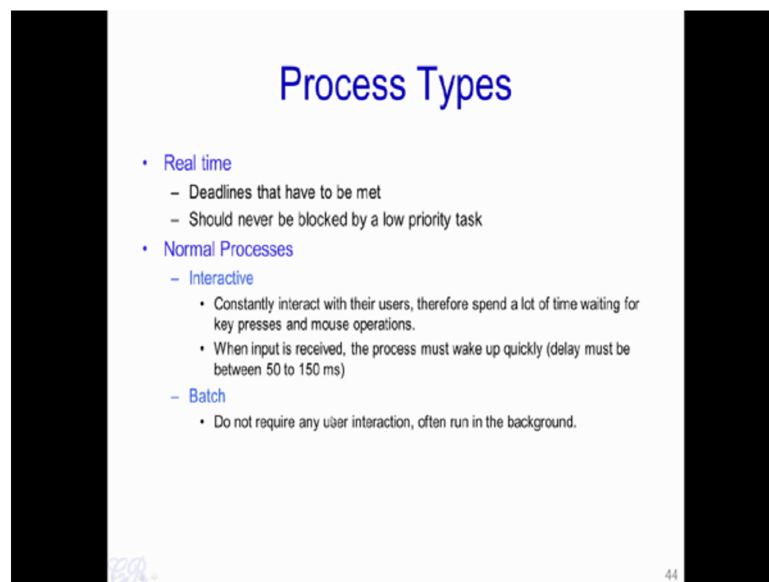
Introduction to Operating Systems
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

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 third Edition by Daniel Bovet and Marco Cesati.

(Refer Slide Time: 00:38)



Linux classifies processors into 2 types, one is the real time process and the other is normal processes. Essentially a process could be either real time or normal processes. 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.

Typically, you could consider these as IO bound processes when an input is received the process typically should be wake up with in 50 to 150 milliseconds. If it gets more than 150 seconds 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, the 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 have fall into these types of processors.

(Refer Slide Time: 02:15)

Process Types

- Real time
  - Deadlines that have to be met
  - Should never be blocked by a low
- Normal Processes
  - Interactive
    - Constantly interact with their users, therefore spend a lot of time waiting for key presses and mouse operations.
    - When input is received, the process must wake up quickly (delay must be between 50 to 150 ms)
  - Batch
    - Do not require any user interaction, often run in the background.

Once a process is specified real time, it is always considered a real time process

45

Once a process is specified as a real time process, it is always consider 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)

Process Types

- Real time
  - Deadlines that have to be met
  - Should never be blocked by a low
- Normal Processes
  - Interactive
    - Constantly interact with their users
key presses and mouse operations
    - When input is received, the proces
between 50 to 150 ms)
  - Batch
    - Do not require any user interaction,

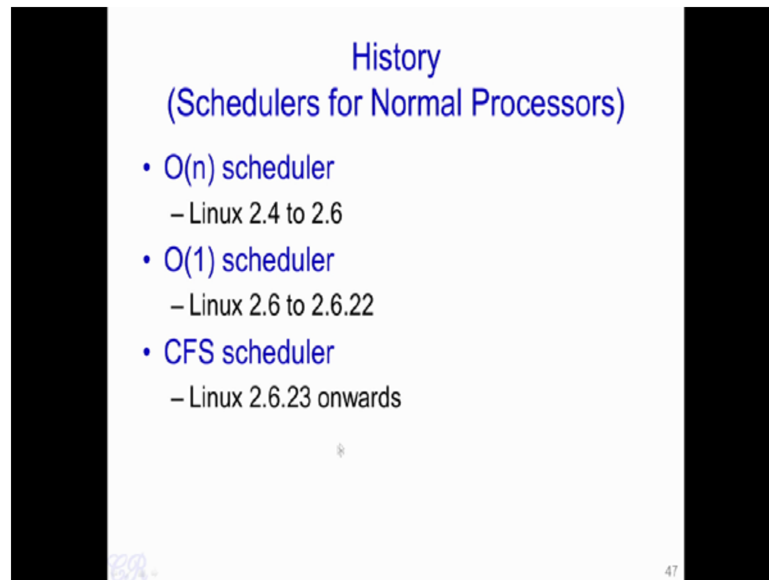
A process may act as an interactive process for some time and then become a batch process.

Linux uses sophisticated heuristics based on past behavior of the process to decide whether a given process should be considered interactive or batch

46

On the other hand, for processes which are non real time that is the normal processes. It can behave as either an interactive 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.

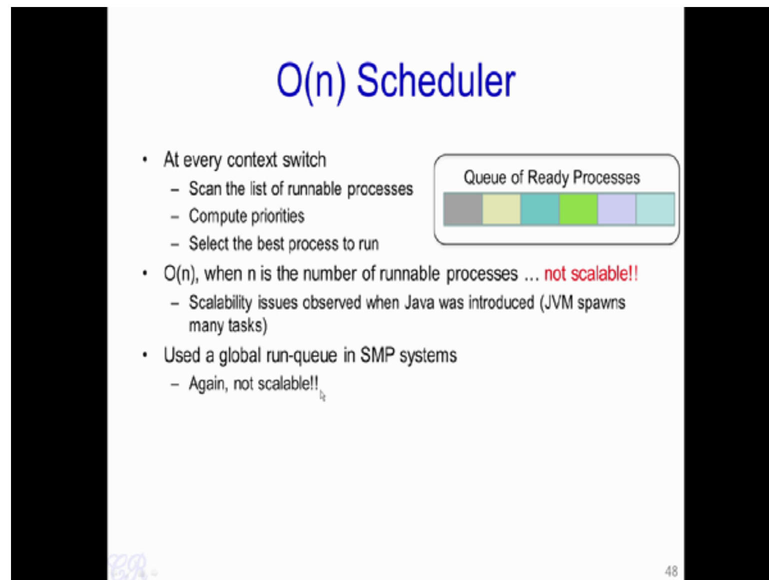
(Refer Slide Time: 03:22)



So, looking from a historic prospective, 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 CFS scheduler.

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

(Refer Slide Time: 03:54)



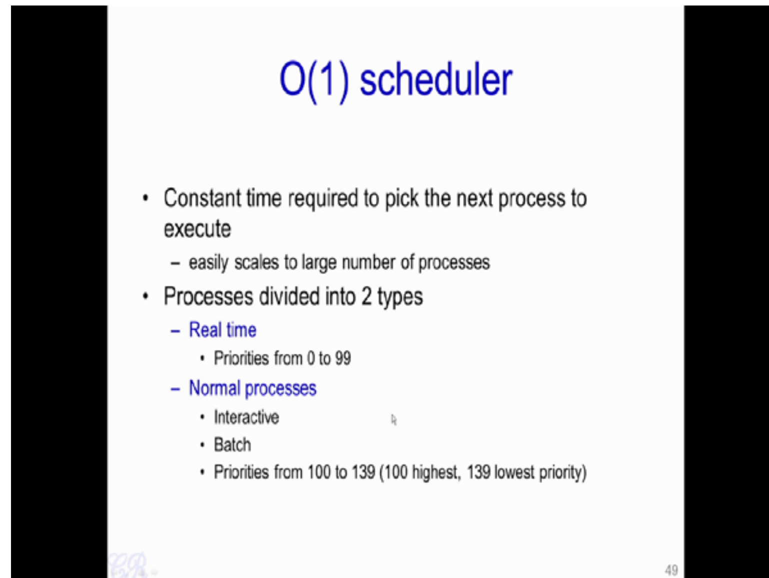
Let us start with the O(n) scheduler, O(n) scheduler means that the scheduler looks into the queue of ready processes which could be at most n size, and it takes into each of these processes and then makes a choice about the process that needs to execute. So, essentially at every context which the O(n)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

So, obviously, is not scalable it is not scalable because as n increases in number that is as more and more processes or task enter the ready queue, the time to make a context which increases. Essentially the context switch over heads increase and this is not something which is wanted for.

So, these scale 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 task 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 single global run queue for SMP systems essentially when you had multiprocessors system, as we seen in

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

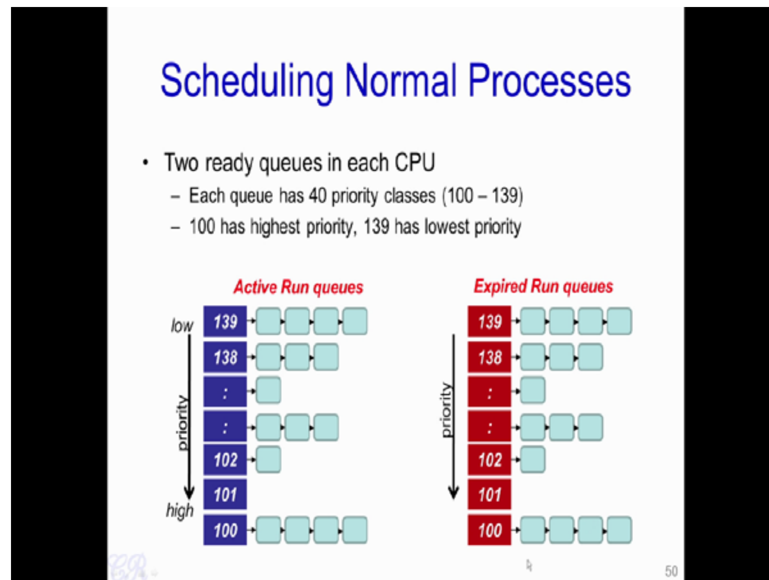
(Refer Slide Time: 05:48)



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(n) scheduler processes are divided into two types; the real time processes and normal time processes as we have seen. So, the real time processes are given priorities from 0 to 99 with 0 bring 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 either interactive or batch. We will see later how the scheduler distinguishes between the interactive and batch processes.

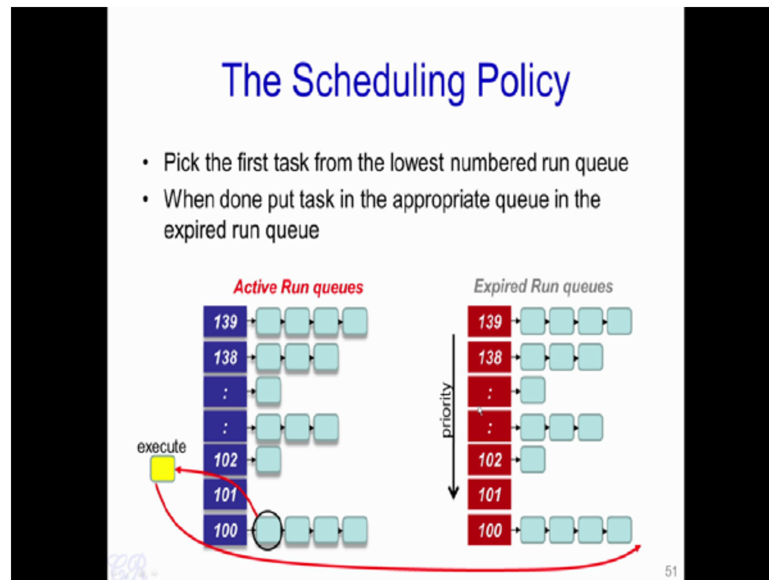
(Refer Slide Time: 07:00)



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 there 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.

So, corresponding to each of this task there is 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 hundredth 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 now based on this priority class, the scheduler maintains two 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)

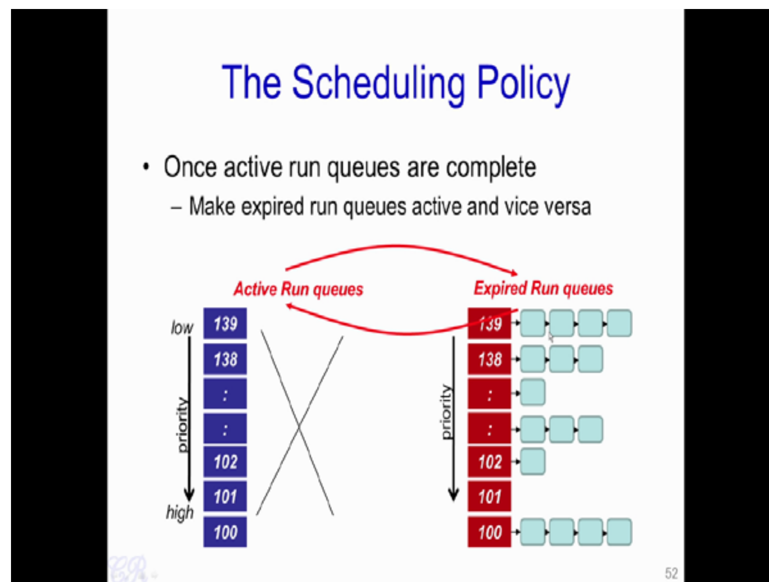


When a context which occurs, the scheduler would scan the active run queues, starting from the hundredth run queue and going up to 139 it picks the run queue which has a non zero or non empty queue present in that.

For instance over here, it starts from 100 and picks this first particular process 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 processors from the active run queues executes them and puts them into the expired run queues. After a while we would have 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 the scheduler would switch between the active run queues and the expired run queues. Now this is the queue which actually becomes active, while this becomes the expired run queues.

Now, for consecutive time slice completions, the scheduler would pick task from this particular queue and execute these task. After execution these task could be moved to this queue over here, 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 go goes on continuously, the main reason for having such 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)

contant time?

- There are 2 steps in the scheduling
  1. Find the lowest numbered queue with at least 1 task
  2. Choose the first task from that queue
- step 2 is obviously constant time
- Is step 1 contant time?
  - Store bitmap of run queues with non-zero entries
  - Use special instruction '*find-first-bit-set*'
    - *bsfl* on intel

53

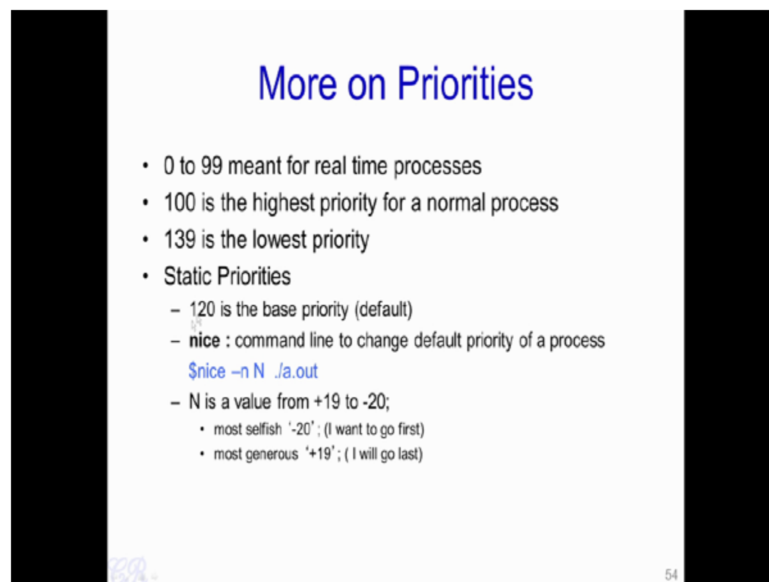
Recall that we call this particular scheduler as an O(1) scheduler, essentially this means it is constant time scheduler irrespective of the number of processes present in the ready queues.

Now, let us analyze how this particular thing is constant time, now if we recall there are two steps involved in scheduling a process. First is to find the lowest number queue with at least one 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 number 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 other to do this the scheduler or the Linux scheduler makes use of two things, the first is a bitmap this is a forty bit bitmap which

stores the run queues with non zero entities, essentially this particular bitmap stores 0 for a class which has which is empty, and it stores one for a class which has non 0 entities. Second in it uses a special instruction known as find first bit set. So, this particular instruction for example, there is a BSFL instruction on Intel would look at this forty bit bitmap and choose the lowest index, which is non 0 essentially this would give us the lowest numbered non 0 bit in the bitmap and this corresponds to the a lowest non 0 or other 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 have seen that a priority 0 to 99 I mean for real time task, 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 dot 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 one twenty to something else.

So, the command would look something like this way. So, nice minus n a capital n dot slash a dot out, where the capital n can take a value of plus 19 to minus 20. So, essentially the value that we specified here gets added to the base priority. So, for example, if I say nice minus n and plus 19 a dot out. So, the process that I am starting will have a priority 120 plus 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 minus n minus 20 dot slash a dot 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 it set based on whether the process has more of IO operations or is more CPU bound.

So, the scheduler uses some heuristics which is present over here to compute the dynamic priority, essentially it takes the static priority which we have set which 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 this that

is static priority minus bonus plus 5 and 139 and it then takes the max of 100 and this value. So, for instance let us say we start the process with the default static priority that is 120 and we give bonus of say 3, so then we have like 120 minus 3 plus 5 that is one 122. So, min of 122 and 139 is 122 and you take max of 100 and 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 processes is assumed to be more of a CPU bound process the dynamic priority is therefore, decrease essentially the dynamic priority goes towards 139 of a low priority task. 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 IO boundary process and essentially the process behaves more like an interactive process, the dynamic priority is therefore, increased that is it goes towards 100.

So, you can take two examples, for instance we have seen a bonus of three and a we had seen that it is resulted in dynamic priority of 122 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 it and a 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)

Dynamic Priority (setting the bonus)

- To distinguish between batch and interactive processes
- Based on average sleep time
  - An I/O bound process will sleep more therefore should get a higher priority
  - A CPU bound process will sleep less, therefore should get lower priority

dynamic priority = MAX(100, MIN(static priority – bonus + 5), 139))

heuristic

Average sleep time	Bonus
Greater than or equal to 0 but smaller than 100 ms	0
Greater than or equal to 100 ms but smaller than 200 ms	1
Greater than or equal to 200 ms but smaller than 300 ms	2
Greater than or equal to 300 ms but smaller than 400 ms	3
Greater than or equal to 400 ms but smaller than 500 ms	4
Greater than or equal to 500 ms but smaller than 600 ms	5
Greater than or equal to 600 ms but smaller than 700 ms	6
Greater than or equal to 700 ms but smaller than 800 ms	7
Greater than or equal to 800 ms but smaller than 900 ms	8
Greater than or equal to 900 ms but smaller than 1000 ms	9
1 second	10

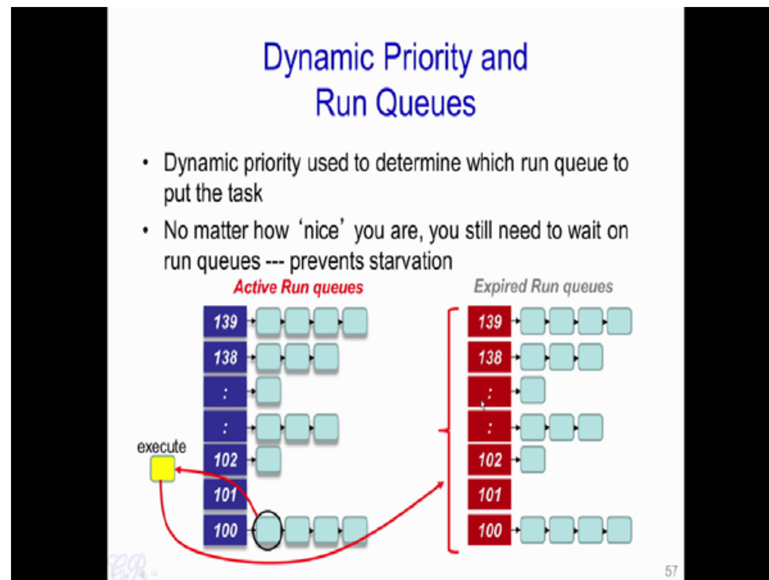
56

So, the bonus is in the scheduler is determined by the average leap time of the process. The assumption here is IO bound processes will sleep, or on other words will block more therefore, should have got a higher priority. So, therefore, IO 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 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 a 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 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 the dynamic priority determines which queue the particular process would be placed in. For instance when we choose particular process to execute, say from the hundredth priority class and after that it executes it is average sleep time is then used to compute the bonus, and thereafter to use to compute the dynamic priority, and then this process is placed in the corresponding priority class.

As 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 it is dynamic priority.

(Refer Slide Time: 20:28)

Setting the Timeslice

- Interactive processes have high priorities.
  - But likely to not complete their timeslice
  - Give it the largest timeslice to ensure that it completes its burst without being preempted. More heuristics

```
If priority < 120
    time slice = (140 – priority) * 20   milliseconds
else
    time slice = (140 – priority) * 5   milliseconds
```

Priority:	Static Pri	Niceness	Quantum
Highest	100	-20	800 ms
High	110	-10	600 ms
Normal	120	0	100 ms
Low	130	10	50 ms
Lowest	139	19	5 ms

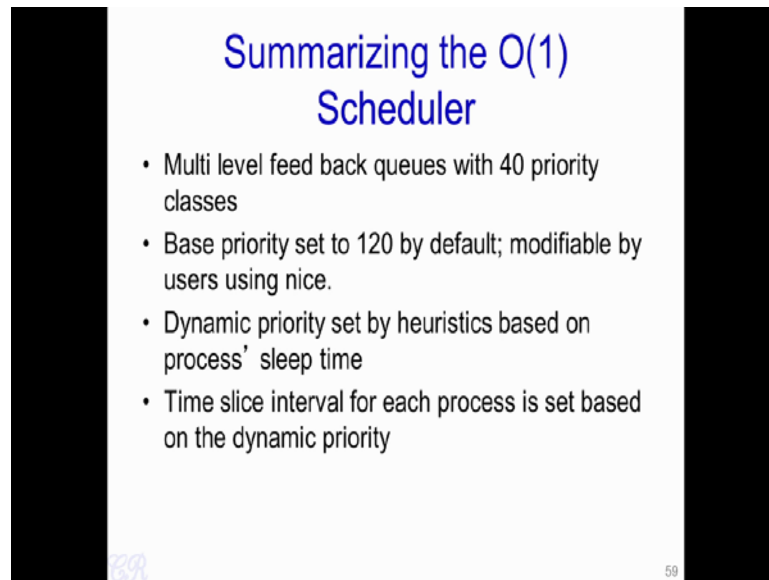
58

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 like likely to have short CPU burst times these processes essentially interactive processes are given the largest time slice to ensure that it completes it is burst without being pre empted. So, in order to set the time slice we are using more heuristics, such as this. Essentially if the priority is less than 120 that is it is an interactive process then the time slice is given as 140 minus priority into 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 that is time slice is equal to 140 minus priority into 5 milliseconds. So, as a result of this if you see this particular table, you see that if you have a static priority of 100 that is obtained by having a nice value of minus 20. So, it means that this is a interactive process and is given a large rime slice of 800 millisecond on the other hand, a process with static priority of 139 that is 120 plus having a nice value of 19 it is given the lowest time slice of the smallest time slice of 5 milliseconds.

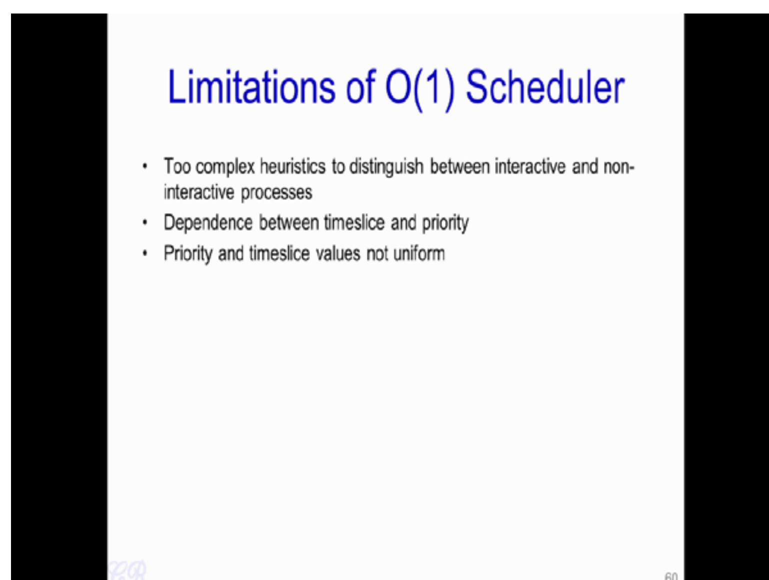
So, thus we see that based on the priority of the process the corresponding time slice given to that process is fixed.

(Refer Slide Time: 22:15)



Summarizing the O(n) scheduler it is a multi level feedback queue with for 40 priority classes. The base priority is set to 120, but modifiable by the use of this command called nice. The dynamic priority said 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)



The limitations of the O(1) scheduler is as follow; 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 with the processes with the priority of 139; if you compute the slices in these two ranges you see that there is it is not uniform.

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

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.