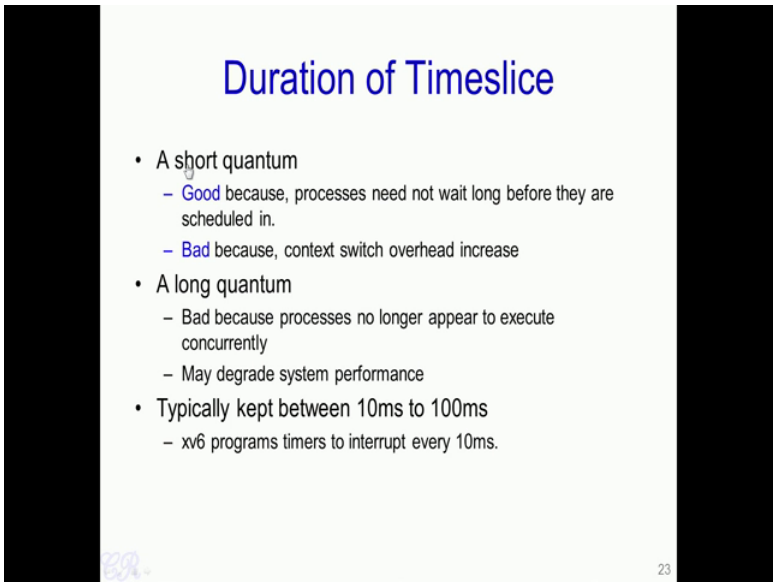
On the other hand, the number of context switches has increased from 7 to 11. Since we are having timer interrupts which are more frequent therefore, there is more likely that a context switch will occur.

(Refer Slide Time: 35:10)



Now, if we take the same example, but with the time slice of 5 instead of 1, and if we compute the average waiting time and average response time (refer above image), we see that the response time increases considerably to 2.75 while the number of context switches reduces quite a bit to 4. On the other hand, we see that the scheduling begins to behave more and more like the first come first serve. The response time is bad because due to the large time slice the scheduling behaves more and more like the first come first serve which we know has a bad response time. So, from all this examples that we have seen so far we seen we can conclude that the duration of a time slice is very critical. So, it effects both the response time as well as the number of context switches.

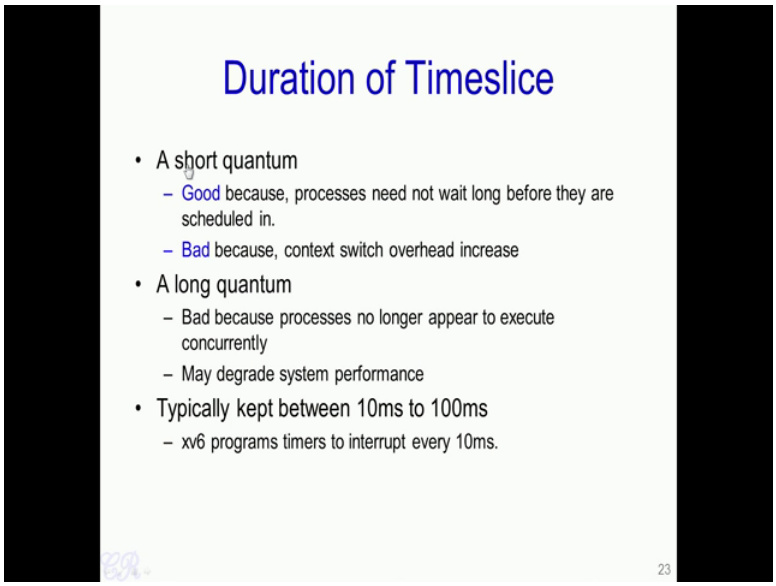
(Refer Slide Time: 36:07)



So, essentially, if we have a time slice which is of a very short quantum, the advantage is that processes need not wait too long in the ready queue before they get schedule into the CPU. Essentially this means that the response time of the process would be very good or it would have a less response time.

On the other hand, having a short time slice is bad, because we would have very frequent context switches. And as we seen before context switches could have considerable over heads. Therefore, it degrades the performance of the system. A long time slice or a long quantum has a drawback that processes no longer appear to execute concurrently, it appears more like a first come first serve type of scheduling algorithm and so this again in turn may degrade system performance. So, typically in a modern day operating systems the time slice duration is kept anything from 10 milliseconds to 100 milliseconds. So, xv6 programs, programs timers to interrupt every 10 milliseconds.

(Refer Slide Time: 37:17)



So the advantage of the round robin scheduling algorithm is as follows. The algorithm is fair because each process gets a fair chance to run on the CPU. The average wait time is low especially when the burst times vary and the response time is very good. On the other hand, the drawbacks of the round robin scheduling algorithm are as follows; there is an increase number of context switching that occurs and as we have seen before context switching has considerable over heads. And the second drawback is that the average wait time is high especially when the burst times have equal lengths.

(Refer Slide Time: 37:56)

The xv6 scheduling policy is a variant of the round robin scheduling policy; the source code is shown over here (refer above slide). So, essentially what the xv6 scheduler does is that it parses through the ptable array. So, we have seen ptable before, which is an array of procs and the scheduler parses through this particular array and finds the next process that is runnable and invokes the switch. So, this is where you invoke the context switch (switchkvm(); in above slide code). So, every time the scheduler executes, the next process in this array (mentioned in circle in above slide) that is runnable gets scheduled to execute.

So next, we will see scheduling algorithms which are based on priority. So, these are the priority based scheduling algorithms.

Thank you.

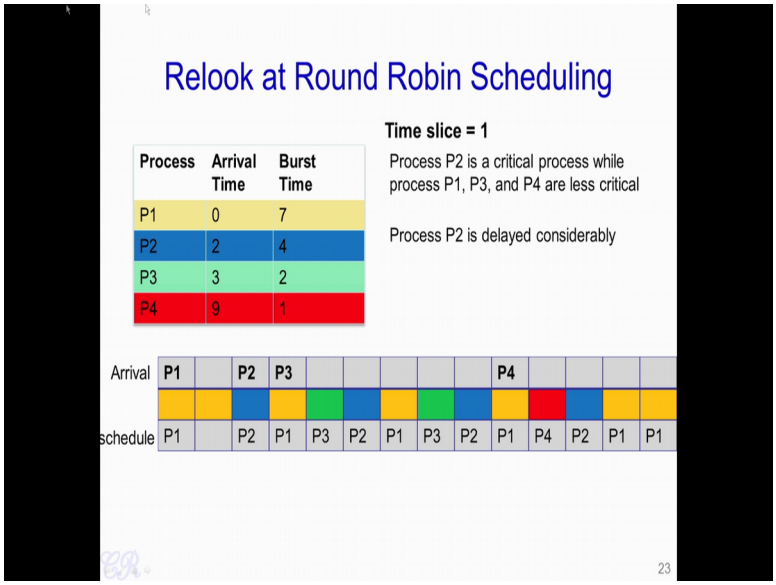
Introduction to Operating Systems
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week – 05
Lecture – 19
Priority Based Scheduling Algorithms

So far we had seen about some Scheduling Algorithms like, First Come First Serve and the preemptive scheduling algorithms like the Round Robin.

So in this particular video, we will look at a class of algorithms for scheduling which are known as the Priority Based Scheduling Algorithms. So we will start this lecture with a motivating example.

(Refer Slide Time: 00:40)



So, let us relook at this particular example (mentioned in above slide) which we have taken in the last video. So we had seen this Round Robin Scheduling Algorithm where we took four processes P1 to P4, and these processes arrived at different times 0, 2, 3 and 9, and they had different burst times like 7 cycles, 4, 2, and 1 cycles respectively.

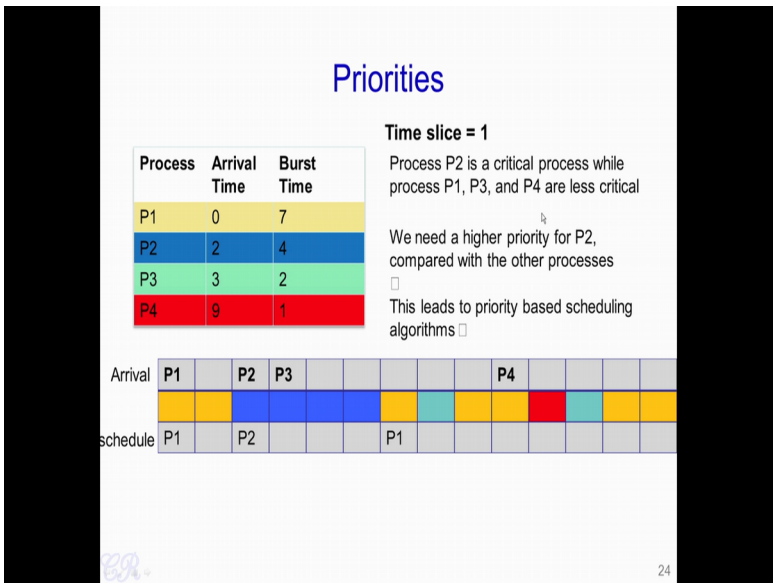
Now we will add a bit above the processes which are involved. Let us assume that P2 is a critical process, while P1, P3 and P4 are less critical. So, what is a critical process? So let us take an example of an operating system that runs in a micro processor present in a car,

and whenever the breaks are pressed process P2 will execute and it will do several task related to breaking of the car, while process P1, P3 and P4 may be less critical tasks. With respect to the car again for instance process P1, P3 and P4 may be the music player, so one of the process may be the music player while another one could be for instance controlling the AC or the heater and so on.

So, lets us look at this Gantt chart again, we see that P2 arrives in the second cycle and it takes considerable amount of cycles to complete executing. The main reason for this is that this is a round robin scheduling scheme and other processes have a fair share of the CPU. So, P2 first executes here (1st blue block in Gantt chart) and then there is a P1 and P3 which executes, and then P2 executes again and again (2nd and 3rd blue block) and finally completes executing its burst at this particular time (4th blue block).

Now, you will see that there is an obvious problem with this particular scheme. Since, P2 is a critical process such as controlling the breaking of the car it has taken really long time for it to complete its execution, and this could lead to catastrophic accidents.

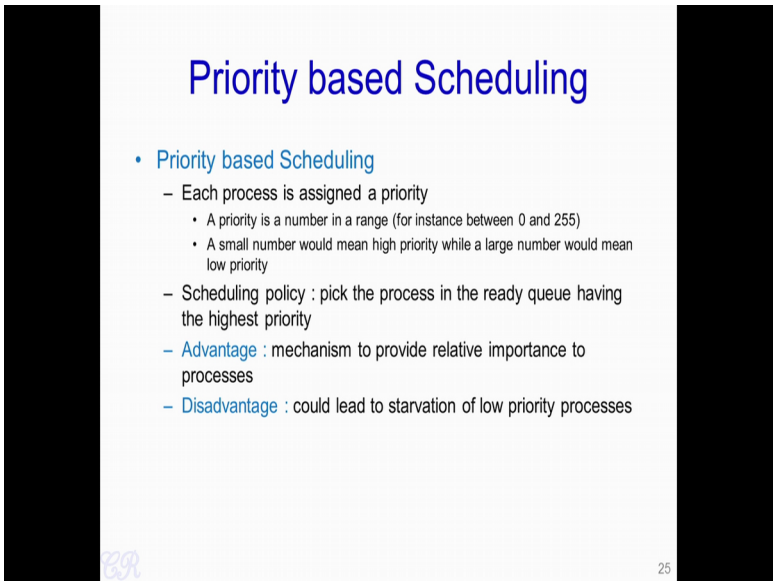
(Refer Slide Time: 02:49)



So what one would have expected is something like this. Whenever a critical process such as P2 arrives in the ready queue, it should get a priority and it should be able to execute continuously, irrespective of the fact that there are other processes with the lower priority present in the queue.

So this is what we would like to have in a more realistic situation. So scheduling algorithms which take care of such priorities among processes are known as Priority Based Scheduling Algorithms, and this is what we are going to study in this particular video lecture.

(Refer Slide Time: 03:28)



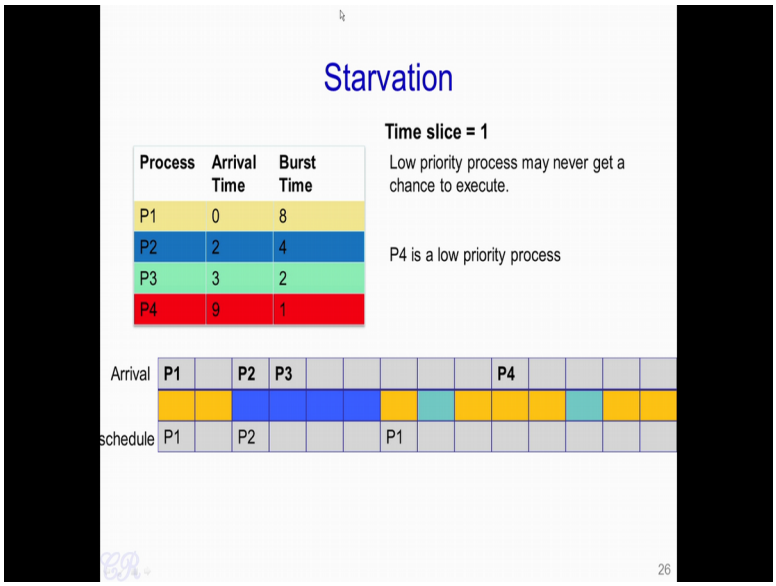
In a priority based scheduling algorithm each process is assigned a priority number. A priority number is nothing but a number within a particular range, so this range is predefined by the operating system. For instance, the priority range is between 0 and 255. Therefore, every process that is executed would be assigned a priority number between 0 and 255. A number which is small, that is a priority number which is small that is close to 0 would mean that the process is a high priority process or a high priority task. On the other hand, a large priority number would mean that the process has a low priority.

So this is just a nomenclature which Linux follows, other operating systems may have a different approach. So it would keep for instance a higher number indicating a high priority process while a lower priority number indicating a low priority process, but for this lecture we will goes with the Linux nomenclature where a small priority number means a high priority process and vice versa.

Now, in a priority based scheduling algorithm even though there may be several tasks present in the ready queue, the scheduling algorithm will pick the task with the highest priority. The advantage of the priority based scheduling algorithms is quite easy to see. Essentially in an operating system which supports a priority based scheduling algorithm, tasks can be given relative importance or processes could be given relative importance. High priority processes would get a higher priority to actually execute in the CPU compare to low priority processes.

On the other hand, the main drawback of having a priority based scheduling algorithm is that it could lead to what is known as starvation. So essentially, a starved process is one which is not able to execute at all in the CPU, although it is in the ready queue for a long time. It is quite easy to see that starvation mainly occurs for low priority processes.

(Refer Slide Time: 05:39)

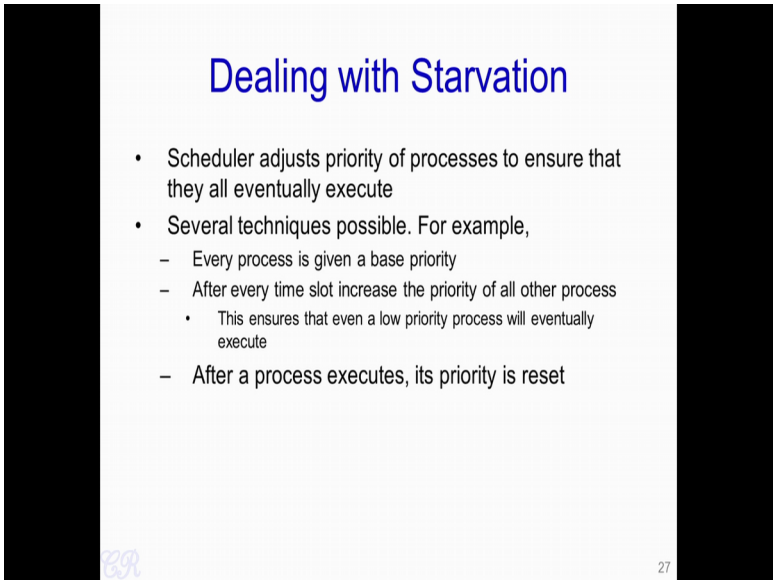


SO let us see how starvation occurs with an example. So we will take the same four processes P1 to P4 arriving at different times 0, 2, 3 and 9, and having different burst times 8, 4, 2 and 1. Also, let us assume that the priority of P1, P2 and P3 is higher than P4 or in other words P4 is the lowest priority process. We will also make this assumption that every 15 cycles this particular sequence repeats, that is after the 15th cycle of the processor, P1 again arrives, P2 again arrives, P3 and similarly P4 arrives, so this sequence of processes arriving into the ready state continuous infinitely.

So what we see, if you look at the Gantt chart is that depending on the priority P1, P2 and P3 are scheduled. However, the low priority task never gets scheduled within the 15 cycles. Additionally, since P1, P2, P3 and P4 arrive again into the ready queue after 15 cycles. So, again P1, P2 and P3 get scheduled while P4 never get scheduled. So as a result what we see is that, this lowest priority process gets starved.

Essentially, even though it is in the ready queue and just waiting for the CPU to be allocated to it, it never gets a chance to execute in the CPU because there are always higher priority processes which are arriving in the ready queue.

(Refer Slide Time: 07:28)



So let us see how priority based scheduling algorithms deals with starvation. So essentially, in operating systems which have a priority based scheduling algorithm, the scheduler would dynamically adjust the priority of a process to ensure that all processes would eventually execute in the CPU. What this mean is that, in our example the priority of process P4 would gradually increase over a period of time until the point that it has a priority which is greater than or equal to the priority of the processes P1, P2 and P3, at that time process P4 will get a chance to execute in the CPU.

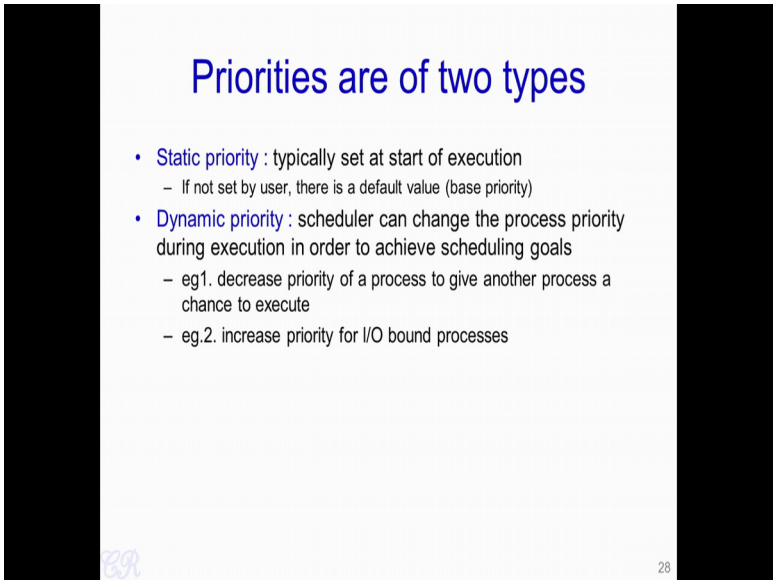
Thus, we see by elevating process P4 over time the starvation that would otherwise have occurred, since P4 was a low priority process is eliminated. So there are several techniques that priority based scheduling algorithm could deal with starvation. So we

give one particular example here in which an operating system with a priority based scheduling algorithm could ensure that every process, even the low priority process will eventually execute in the CPU. So, let us say in this particular operating system when a process starts it is given a base priority. Now, this base priority will differ from process to process, a high priority process will be given a high priority number, while a low priority process will be given a low priority number.

Now, after every time slot or time slice the priority of the process is increased by a fixed value. So, essentially all processes in the ready queue have their priority increased by a fixed value except for the process that is currently being executed in the CPU. So what you would see that, over a period of time all processes in the priority queue would gradually have their priority increased. Now even a low priority process would have its priority increased up to the point that its priority is high enough so that it is scheduled the CPU.

So, after it is scheduled into the CPU and executes in the CPU, at the end of its time slice its priority is reset back to its base priority. Thus the process which starts off with the base priority of 250 will gradually have its priority increased, so it is a point that is begins to execute and after it's executes its priority is reset to 250. Thus, starvation would be avoided.

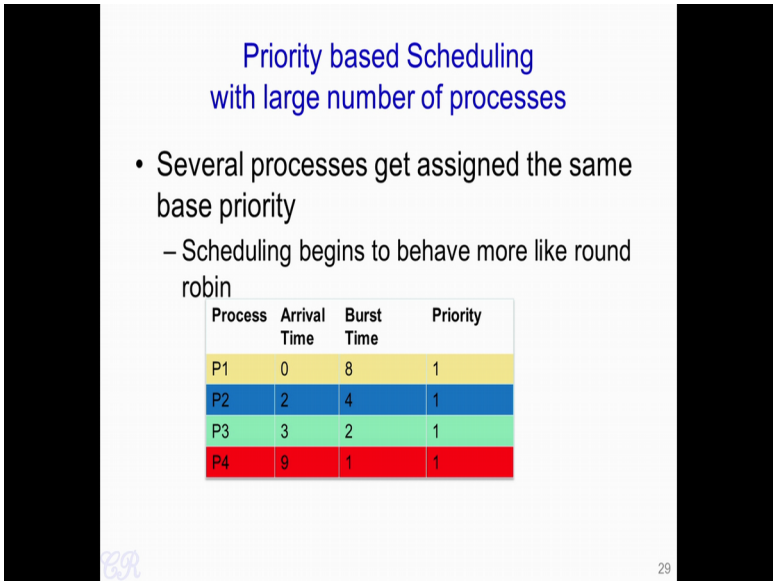
(Refer Slide Time: 10:20)



So based on this, there are two types of priorities present for a process. There is the static priority which is typically set at the start of execution, so it can be set by user who is starting that application, and by chance if the user does not set that particular priority then a default value is taken.

On the other hand, the second type of priority is known as the Dynamic Priority, where the scheduler can change the process priority during the execution, in order to achieve some scheduling goals. For example, the scheduler could decide to decrease the priority of a process in order to give another process a chance to execute. Another example as you have seen before is to increase the priority of I/O bound processes. Since, I/O bound processes would typically require a faster response time.

(Refer Slide Time: 11:19)

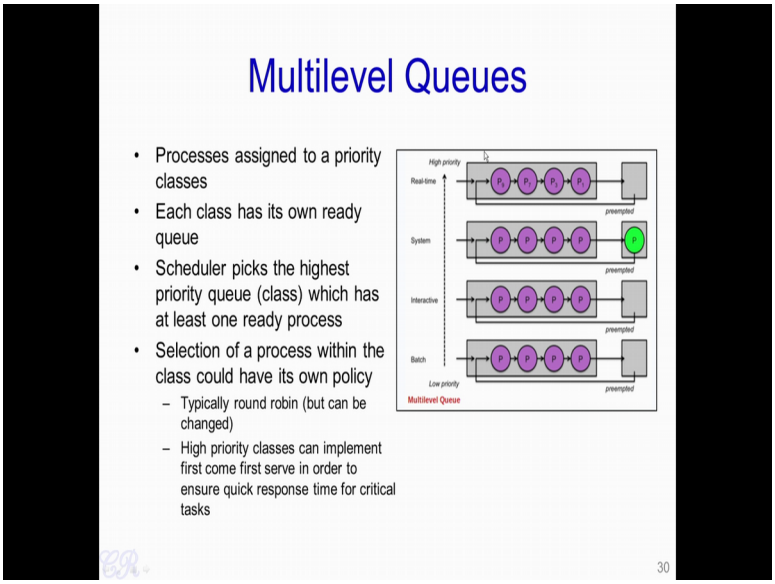


One disadvantage of having a range of numbers from which a process obtains its priority, is the case when there are a large number of processes present in the system. In such a case, there may be several processes which are executing with exactly the same priority. For instance, let us say that a priority range is from 0 to 255 and there are may be over a ten thousand processes running in the system. In such a case each priority number there may be several processes executing with that priority number.

So this is depicted in this particular figure (mentioned in above slide image), where we have four processes arriving at different times and having different burst times, but all of

them have the same priority. So in such a case, the scheduling would begin to behave like non priority based scheduling algorithm. For instance it would begin to behave like a round robin scheduling algorithm.

(Refer Slide Time: 12:22)



So in order to handle this, what modern operating systems have is Multilevel Queues (mentioned in above slide) that is instead of assigning a single priority number to each process; processes are assigned a priority class. Now there could be several priority classes within the operating system and each of these priority classes could range from a high priority to a low priority. For instance, there could be a real time priority class, a system priority class and interactive priority class or batch priority class and so on.

All real time processes would be present in this priority class. So each of these priority classes has a ready queue and when a real time process is ready to execute it gets added on to this particular ready queue, that is it gets added on to the ready queue present in the real time priority class.

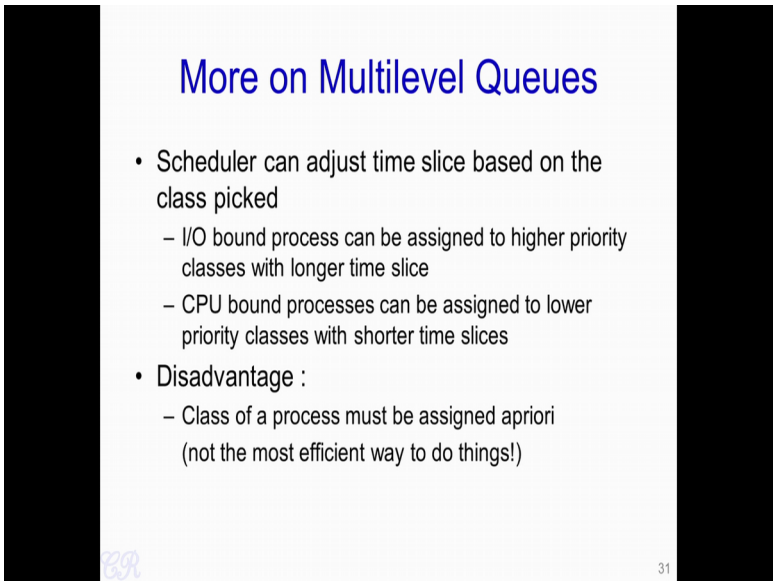
Similarly, if an interactive process wants to execute and it is ready to execute it gets added to this ready queue present in the interactive priority class. Now, at the end of a time slice when the scheduler executes, the scheduler picks the highest priority class which has at least one ready process. So for instance, the scheduler will scan through these priority classes starting from the highest priority to the least priority, and it will

select that particular priority class which has at least one process in the ready queue. Now, if there is exactly one process in the ready queue then we do not have a problem, now that process is going to execute in the CPU.

However, if we have multiple processes in the ready queue corresponding to the ready queue chosen by the scheduler, then a second scheduling algorithm would be used to pick or to choose from this set of processes. For instance, let us say the scheduler has decided to choose the interactive priority class, because there were no another processes ready in the real time and the system priority classes.

Further, let us assume that there are 4 processes present in the interactive class. Now the second scheduling algorithm such as a Round Robin or a FIFO scheduling algorithm will then execute to determine which of these four processes will be assigned the CPU. Thus, in multilevel queues there are two scheduling algorithms involved. The first is a priority based scheduling algorithm, which chooses one of the priority classes, while the second is a non priority based algorithm, which chooses among the processes in that priority class. The second scheduling algorithm would typically the round robin, while for high priority processes the second scheduling algorithm is typically something like first come first serve and so on.

(Refer Slide Time: 15:36)

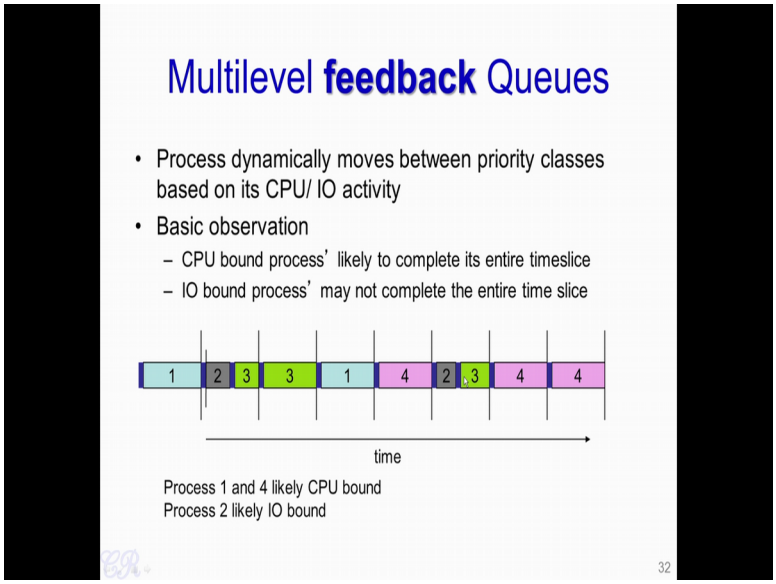


Further, in schedulers that support multilevel queues, the scheduler could adjust the time slice based on the priority class that is picked. So, I/O bound processes are typically assigned a higher priority class and a longer time slice. This ensures that the I/O bound processes would be quickly serviced by the CPU as soon as possible. Also the longer time slice will ensure that the burst of the I/O bound process completes before the time slice completes.

CPU bound processes on the other hand are assigned a lower priority class and given shorter time slices. The main drawback of having a multilevel scheduling queue based scheduling algorithm, is that the class of the process must be assigned apriori that is we would require to know whether a process is a CPU bound process or an I/O bound process. Now this is not an easy thing to do; first it is very difficult to identify whether a process is indeed an I/O bound process or a CPU bound process. Second, a single process may act like an I/O bound process at one time, but at another time it can act like a CPU bound process.

That is, at one time it may have short CPU burst and long burst of idle time, while at other times it may have long CPU burst and short I/O burst like a CPU bound process. Therefore, deciding apriori whether a particular process belongs to I/O bound process and it should be given a higher priority is difficult.

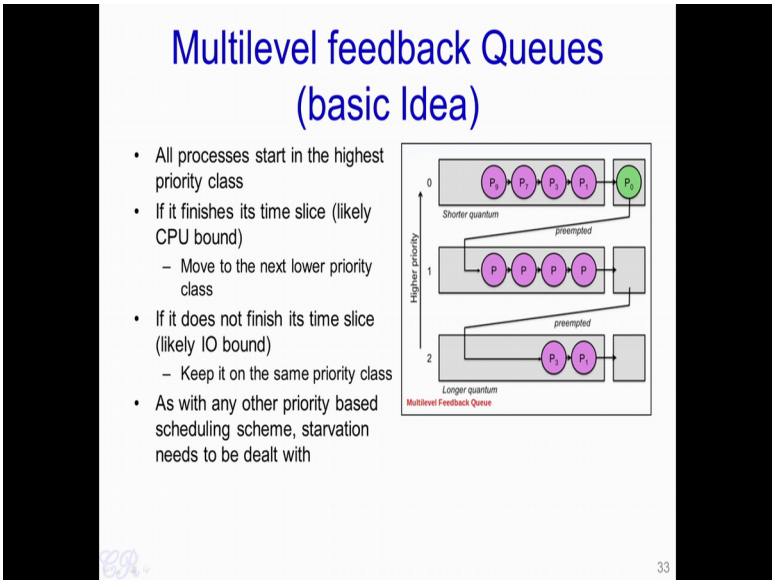
(Refer Slide Time: 17:24)



So, one way to mitigate this particular limitation is to use schedulers which use multilevel feedback queues. In such schedulers processes are dynamically moved between priority classes depending on its CPU, I/O activity. So these particular schedulers work on this basic observation that CPU bound processes are likely to compute its entire time slice. This is because a CPU bound process will have a long burst of CPU and therefore it is quite likely that it would use up the entire time slice.

On the other hand I/O bound processes have a very short CPU burst time and therefore it is not likely to complete its entire time slice. So if you take this particular example (mentioned in above slide), we can say that process 1 and 4 have completed their time slice; therefore these processes are likely to be CPU bound. On the other hand, process 2 has very short CPU burst; therefore process 2 is likely to be an I/O bound process rather than a CPU bound process.

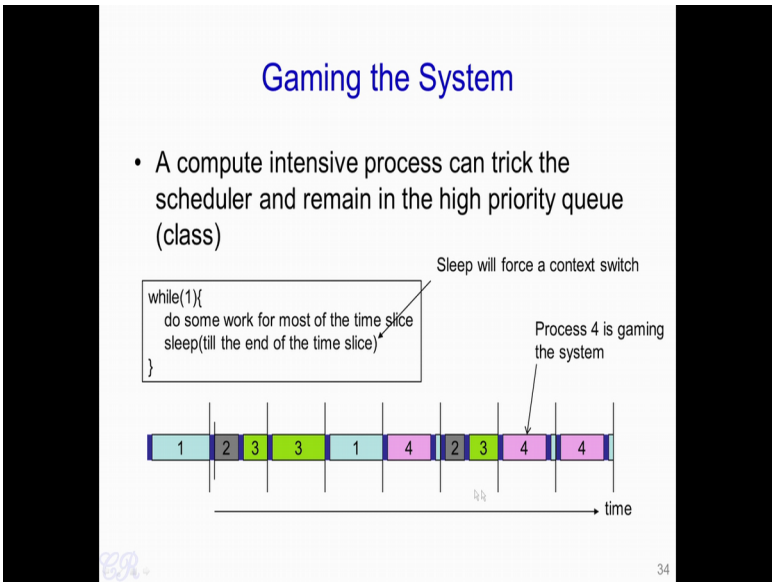
(Refer Slide Time: 18:41)



The basic idea of a multilevel feedback queue based scheduling algorithm is the following; all processes start with the highest priority or a base priority. Now if the process finishes its time slice, that is if the process executes until its time slice completes then the assumption is made that process is a CPU bound process. So it is moved to the next lower priority in the class.

On the other hand if the process does not complete its time slice, it is assume that the process is an I/O bound process and therefore it would either increase the priority class to a higher priority class or keep it on the same priority class. So this as you can see is a dynamic way of changing priority classes of a process. However, the drawback is that we could still have starvation and then techniques are to be implemented where starvation needs to dealt with.

(Refer Slide Time: 19:46)



Another drawback of this particular scheduling algorithm is that, a malicious process could game the system and always remain in the high priority class. So, let us see how this works with an example (mentioned in above slide). So let us assume the malicious process knows the time slice of the system that is it knows whenever the time slice completes and it knows when a context switch occurs. So let us say it has a loop such as this over here (refer above slide). Essentially, the malicious process will do some work for most of the time slice and then it will sleep till the end of the time slice.

Now as we know when a process goes to sleep, it goes from the running state to the blocked state. Therefore, sleep will force a context switch to occur. Therefore, what is going to happen is that a new process will execute for the remaining of the time slice. Thus, for the entire time slice for instance over here (refer above slide) process 4 which is a malicious process would run for most of the time slice and then just before the time slice completes it goes to sleep; thus, forcing an other process to execute.

The other process 1 will now execute and just execute for a small duration. At the end of the time slice the scheduler is going to see that process 1 has completed the time slice, so it is going to assume that process 1 is a CPU bound process and move the process 1 to a priority class which is lower.

On the other hand, process 4 which has blocked on a sleep will remain in the high priority class. Thus, process 4 will be able to game the system. It will continue to execute from the high priority class while another process such as process 1 will be moved on to a lower priority class.

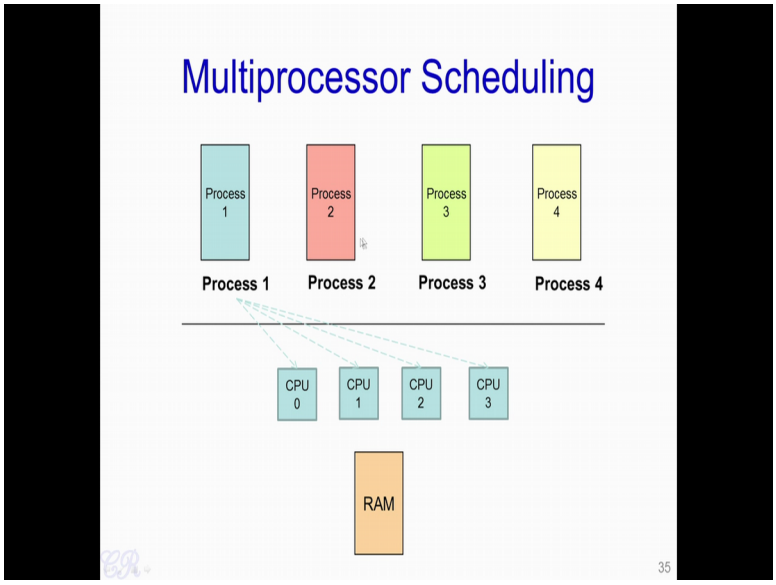
Thank you.

Introduction to Operating Systems
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week - 05
Lecture – 20
Multiprocessor Scheduling

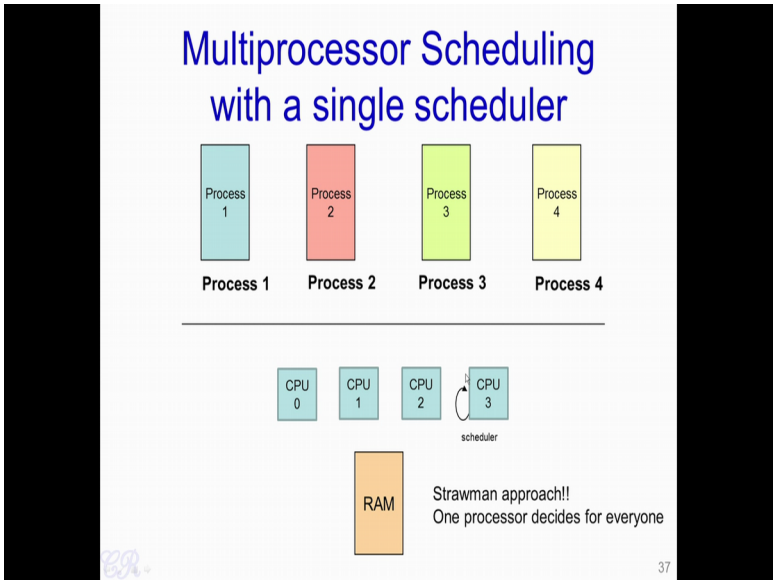
Hello. So far we had seen how scheduling algorithms are designed for a single CPU. So, the scheduling algorithm would choose a process to execute for the CPU. In this particular video, we will look at Multiprocessor Scheduling Algorithms.

(Refer Slide Time: 00:35)



Essentially, we will see that if we have multiple processors in the system, how a scheduling algorithm could schedule processes into these various CPU's?

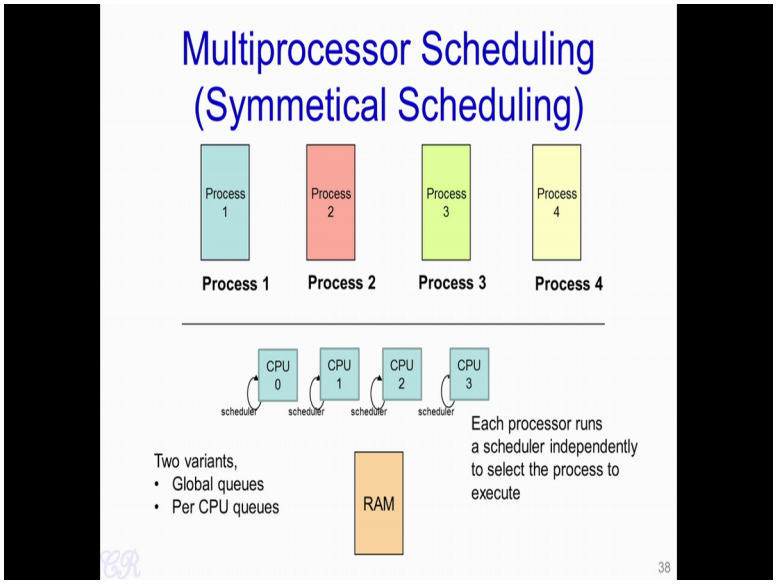
(Refer Slide Time: 00:45)



One very simple scheme for a multiprocessor scheduler is where there is a dedicated CPU to run the scheduler. This scheduler decides for everyone (CPU 3 is scheduler mentioned in above slide), essentially decides which process should run in which CPU. Now implementing this scheme is very simple. So, this scheduler is going to have a local queue of processes which are ready to run and it would use some mechanism to decide which of these processes should be scheduled into which CPU. The limitation, as one would expect is the performance degradation.

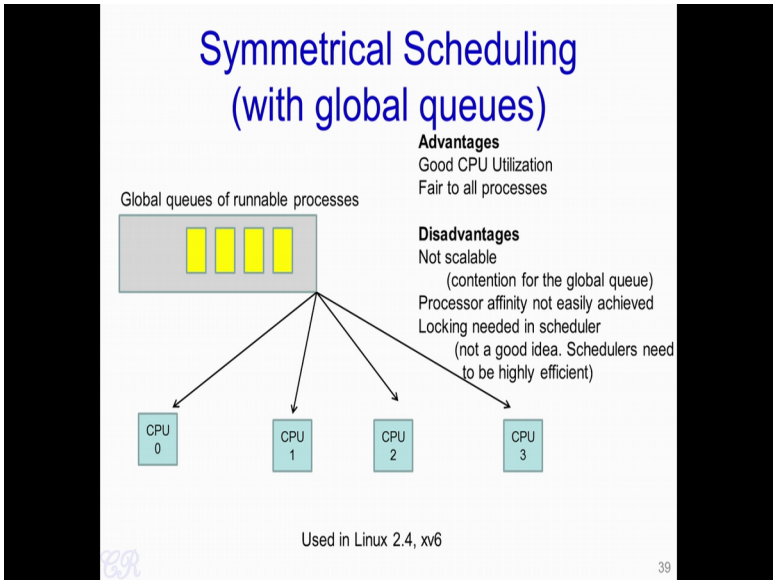
Since, all CPU's are waiting for this scheduling CPU to tell it which process to execute next and this could happen for instance every 10 milliseconds. Therefore, there could be significant overheads due to scheduling.

(Refer Slide Time: 01:37)



Another multiprocessor scheduling scheme is the symmetrical scheduling scheme. Here instead of a single CPU which decides for all processors in the system, here each CPU runs its own scheduler which is independent of each other (mentioned in above slide). Therefore, each CPU at the end of a time slice decides which process it is going to execute next. Now, there are two variants of the symmetrical scheduling scheme - one is with a Global queue another with Per CPU queues. We will look at each of these variants next.

(Refer Slide Time: 02:16)



In the symmetrical scheduling scheme with global queues, there is exactly one queue of ready processes which is shared among all processors (refer above slide). These schedulers in each CPU would need to look up this global queue to decide upon which process to run next.

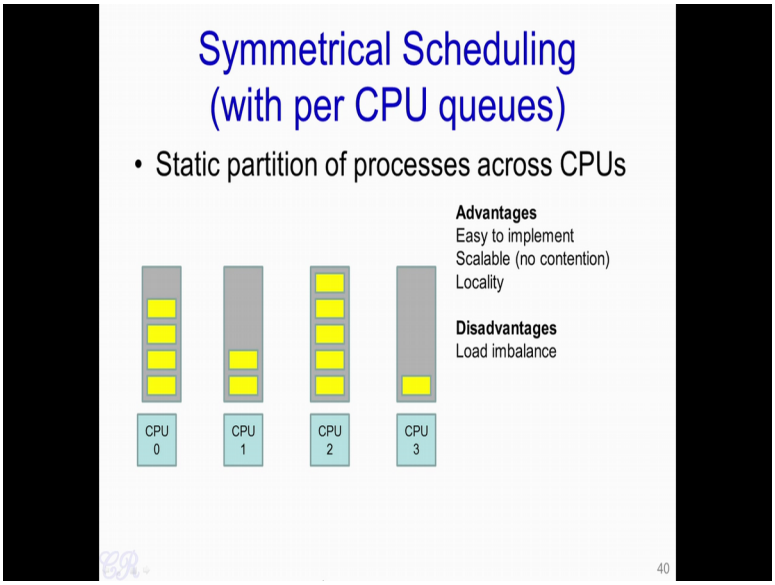
For instance, CPU 0 will need to look up this global queue and decide on a particular process to run next. That process will then change from the ready state to the running state. The advantage of the scheme is that there is good CPU utilization and fairness to all processes. The drawback of this particular scheme comes from the fact that a single queue is shared among the various processors in the system. Thus, we could reach a state where there are two processors which query the queue at exactly the same time and pick exactly the same process to execute. Thus, a single process may execute in the same instant of time in two different CPU's.

In order to prevent such issues it is require that access to this particular queue is serialized. That is, if CPU 0 wants to access the queue no other CPU should be able to access that queue during that particular time. This ensures that only CPU could choose a process at a particular instant of time.

Now, this mechanism of serializing access to the global queue is achieved by a technique known as Locking. Now, while this locking mechanism would work it is not a good idea because it will not scale. Essentially, consider the fact that instead of 4 processors in the system there are now 64 processors and all these processors have a serialized access to this global queue. Thus, when CPU 0 is accessing this global queue, all other processors will have to wait and this could lead to considerable amount of time during scheduling.

Another disadvantage comes from the fact that processor affinity is not easily achieved. So, what is Processor Affinity? Processor Affinity is an option given to users in the system to choose which processor they want their process to execute in. For instance, user may decide that he wants his process to execute only in CPU 0 and no other CPU's. Now, in this particular scheme it is more difficult to implement processor affinity. Now this particular scheme is used in Linux 2.4 Kernels as well as the xv6 operating system.

(Refer Slide Time: 05:12)



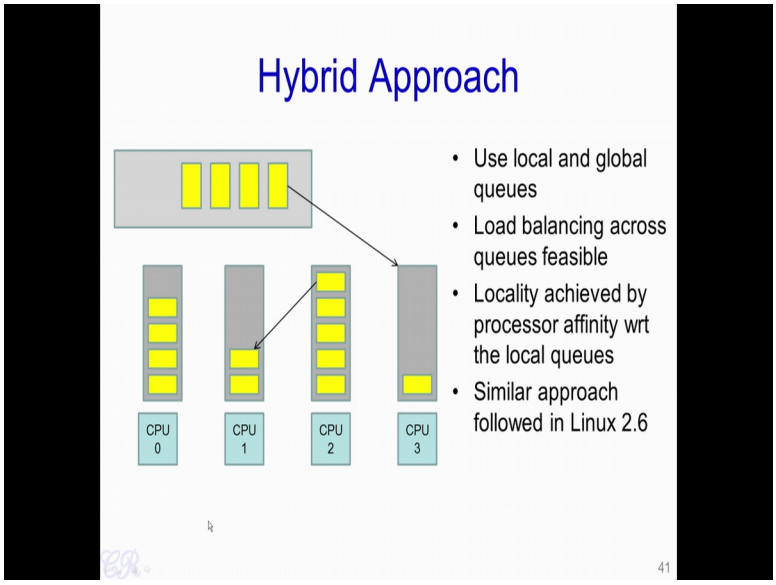
The second variant of symmetrical scheduling is where there is a single CPU queue for every CPU that is each CPU has its own queue of ready processes. Thus, each CPU needs to only look at its own queue to decide which process it needs to execute next.

Now, this particular strategy uses a static partitioning of the processes, that is at the start

of execution of a process either the user or the operating system decides which CPU the process needs to execute, and the process is then placed in that corresponding CPU's queue (refer above slide). As you can see, the advantage is that it is easy to implement, there is no locking mechanism present, so it is quite scalable as well as due to locality, in the sense that each CPU just needs to look at its own CPU queue and is not bothered about any other CPU queues that are present. Therefore, there is no degradation of performance. The advantage of the scheme is that it is easy to implement and it is scalable. Scalable comes from the fact that there is no locking mechanism or serialization of the accesses to the CPU queues.

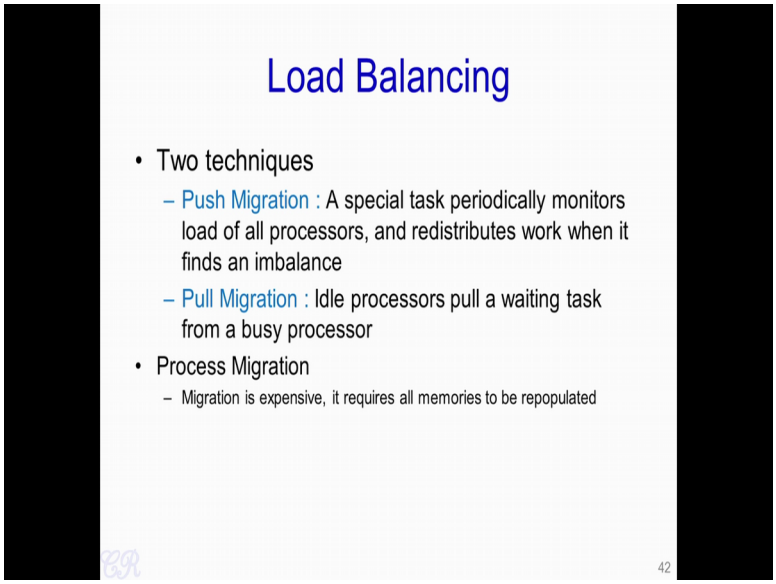
Another advantage is that the choice is local. Essentially the CPU 0 is only concerned with its own queue and not concerned with any other queue present in the system. Therefore, the choice made is very quick, there is minimum performance degradation. The drawback of the scheme is that it could lead to load imbalance. Load imbalance occurs when some CPU queues have a lot of processes to execute or rather some CPUs have a lot of processes in the ready state while other processes such as CPU 3 have just very few processes in the ready state (mentioned in above image). Thus CPU 2 is doing lot more work compare to CPU 3.

(Refer Slide Time: 07:29)



A third way to do symmetrical scheduling is a Hybrid Approach. So, this approach is used in Linux Kernels from 2.6 onwards. So, essentially this approach uses both local queues as well as global queues (mentioned in above slide). The local queues are the queues which are associated with each CPU. Essentially, each CPU has its own local queue while the global queue is shared among the CPU's. The global queue is used to ensure that load balancing is maintained that is it is going to ensure that each CPU would have a fair share of the processes to execute. The local queues would ensure locality, by having the local queue the performance degradation of the system is minimized.

(Refer Slide Time: 08:23)



Beside the global queue there are two more techniques to achieve Load Balancing - one is the Push Migration and the other is the Pull Migration. With the Push Migration, a special task would periodically monitor the load of all processors and redistribute work whenever it finds an imbalance among the processors. With Pull Migration, an idle processor will pull a task from a busy processor and start to execute it.

Thus migrating processes from busy processors to less busy processors will achieve load balancing. This being said, process migration should be done with the pinch of salt. Essentially, migrating a process from one CPU to another is expensive. Whenever a process migrates from one CPU to another, all memories with related to the new CPU

needs to be repopulated, by this we mean the cache memory associated with that CPU needs to load the migrating processes instructions and data. Similarly, the TLB needs to be flushed and so on. This could lead to significant overheads. Thus process migration should be done with only if required.

With this we will end this video which looked at Multiprocessor Scheduling Schemes and Load Balancing among the different CPU's.

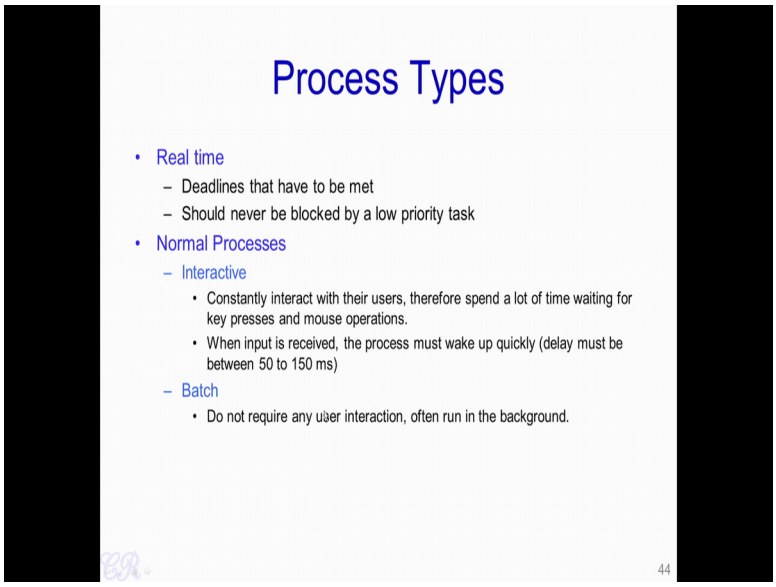
Thank you.

Introduction to Operating Systems
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week - 05
Lecture - 21
Scheduling in Linux (O(n) and O(1) Scheduler)

Hello. In this lecture we will look at Scheduling in the Linux operating systems. Essentially, we will look at how the schedulers in the Linux kernel evolved over time. So, the reference for this particular lecture is this particular book, *Understanding the Linux Kernel*, 3rd Edition by Daniel Bovet and Marco Cesati.

(Refer Slide Time: 00:38)



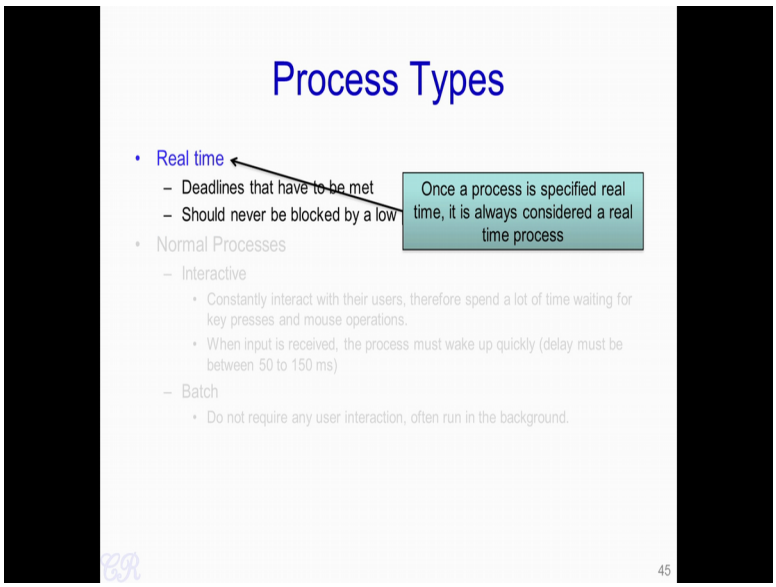
Linux classifies processes into 2 types, one is the real time process and the other is normal processes. Essentially a process could be either real time or a normal process. Real time processes are those which have very strict deadlines. For instance, there could processors involving control of a robot or data acquisition systems or EIC controllers. Real time processes should never miss a deadline and they should never be blocked by a low priority task.

The other type of processes, are the non real time processes or in generally called normal

processes. So, normal processes are of two types they are Interactive or Batch. An interactive process constantly interacts with the users, therefore spends a lot time waiting for key presses and mouse operations.

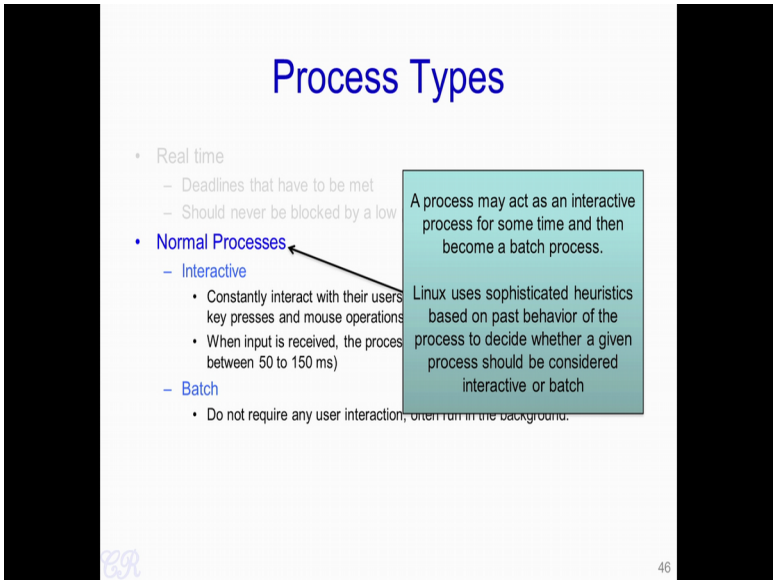
So typically, you could consider these as I/O bound processes. When an input is received, the process typically should wake up with in 50 to 150 milliseconds. If it gets delayed more than 150 milliseconds then the system begins to look sluggish, in the sense that a user will not feel very comfortable using the system. On the other hand, batch processes are most closely similar to the CPU bound processes, which do not require any user interaction and they often run in the background. For instance - gcc compilation or some scientific operations like MATLAB are fall into these type of processes.

(Refer Slide Time: 02:15)



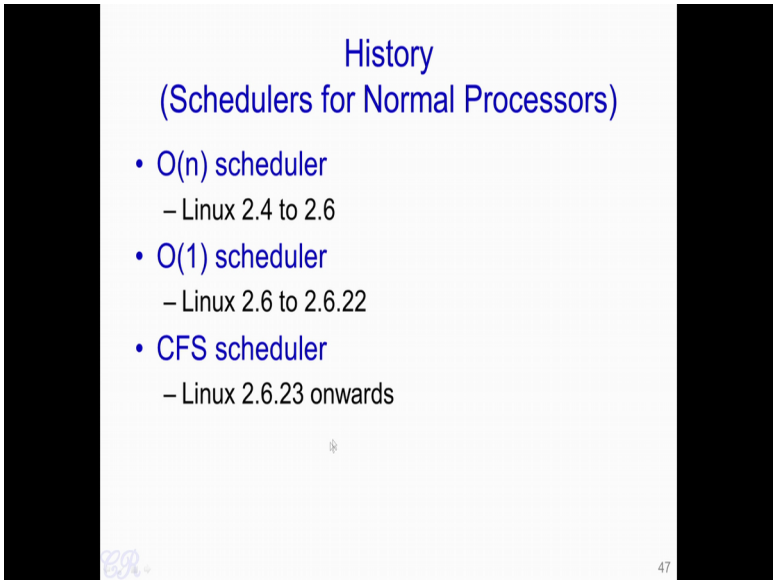
Once a process is specified as a real time process, it is always considered as a real time process. In the sense that once a user specifies that a process is a real time process, then it is always going to be a real time process even though it does nothing, but sleeping.

(Refer Slide Time: 02:38)



On the other hand, for processes which are non real time that is the normal processes, it can behave as either an interactive process at one point of time or at other points in time as a batch process. Essentially a normal process could behave as an interactive process or a batch process and this behavior could vary from time to time. So, in order to distinguish between interactive and batch processes, Linux uses sophisticated heuristics based on past behavior of the process to decide whether a given process should be considered a batch or an interactive process. So, we will look more into detail about this.

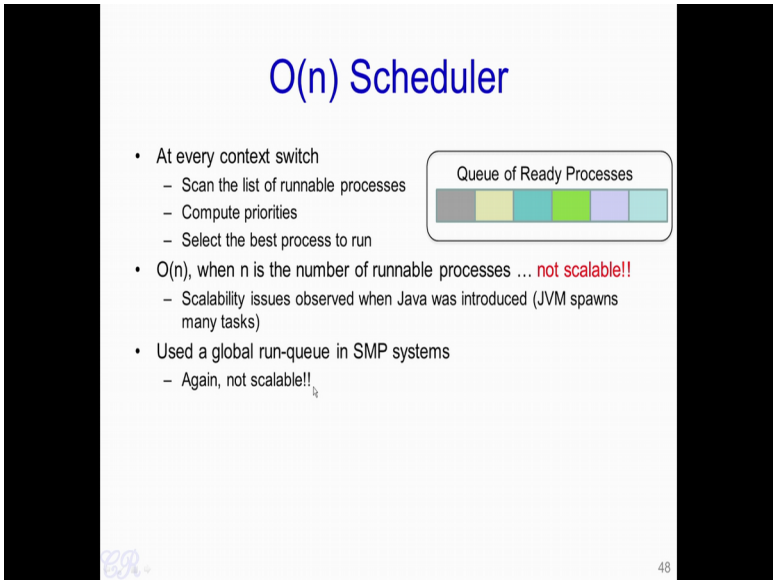
(Refer Slide Time: 03:22)



So, looking from a historic perspective, over the years starting from Linux 2.4 to 2.6 there was a scheduler which was adopted which was known as the O(n) scheduler. Then from Linux 2.6 to 2.6.22 the scheduler was called the O(1) scheduler and the scheduler currently incorporated is known as the CFS scheduler (Linux 2.6.23 onwards).

Now, we will look at each of these schedulers in the following lecture.

(Refer Slide Time: 03:54)



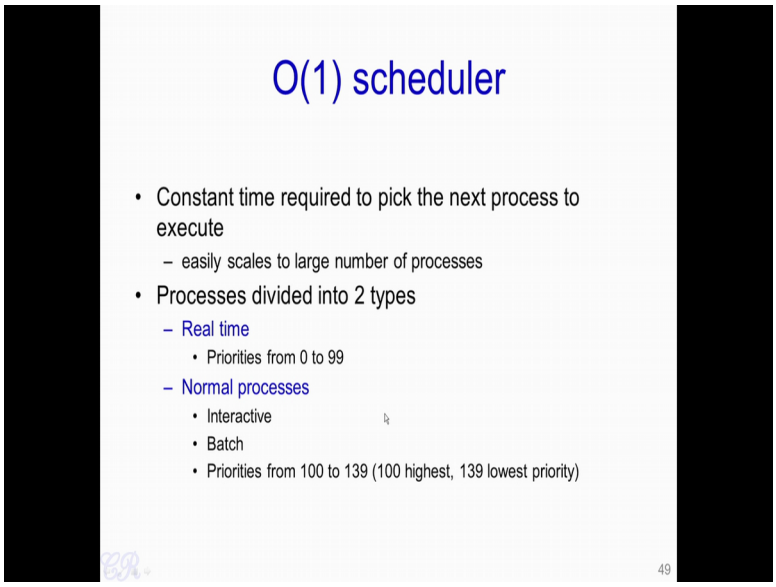
So let us start with the O(n) scheduler (refer above slide). O(n) scheduler means that the scheduler looks into the queue of ready processes which could be at most ‘n’ size, and it takes into each of these processes and then makes a choice about the process that needs to execute. So, essentially at every context switch, the O(n) scheduler will scan the list of runnable processes, compute priorities and select the best process to run. So, scanning the list of runnable processes is an O(n) job, essentially we have assume that the ready queue has up to ‘n’ processes present.

So, this obviously is not scalable. It is not scalable because as ‘n’ increases in number that is as more and more processes or tasks enter the ready queue, the time to make a context switch increases. Essentially the context switch overheads increase and this is not something which is wanted for.

So, these scalability issues with the O(n) scheduler was actually noticed when Java was introduced, roughly the early 2000s and essentially due to the fact that JVM - the Java Virtual Machine spawns multiple task, and each of these tasks are present in the ready queue. So, the scheduler would have to scan through this ready queue in order to make their choice. Another limitation of the O(n) scheduler was that it used a single global run queue for SMP systems. Essentially when you had multiprocessor systems, as we have

seen in the previous lecture, a single global run queue was used. So, this again as we have seen is not scalable.

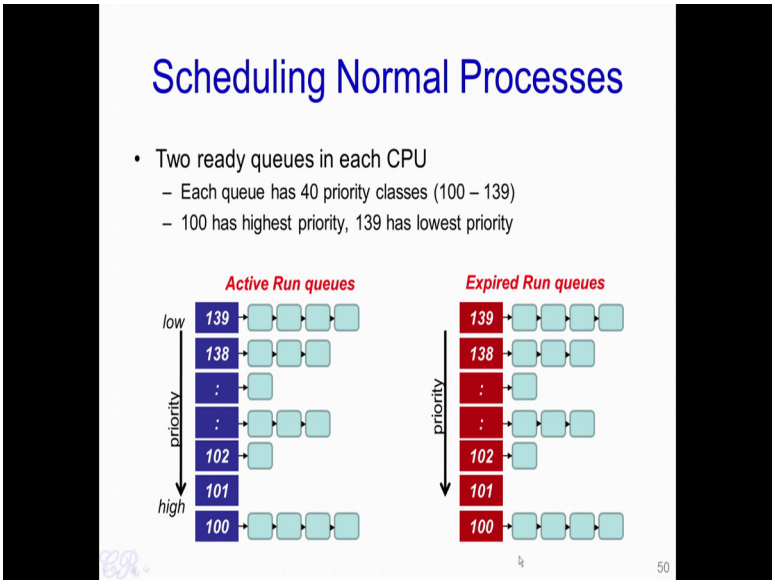
(Refer Slide Time: 05:48)



So the next choice was something known as the O(1) scheduler. The O(1) scheduler would make a scheduling decision in constant time, irrespective of the number of jobs which are present in the ready queue. Essentially it would take a constant time to pick the next process to execute in the CPU. So, this quite obviously scales, even with large number of processes that are present in the ready queue. In the O(1) scheduler, processes are divided into two types; the real time processes and normal processes as we have seen. So, the real time processes are given priorities from 0 to 99, with 0 being the highest priority and 99 being the least priority of a real time task.

Now, normal processes on the other hand are given priorities between 100 to 139. So, 100 is the highest priority that a normal process could have while 139 is the lowest priority that any process could have, and as we have seen before the normal processes could be either interactive or batch. We will see later how the scheduler distinguishes between the interactive and batch processes.

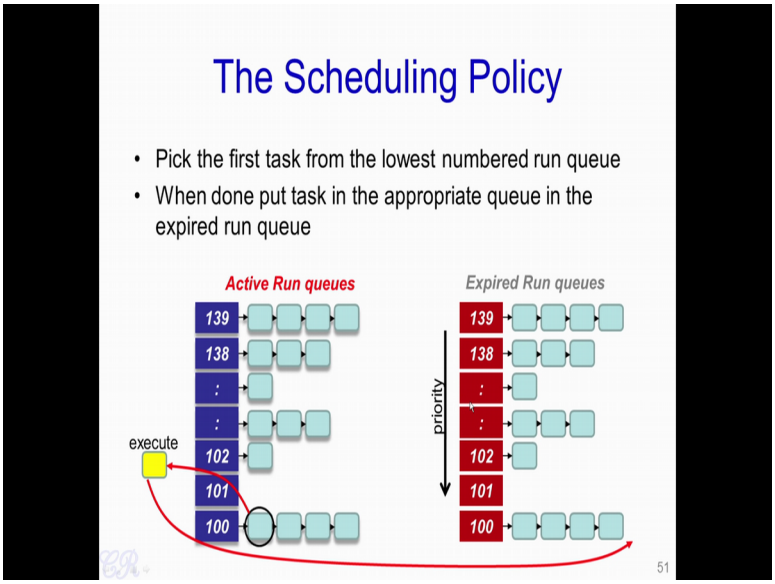
(Refer Slide Time: 07:00)



Now, we will talk about a scheduling in normal processes that is the non real time processes. Essentially the scheduling algorithm which is used is very similar to the multi level feedback queue with a slight variation. Essentially in this particular scheduler there are 40 priority classes, which are labeled from 100 to 139, and as we have discussed before the lowest priority is corresponds to the class 139 while the highest priority for the normal task is at 100 (mentioned in above slide).

So, corresponding to each of these tasks there is a ready queue. So, every process or every task present in the ready queue corresponding to a particular priority class has equal priority to be executed on the CPU. On the other hand, a process present in the 100^{th} priority class has a higher priority to execute with respect to a process present in 102 that is a process present in the ready queue corresponding to 102 (mentioned in above slide). Now based on this priority class, the scheduler maintains 2 such queues. So, one is known as the Active Run queues and the other one is known as the Expired Run queues.

(Refer Slide Time: 08:26)

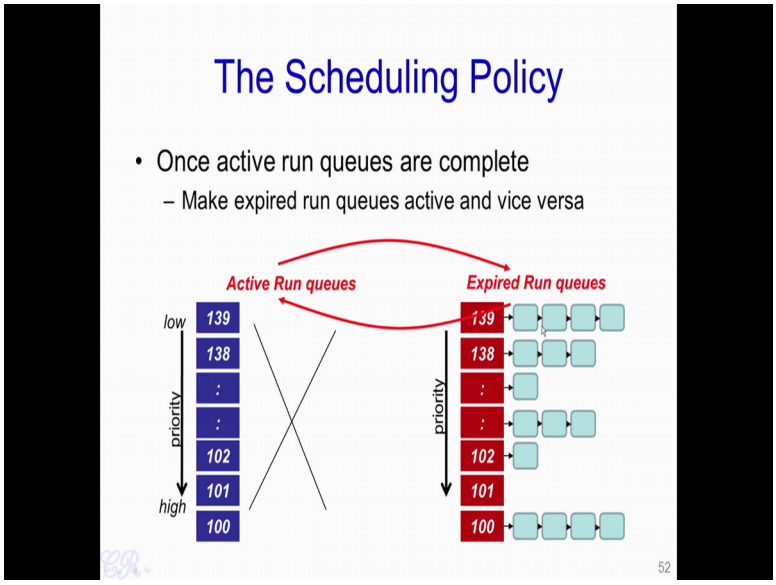


When a context switch occurs, the scheduler would scan the active run queues, starting from the 100th run queue and going up to 139. It picks the run queue which has a non zero or a non empty queue present in that (refer above slide).

So for instance over here (mentioned in above slide) it starts from 100 and picks this first particular process (circled in black) and this process is executed in the CPU. At the end of the time slice this process is put into the expired run queue. So, gradually you will see that as time proceeds and time slices complete, the scheduler keeps picking out processes from the active run queues, executes them and puts them into the expired run queues. After a while we would have a point where the entire processes in the active run queue is complete. Essentially we come to the point that none of these priority classes have any processes with them.

On the other hand the expired run queue is now filled up. Essentially because the scheduler has been adding processes to each priority class in the expired run queue.

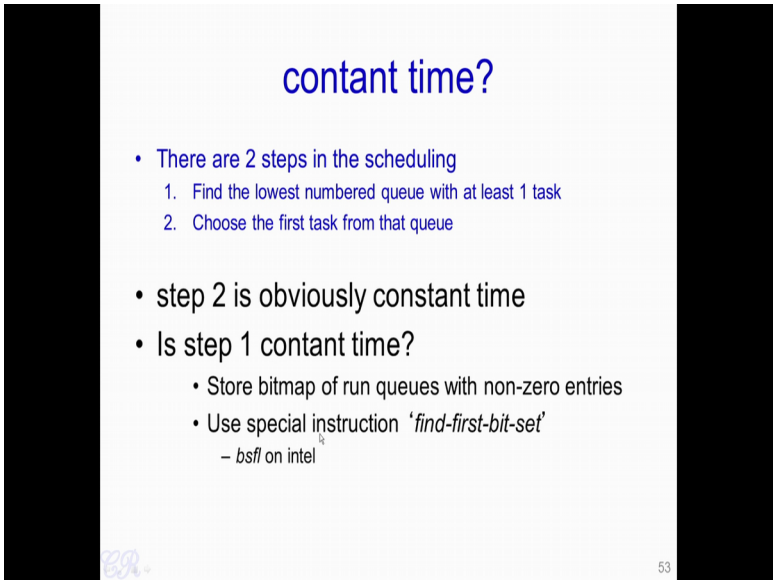
(Refer Slide Time: 09:36)



When this happens (as mentioned in above para), the scheduler would switch between the active run queues and the expired run queues. Now this (Expired Run queues mentioned in above slide) is the queue which actually becomes active while this (Active Run queues) becomes the expired run queues.

Now, for consecutive time slice completions, the scheduler would pick task from this particular queue (from right hand side queues mentioned in above slide) and execute these task. After execution these task could be moved to this queue over here (left hand side queues), which is now the expired run queue. In this way there is a toggling between these two queues. All tasks present in one queue are executed and moved on to the task in the other queue. Then all tasks over here executed and after executing the task are moved to the previous queue and so on. So, this process goes on continuously. The main reason for having such a technique in the Linux kernels scheduler is that it prevents starvation it ensures that every process gets a turn to execute in the CPU.

(Refer Slide Time: 10:39)



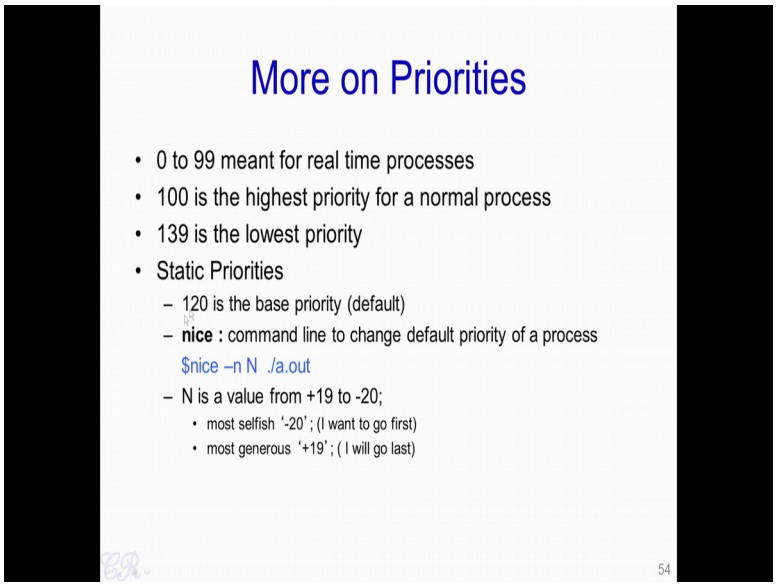
Recall that we call this particular scheduler as an O(1) scheduler. Essentially this means it is constant time scheduler irrespective of the number of processes present in the ready queues.

Now, let us analyze how this particular thing is constant time. Now if we recall there are 2 steps involved in scheduling a process. First is to find the lowest numbered queue with at least 1 task present in that corresponding ready queue. Second choose the first task from that ready queue. Now how are these two things constant time? It is quite obvious that the second step is constant time, essentially we are choosing from the head of the queue and that is obviously done in a fixed amount of time irrespective of the size of the processes. Next question is how is step 1 constant time? So, essentially we know that the scheduler scans through all the priority queues starting from 100 and going up to 139 and chooses the lowest numbered queue which has at least one task, essentially it chooses the lowest non empty queue.

So, this does not seem to be constant time. So, how does the scheduler actually implement this, so that the time taken to actually find the lowest non empty queue is independent of the number of processes? In order to do this, the scheduler or the Linux scheduler makes use of two things, the first is a bitmap. This is a 40 bit bitmap which

stores the run queues with non zero entities. Essentially this particular bitmap stores zero for a class which is empty, and it stores one for a class which has non zero entities. Second it uses a special instruction known as find first bit set. So, this particular instruction (mentioned in above image) for example, there is a bsfl instruction on Intel would look at this 40 bit bitmap and choose the lowest index, which is non zero. Essentially this would give us the lowest numbered non zero bit in the bitmap and this corresponds to the lowest non zero or rather lowest non empty priority queue.

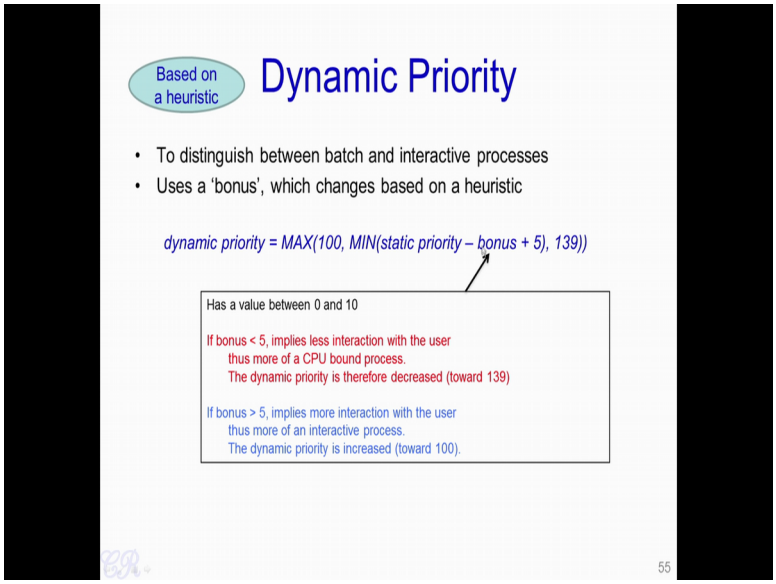
(Refer Slide Time: 13:16)



Now, let us look more at the priorities of the O(1) scheduler. So, we had seen that a priority 0 to 99 are meant for real time tasks, while priority 100 is the highest priority that can be given to a normal process, while priority 139 is the lowest priority that a normal process could have. Now these are the ranges of the priorities for the normal process that is a normal process could have anything from a priority of 100 to a priority of 139. The priority 120 is known as the base priority and taken by default. So, whenever you start a program for example, whenever you start a.out program it is given the priority 120. Now you could change this base priority by using the command called nice. The nice command line is used to change the default priority of the process from something from 120 to something else.

So, the command would look something like this way (mentioned in above slide). So, $nice –n N ./a.out, where the capital N can take a value of +19 to -20. So, essentially the value that we specified here (in place of N) gets added to the base priority. So, for example, if I say $nice –n +19 ./a.out, so, the process that I am starting will have a priority 120 + 19. So, that is 139. So, this would be the least priority that it could have. On the other hand, you could also specify a priority like $nice –n -20 ./a.out. So, this would give the process a priority of 100. So, this is the highest priority that the process could get considering that it is a normal process. So, these are static priorities that the process could get during the start of execution.

(Refer Slide Time: 15:19)



Besides the static priority, the scheduler also sets something known as the Dynamic priority. This dynamic priority is based on heuristics and is set based on whether the process acts like a batch process or an interactive process. Essentially its set based on whether the process has more of I/O operations or is more CPU bound.

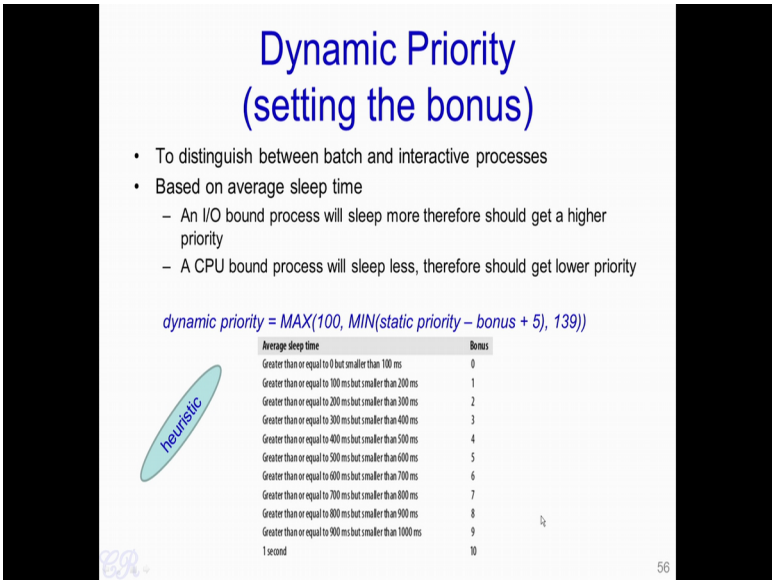
So, the scheduler uses some heuristics which is present over here (mentioned in above slide) to compute the dynamic priority. Essentially it takes the static priority which we have set which is either 120 by default a given at the start of the execution, or the priority based on the nice value and subtract something known as the bonus and adds plus 5. So,

it takes min of static priority minus bonus plus 5 and 139 (i.e MIN(static priority – bonus + 5), 139) and it then takes the max of 100 and this value i.e (MAX(100, MIN(static priority – bonus + 5), 139)). So, for instance let us say we start the process with the default static priority that is 120 and we give a bonus of say 3, so then we have like 120 - 3 + 5 that is one 122. So, MIN(122,139) is 122 and you take MAX(100,122) is 122. So, the dynamic priority for this particular example was 122.

Now, the crucial thing is this bonus. So, how is this particular bonus set? So, essentially this bonus has a value between 0 and 10 if the bonus is less than 5, it implies there is less interaction with the user. Thus the process is assumed to be more of a CPU bound process. The dynamic priority is therefore, decreased, essentially the dynamic priority goes towards 139 of a low priority task (refer above slide). If the bonus is greater than 5 on the other hand it implies that more there is more interaction with the user that is the process acts more like a I/O bound process and essentially the process behaves more like an interactive process. The dynamic priority is therefore, increased that is it goes towards 100 (refer above slide).

So, you can take two examples. For instance we have seen a bonus of 3 and we had seen that it is resulted in a dynamic priority of 122 (from the example given in above para) implying that it is more of a CPU bound process therefore, is given a lower priority in the system. On the other hand if you take a bonus of 8 and compute the same thing you will see that the dynamic priority actually reduces. So, the crucial part is now how do we actually set the value of bonus?

(Refer Slide Time: 18:11)

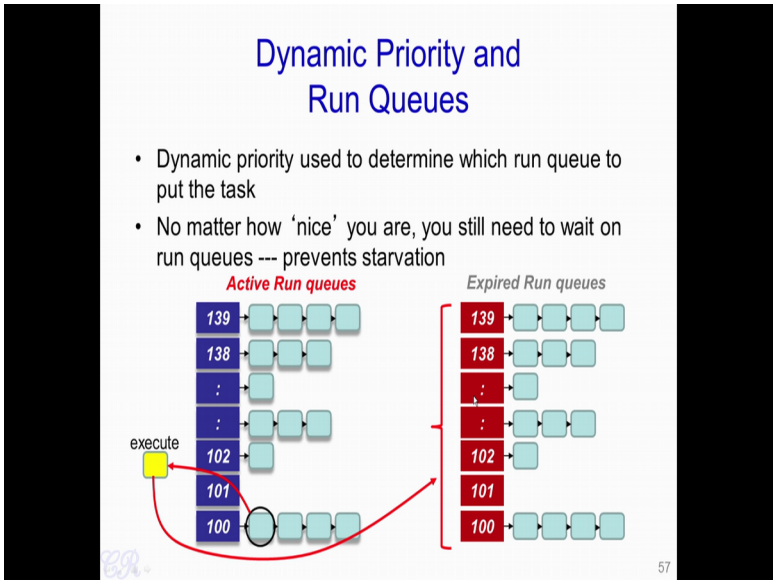


So, the bonus is in the scheduler is determined by the average sleep time of a process. The assumption here is I/O bound processes will sleep, or in another words will block more, therefore should get a higher priority. So, therefore, I/O bound processes will have higher value of bonus.

On the other hand CPU bound processes will sleep less, therefore should get a lower priority. If we look at this table (refer above slide) and if we see that the average sleep time for a process is greater than 200, but less than 300 milliseconds, it is given a bonus of 2. So, this is just heuristic and followed as a formula in the kernel. On the other hand if the process has an average sleep time greater than or equal to 800 millisecond but smaller than 900 millisecond, we give it a larger bonus of 8. Thus these processes, process with the high value of bonus, that is those processes which sleep more are considered interactive processes and given a dynamic priority which is closer to 100.

On the other hand processes which sleep less are assumed to be CPU bound processes, and given a bonus which is low, and as a result the dynamic priority is lowered that is it goes towards 139.

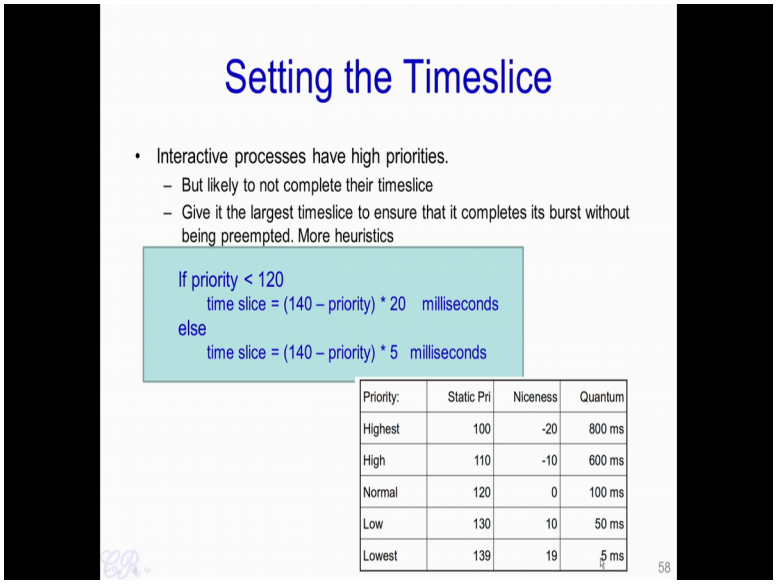
(Refer Slide Time: 19:40)



So, how are these dynamic priorities affecting the scheduling algorithm? Essentially if you come back to the active and expired run queues (mentioned in above slide), the dynamic priority determines which queue the particular process would be placed in. For instance when we choose a particular process to execute, say from the 100th priority class and after it executes its average sleep time is then used to compute the bonus, and thereafter use to compute the dynamic priority, and then this process (process which was executed now) is placed in the corresponding priority class.

So as a result based on the average sleep time, when the process is placed back in the expired run queue the class in which it is placed in would depend on its dynamic priority.

(Refer Slide Time: 20:28)



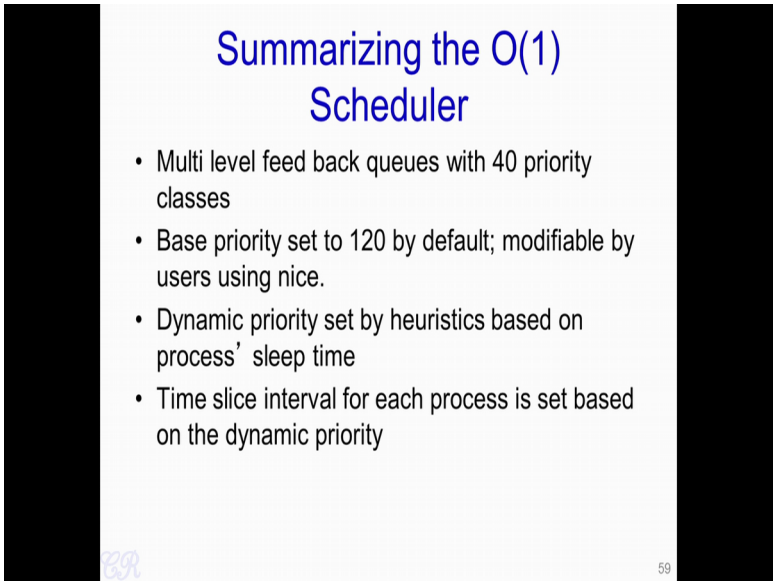
So besides the dynamic priority, there is also the time slice which is adjusted by the scheduler. Essentially interactive processes have high priorities, but they are more of I/O bound processes, and they are likely to have short CPU burst times. These processes, essentially interactive processes are given the largest time slice to ensure that it completes its burst without being preempted. So, in order to set the time slice we are using more heuristics, such as this (If condition statement in above slide image). Essentially if the priority is less than 120 that is it is an interactive process then the time slice is given as (140 – priority) * 20 milliseconds.

However if it is more of a batch process that is, using more of CPU activity then the time slice is set by the lower statement (else statement in above slide image) that is time slice is equal to (140 – priority) * 5 milliseconds. So, as a result of this if you see this particular table (refer above slide), you see that if you have a static priority of 100 that is obtained by having a nice value of -20. So, it means that this is a interactive process and is given a large time slice of 800 millisecond. On the other hand, a process with a static priority of 139 that is 120 + having a nice value of 19 (120 + 19) it is given the lowest time slice or the smallest time slice of 5 milliseconds.

So, thus we see that based on the priority of the process the corresponding time slice

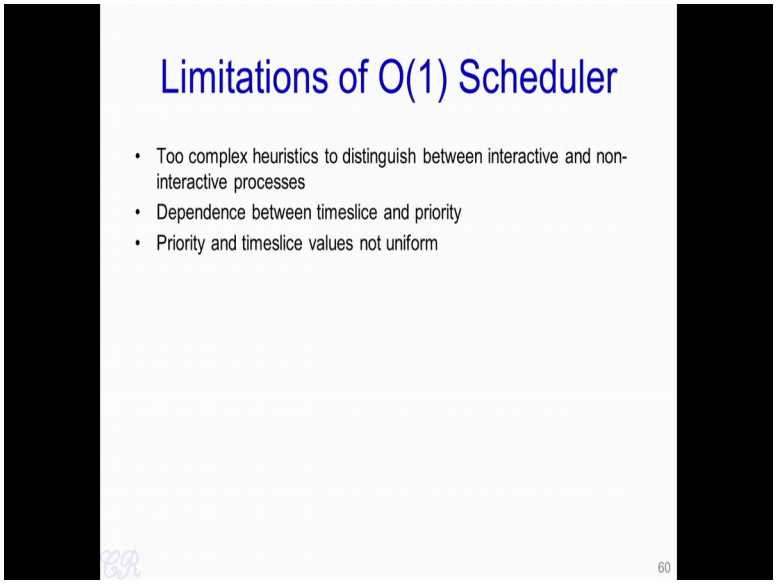
given to that process is fixed.

(Refer Slide Time: 22:15)



So summarizing the O(1) scheduler it is a multi level feedback queue with 40 priority classes. The base priority is set to 120, but modifiable by the use of this command called nice. The dynamic priority set by heuristics is based on the processes average sleep time. The time slice interval for each process is set based on the dynamic priority.

(Refer Slide Time: 22:39)



So the limitations of the O(1) scheduler is as follows; So, the O(1) scheduler uses a lot of complex heuristics to distinguish between interactive and non interactive processes. So, essentially there is a dependency between the time slice and the priority, and the priority and time slice values are not uniformed. That is for processes with a priority of 100 compare to processes with the priority of 139; if you compute the time slices in these two ranges you see that it is not uniform.

If you look back in the table over here (refer slide time 20:28), you see going from a priority of 100 to 110 has a time quantum of a 200 milliseconds; however, going from 130 to 139 has a time slice difference of just 45 milliseconds. So, with this we will end this particular video lecture.

So in the next lecture we look at the CFS scheduler, which is the latest scheduler or the current scheduler which is used in Linux kernels.

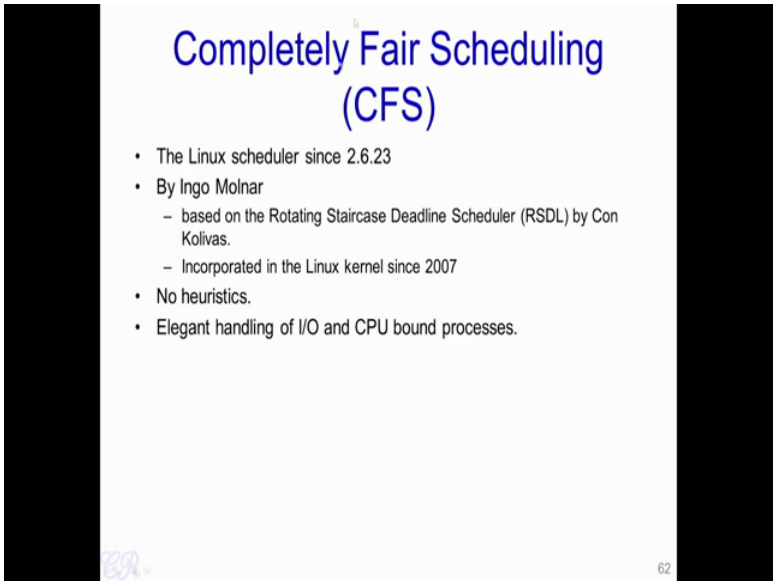
Thank you.

Introduction to Operating Systems
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week – 05
Lecture – 22
Completely Fair Scheduling

In this video lecture, we will look at the Completely Fair Scheduler. So, the completely fair scheduler or the CFS scheduler is the default scheduler used in Linux kernels in the latest versions.

(Refer Slide Time: 00:29)

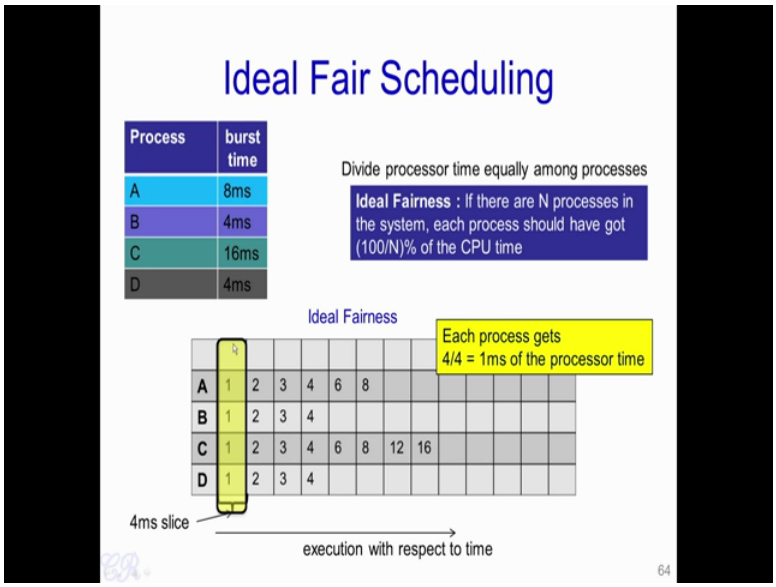


So the CFS scheduler has been incorporated in the Linux kernel since version number 2.6.23, and has been used as the scheduling algorithm since 2007. So, it was based on the Rotating Staircase Deadline Scheduler by Con Kolivas. So, the advantage of the CFS scheduler compared to the O(1)scheduler in particular is that there are no heuristics which are used and there is very elegant handling of I/O bound and CPU bound processes.

Essentially the interactive and non-interactive or batch processes are very easily fit into this particular scheduler. So, we will see a very brief overview of the CFS scheduler.

Now as the name suggest the CFS scheduler or the completely fair scheduler aims at dividing the processor time or the CPU time fairly or equally among the processes.

(Refer Slide Time: 01:34)



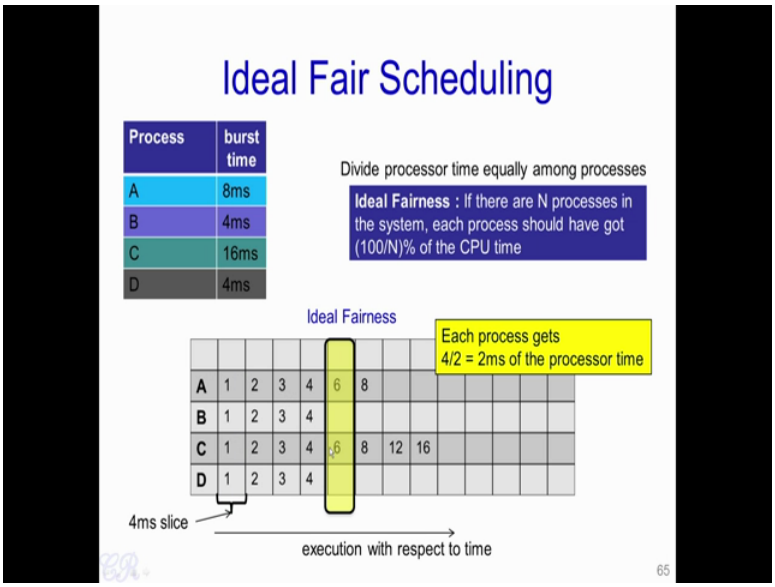
In other words, if there are N processes present in the system or present in the ready queue and waiting to be scheduled, then each process will receive (100/N)%of the CPU time. So, this is the Ideal Fairness. Let us take a small theoretical example for this ideal fair scheduling.

Let us consider the four processes A, B, C and D, and having the burst time 8 milliseconds, 4 milliseconds, 16 milliseconds and D has the 4 milliseconds respectively (refer above slide). So, what we will do is let us just divide the time into quanta of 4 milliseconds slices and what we will now see is how the ideal fair scheduling should take place. So, in an ideal fair scheduling, at the end of say this 4 milliseconds epoch, all processes which are in the ready queue should have executed for the same amount of clock cycles.

For instance, if we look at this particular first epoch (first cycle), so it has 4 milliseconds, and we have four processes are present in the ready queue; therefore each process should get 4 divided by 4 that is 1 milliseconds of processor time (refer above Ideal fairness

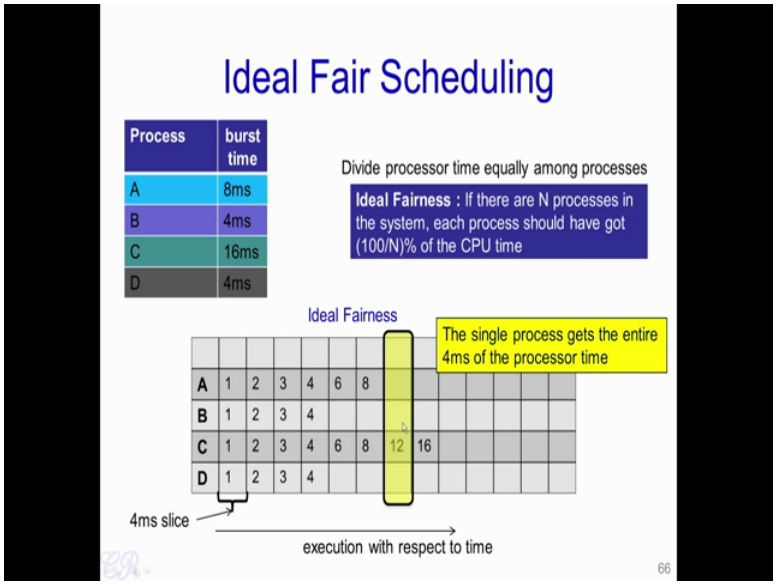
block). Therefore, A, B, C and D will execute for 1 millisecond (each). In a similar way for the 2nd cycle, there are four processes again, and therefore, these four processes get an equal share of the slice. So, each process executes for 1 millisecond again. So, therefore, in all A has executed for 2 milliseconds, B for 2, C for 2, and D for 2 milliseconds. Similarly, for 3 and for 4 (same cycle as first and second), so at the end of the 4th epoch, we see that processes B and D have completed (4ms of both processes are over). So, what happens next?

(Refer Slide Time: 03:32)



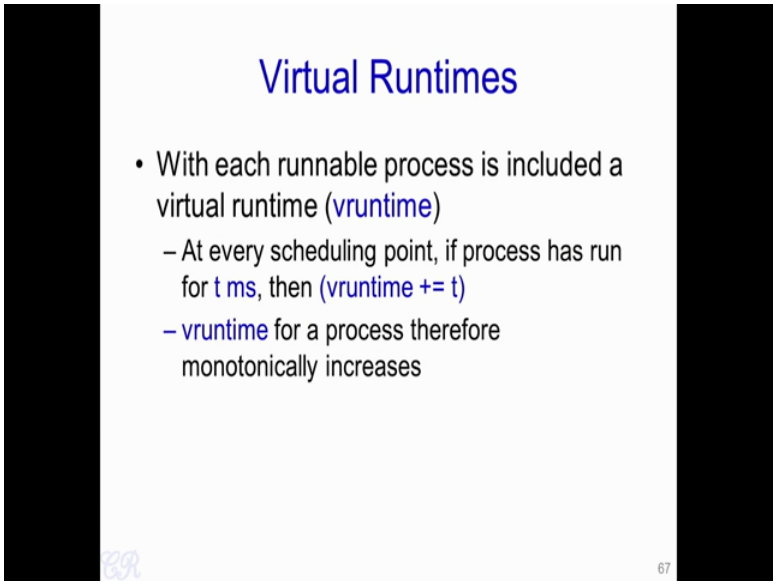
Now after B and D completes, we see that we have two processes present in the ready queue that is A and C, also the time quanta remains as 4 milliseconds. So, now each process gets 4 (time slice) divided by 2 (remaining process) that is 2 milliseconds of the processor time. Therefore, process A executes for 2 milliseconds, similarly process C executes for 2 milliseconds. Similarly, for the next epoch, A executes for 2 more milliseconds, and C executes for 2 more milliseconds. So, both have executed for 8 millisecond and as a result, A has completed executing.

(Refer Slide Time: 04:16)



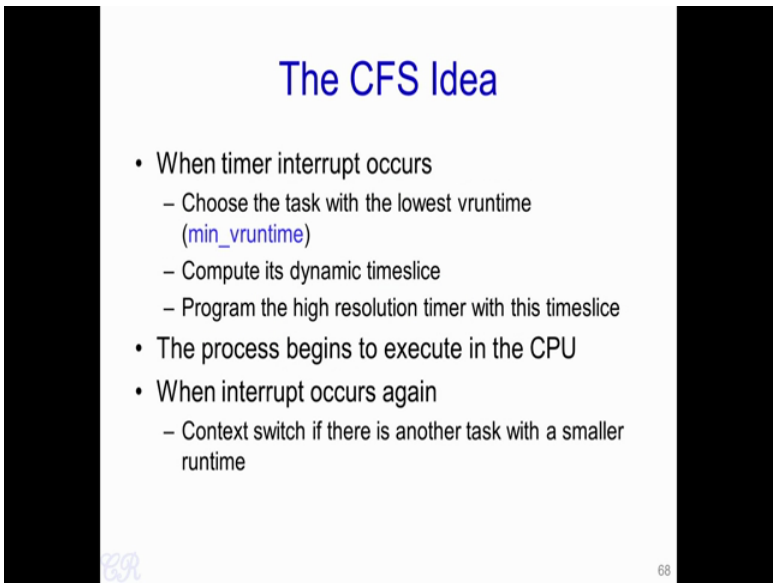
Now, the last part we see that only C is present in the ready queue and it is the since it is the only process which is present in the ready queue. So, it is given the entire slot of 4 milliseconds. So, C executes for 4 milliseconds and followed by the final slot where it executes for another 4 milliseconds to complete its burst time. So, what you see in this ideal scheduling is that in each epoch or in each slot, the scheduler is trying to divide the time equally among the processes, so that asymptotically all processes execute for the same amount of time in the CPU. So, you see that all processes execute for 4 milliseconds here, at the end of this all processes execute for 6 milliseconds then 8 milliseconds and so on. How is this ideal fair scheduling incorporated in the CFS scheduler?

(Refer Slide Time: 05:21)



So, this is done by what is known as the Virtual Runtimes. In each processes PCB that is in each processes process control block, an entry is present known as the vrun time or the virtual run time. At every scheduling point, if a process has run for t milliseconds then its vruntime is incremented by t. Vruntime for a process, therefore will monotonically increase.

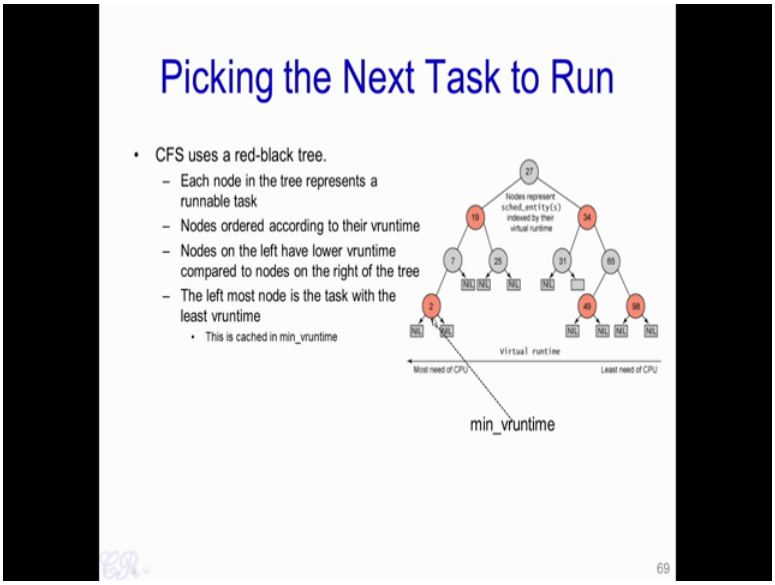
(Refer Slide Time: 05:51)



Now, the basic CFS idea is whenever there is a context switch that is required to be done, always choose the task which has the lowest vruntime. So, this is maintained by a variable called min_vruntime, this is a pointer to the task having the lowest virtual run time. So, then the time slice required is the dynamic time slice for this particular process is computed and the high resolution timer is programmed with this particular time slice.

The process begins to then execute in the CPU. When an interrupt occurs again a context switch will occur if there is another task with a smaller run time. So, you see that this particular process which is selected to run over here (from first point mentioned in above slide) will continue to run until there is another task with a lower run time.

(Refer Slide Time: 06:51)



Now in order to manage this various tasks with various run times, the CFS scheduler with quite unlike the schedulers which we seen so far do not use ready queue; instead, it uses a red black tree data structure (refer above slide). So, in this red black tree or the rb tree data structure, each node in the tree is represented as a runnable task. Nodes are ordered according to their virtual run time.

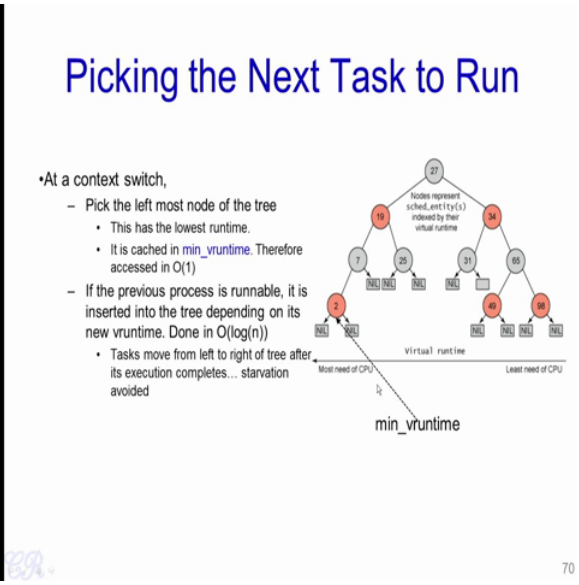
Nodes on the left have a lower run time or lower vruntime compared to nodes on the right of the tree that is if you see these particular things (rb tree in above slide), so each

node is a task and each node has a number written over here (inside the circle) which is the virtual run time for that particular task. So, you see that each task on the left (left side of the tree) has a lower virtual run time compared to task on the right.

Now, the left most node of this rb tree is the task which has the lowest vruntime or the lowest virtual run time. So, in this particular case (mentioned in above slide), it is this particular node (node with vruntime as 2) which corresponds to the task having the lowest virtual run time, therefore the scheduler should pickup this task to run next.

In order to find this task, there are two ways which are possible; one is you could traverse a tree and go towards the left until you reach a leaf, or the other way is we could directly have a pointer like the min_vruntime which points to the left most node of the tree. So, whenever the scheduler needs to make a context switch, it would just need to look into where min vruntime points to and pick out this particular task. This quite naturally will be the lowest or the task with the lowest virtual run time.

(Refer Slide Time: 08:48)

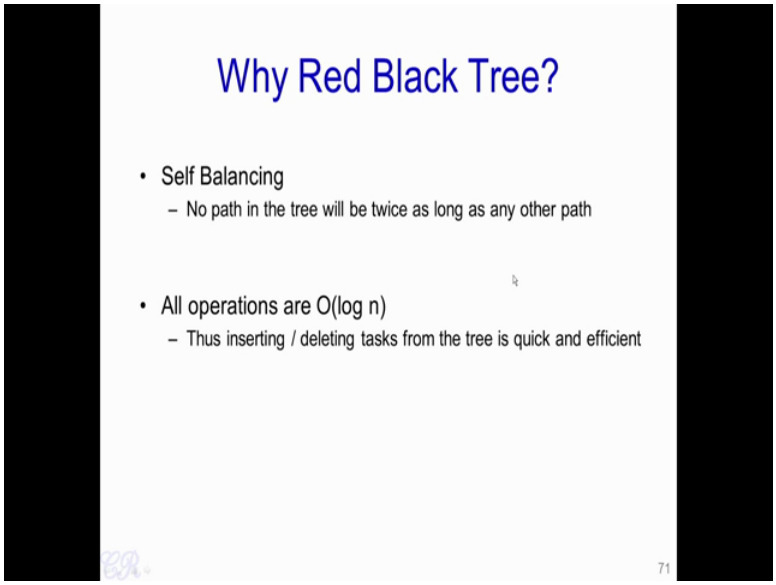


So, this choice of the lowest vruntime is can be done in O(1) and therefore, independent of the number of processes present in the rb tree. So, at the end of the time slice, if this process which is currently executing is still runnable that is it has not blocked on an I/O

or it has not exited then it is new virtual run time is computed based on the amount of time it has executed in the CPU. Then it is inserted back into the tree corresponding to its virtual run time. So, a process in other words would be picked out from the left most part of the tree because it has the lowest virtual run time and then it would execute in the CPU for some time say t milliseconds.

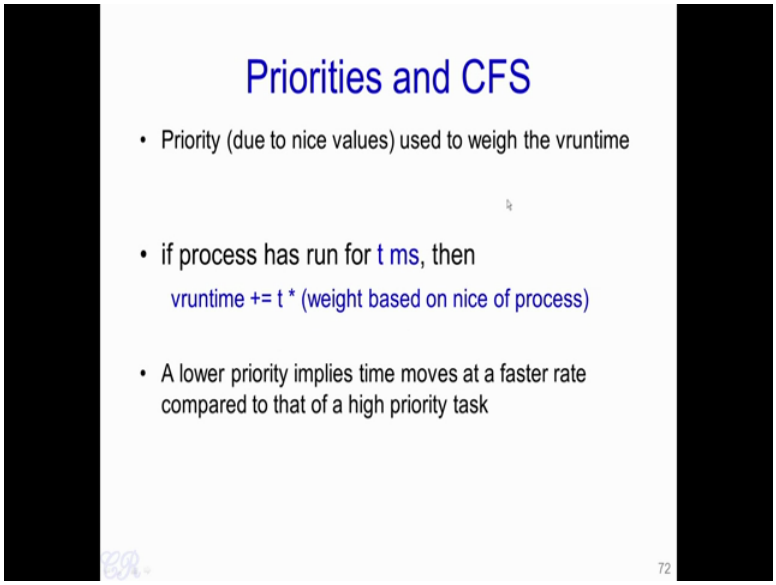
And at the end of its time slice, its virtual run time would be incremented by t values, and it would be inserted again into the tree. Now it will not go to the left of the tree, but it will rather be inserted somewhere in the middle towards the right (right side of the tree). Thus as the virtual run times increment, a process moves towards from the left towards the right. This ensures that every process gets a chance to execute because it ensures that at one point or the other, every process is going to have the minimum virtual run time in this particular tree and therefore, will get executed thus starvation is avoided.

(Refer Slide Time: 10:24)



So, why do we choose the red black tree or rather why did the Linux kernel choose the red black tree for the CFS scheduler. So, one obvious reason is the rb tree is self balancing. So, no path in the tree will be twice as long as any other path because of the self balancing nature of the tree. Due to this, all operations will be O(log n), thus inserting or deleting tasks from the tree can be quick and done very efficiently.

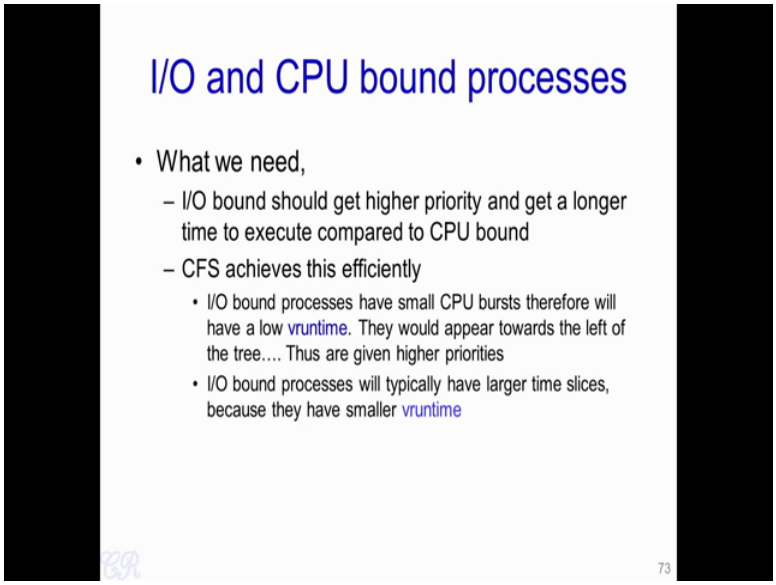
(Refer Slide Time: 10:55)



Now how are priorities implemented in the CFS scheduler? So, essentially, CFS does not use any exclusive priority based queues as we seen in the O(1) scheduler, but rather it uses priorities to only weigh the virtual run time.

For instance, if a process has run for t ms then the virtual run time is incremented by t into weight based on the nice value of the process i.e vruntime+= t * (weight based on nice of process), essentially based on the static priority of the process. So, a lower priority implies that the time moves at a faster rate compared to that of a high priority task. So, essentially what we are doing is we are providing a weight for the time that its executes, that is we are either accelerating the time or decelerating the time at which a process runs. So, this weight is used to implement priorities in the CFS scheduling algorithm.

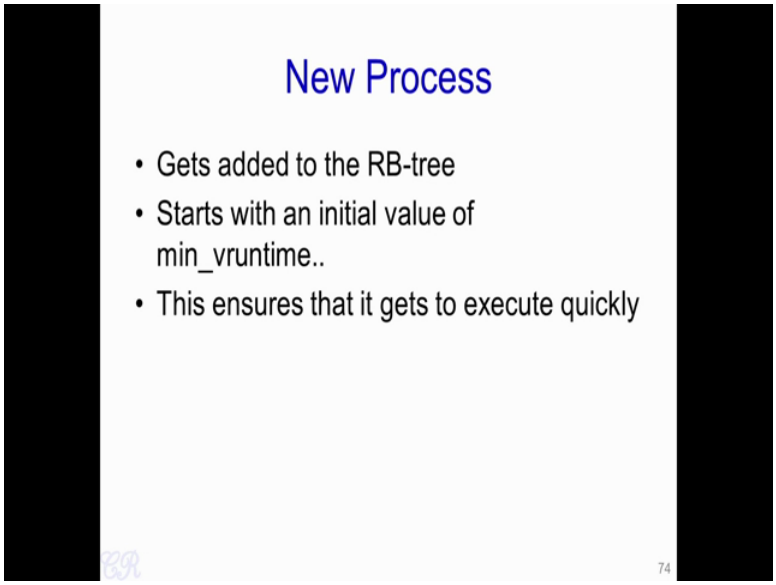
(Refer Slide Time: 12:01)



Next we will look at how the CFS scheduler distinguishes between an I/O bound and a CPU bound process. So, essentially this distinguishing is done very efficiently. It is based on the fact that I/O bound processes have a very small CPU burst, and therefore its vruntime does not increment very significantly. As a result of this, it is more often than not appearing in the left part of the rb tree. Therefore, it gets to execute more often than other processes. This is because of the fact that as we mentioned as time progresses each process in the CFS scheduler is picked up from the left most nodes and executes and then it is placed on the right; therefore, in general every process moves towards the right part of the rb tree present in the scheduler.

Now, for the I/O bound process, since the vruntime does not change too much or increments just by a small margin, it does not move to the extreme right, but rather it's still stays towards the left part of the tree. Thus very soon, it will soon find itself as a process with a lowest vruntime and will have a chance to execute again in the CPU. Now as a second effect due to the small vruntime or the small virtual run time of the I/O bound processes, it is given a larger time slice to execute in the CPU. Thus we see the I/O bound and CPU bound processes are very well distinguished quite inherently by the CFS algorithm.

(Refer Slide Time: 13:50)



When a new process gets created, it gets added to the red black tree. Now its starts with a initial value of min_vruntime therefore, gets placed to the left most node of the tree and this ensures that it gets to execute very quickly. So, as it executes depending on the amount of time it executes whether it is an interactive or a CPU bound process, its position within the rb tree would vary. This was a brief introduction to the CFS scheduler, which is the default scheduler in current versions of the Linux kernel.

Thank you.