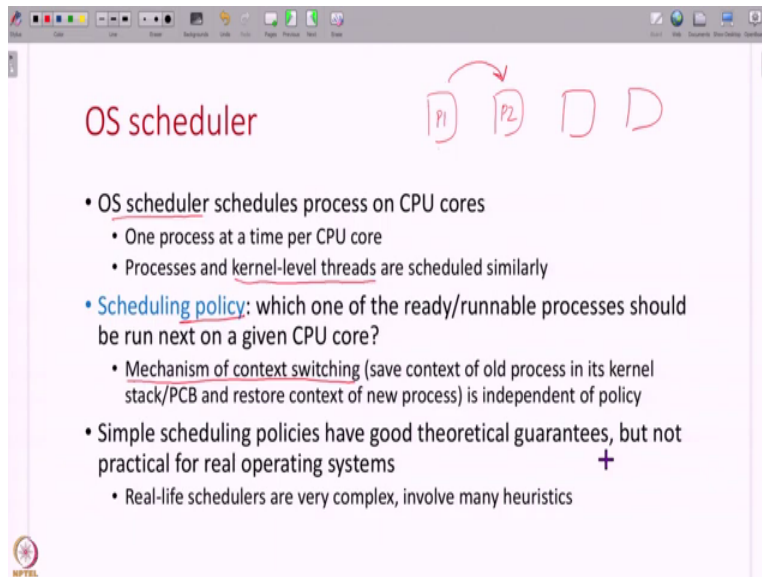


Design and Engineering of Computer Systems
Professor Mythilli Vutukuru
Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Lecture 09
CPU Scheduling Policies

Hello everyone, welcome to the 9th lecture in the course design and engineering of computer systems. In this lecture, we are going to study a little bit more about the CPU scheduler and what are scheduling policies. Let us get started.

(Refer Slide Time: 00:32)



The slide is titled "OS scheduler" in red. To the right of the title is a diagram showing four boxes labeled P1, P2, and two empty boxes. An arrow points from P1 to P2, indicating a sequence of execution. Below the title, there are several bullet points:

- OS scheduler schedules process on CPU cores
 - One process at a time per CPU core
 - Processes and kernel-level threads are scheduled similarly
- **Scheduling policy:** which one of the ready/runnable processes should be run next on a given CPU core?
 - Mechanism of context switching (save context of old process in its kernel stack/PCB and restore context of new process) is independent of policy
- Simple scheduling policies have good theoretical guarantees, but not practical for real operating systems
 - Real-life schedulers are very complex, involve many heuristics

A small NPTEL logo is visible in the bottom left corner of the slide.

So, we have seen this concept of the OS scheduler before. So, the scheduler decides which processes to run on a given CPU core. So, every CPU core can run one process at a time, but there could be multiple processes that are ready to run and from among these multiple processes, the CPU scheduler picks one of the process to run on a CPU core at a given point of time and after some time, it might run P1 for some time and then it might context switch to P2 and so on.

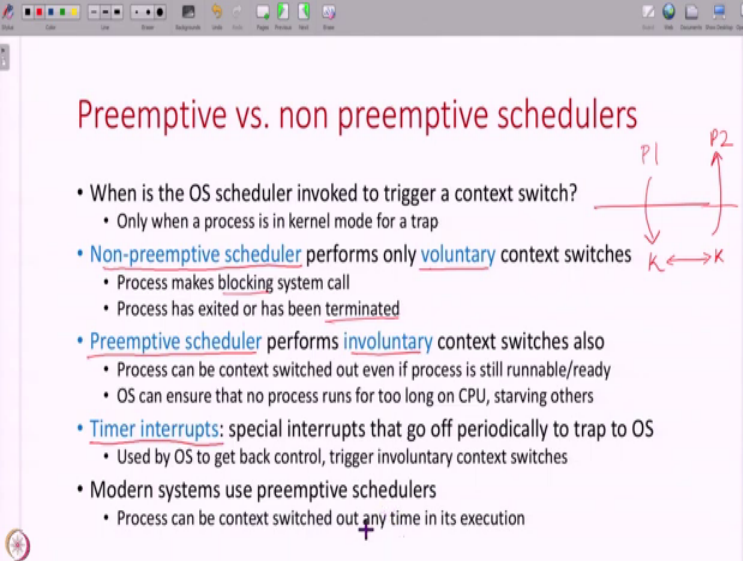
It can run these processes one after the other in some order. So, the scheduler schedules not just processes but also kernel level threads, we have studied threading in the previous lecture and these threads are also scheduled independently like processes by the OS scheduler. So, there are 2 parts to the scheduler, one is the policy which decides there are all of these ready processes to run which one should I run next on a given CPU core, that is the policy.

And once the policy decision is made, then you have the actual mechanism of the context switch itself, which is if I decide to stop running P1 and run process P2 next, then the context switch itself which is saving the context of P1 in its PCB restoring the context of P2 from its PCB, this mechanism itself we have seen it before.

In this lecture, what we are going to study is the policy that is the decision of which process to run. And there are many simple scheduling policies, we will start with studying some very simple scheduling policies that are easy to understand, that have good theoretical properties and so on. But if you look at what are the schedulers that are there in real operating systems, they are actually quite complex, they are not the simple scheduling policies we will study in this lecture.

So, in this lecture, I will start with the simple policies, but I will also try to give you a flavour for what are the real life schedulers looking like and what are some of these complex policies. But of course, understanding a full fledged real life scheduler is beyond the scope of this course.

(Refer Slide Time: 02:42)



Preemptive vs. non preemptive schedulers

- When is the OS scheduler invoked to trigger a context switch?
 - Only when a process is in kernel mode for a trap
- **Non-preemptive scheduler** performs only **voluntary** context switches
 - Process makes blocking system call
 - Process has exited or has been terminated
- **Preemptive scheduler** performs **involuntary** context switches also
 - Process can be context switched out even if process is still runnable/ready
 - OS can ensure that no process runs for too long on CPU, starving others
- **Timer interrupts**: special interrupts that go off periodically to trap to OS
 - Used by OS to get back control, trigger involuntary context switches
- Modern systems use preemptive schedulers
 - Process can be context switched out any time in its execution

So, let us get started with the simple policies and before we go to the policies, let us understand the types of schedulers. A scheduler is of 2 types, it can either be a pre-emptive scheduler or a non-pre-emptive scheduler. So, what are these differences? So, understand that the OS scheduler, when is it invoked for a context switch, there are different ways in which the OS scheduler can be invoked.

So, first of all, for the scheduler to run the process has to be in kernel mode and once a processes in kernel mode for a trap, the non-pre-emptive kind of schedulers what they do is, they only perform what are called voluntary context switches. That is, if a process P1 has come into kernel mode, it has trapped into the kernel and if it has made a blocking system call or it has terminated for some reason, then P1 cannot run any more.

Only in that case will a context switch be done to another process and you will return back to the user mode of another process P2. If P1 is able to run as long as P1 is able to run you will run P1 only when P1 does not want to run, if it makes a voluntary context switch only then the scheduler will switch to another process.

So, such schedulers are called non-pre-emptive scheduler, non-pre-emptive means, it will not kind of interrupt a process that wants to run. Other kinds of schedulers are pre-emptive schedulers, that is they also perform involuntary context switches that is even if this process is not blocked, has not terminated, can still continue to run for some more time, even then, some schedulers will stop this process into a context switch to another process. Such schedulers are called pre-emptive schedulers.

And such context switches are called involuntary context switches because a process has no control over it, it is still running. It is in the middle of doing something but it can still be context switched out. And these are needed modern operating systems do these involuntary context switches because you do not want any process to run for too long on the CPU and starve other processes, deprive other processes of their runtime.

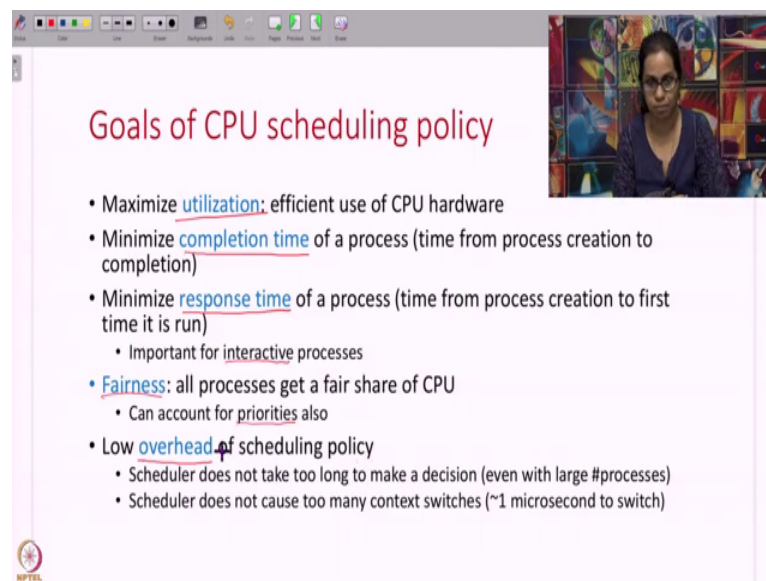
You do not want any process to take over hog the CPU for too long. Therefore, most modern operating systems use these kinds of pre-emptive schedulers and a lot of involuntary context switches are also performed. So, now the question might come up, how is a context switch event triggered? Why will a trap happen? If this process P1 is not making any system blocking system calls, is not terminating, is not giving up the CPU, then how will the process go into kernel mode to trigger this context switch.

For that purpose, modern CPUs have what are called timers, special piece of hardware that generates interrupts periodically. So, these timer interrupts go off periodically and every so often

a process will trap into the operating system. This will ensure that the operating system can set a timer on a process trapped to it and do an involuntary context switch.

So, timer interrupts are critical for pre-emptive schedulers. And most modern systems use pre-emptive schedulers because you want to share the CPU across multiple processes. You do not want any one process to run for too long. And because of this reason, because of the use of pre-emptive schedulers, process can be context switched out any time, these unfortunate context switches that lead to race conditions that we have seen in the previous lecture. These can happen with modern CPU schedulers.

(Refer Slide Time: 06:33)



Goals of CPU scheduling policy

- Maximize utilization: efficient use of CPU hardware
- Minimize completion time of a process (time from process creation to completion)
- Minimize response time of a process (time from process creation to first time it is run)
 - Important for interactive processes
- Fairness: all processes get a fair share of CPU
 - Can account for priorities also
- Low overhead of scheduling policy
 - Scheduler does not take too long to make a decision (even with large #processes)
 - Scheduler does not cause too many context switches (~1 microsecond to switch)

So, what are the goals of a scheduling policy? Before we study various policies, what do we want a good policy to do? We want a good policy to effectively use the CPU. A scheduling policy should not leave the CPU idle when there are processes to run that is inefficient. We want high CPU utilisation, we want processes to complete as fast as possible. We want to minimise the completion time of a process that is from the time a process is created to the time it ends.

We want this to be as fast as possible for as many processes as possible. We also want to minimise response time of a process. So, note that response time is different from completion time, what is the response time? The time from the process creation to the first time it is executed

on the CPU. So, when a process is created, when it is forked by the parent, it is added to some list of processes and at a later point of time, the CPU will run it.

So, this is the response time. This indicates for example, if you click on a programme, if you do some action, when will the action start to show up on the screen for you, that is the response time. So, even if you do not fully finish the process, if you at least schedule it once, give it some time to run, then the process will be responsive. This is very important for interactive processes.

If you are playing a computer game, you click on something you want the process to run and handle the event run the code corresponding to your click. Otherwise, you are going to see a lag when you interact with the process. So, response time is the time until you get the CPU for a short time, for the first time, that is response time completion time is for the entire process to complete these are two different things.

The other properties we want are fairness, if there are multiple processes in the system, you want all of them to get some fair share of the CPU. You can also prioritise, you can also set priorities for processes saying I want this process to get twice as much CPU as some other process that also you can do. But whatever it is, you should be able to control how much time each process gets on the CPU.

And finally, whatever is the scheduling policy, it should have low overhead, that is it should not take too long to decide if you have a large number of processes the scheduler itself should not run for a very long time trying to make this decision of which process to run next. It should quickly make the decision and it should not cause too many context switches because a context switch, you know saving context, restoring context, all of this itself takes time it can take up to like 1 microsecond, it can take like a few 1000 of CPU cycles.

And therefore, you do not want to have too many context switches. You do not want a scheduling policy that runs 1 process for a short time context switch, context switch, you do not want that. You want your overall scheduling policy to not add much overhead to the system.

(Refer Slide Time: 09:13)

Simplest policy: First In First Out $P_1 - P_2 - P_3 - \dots$

- Newly created processes are put in a **FIFO queue**, scheduler runs them one after another from queue
- Non-preemptive**: process allowed to run till it terminates or blocks
 - When process unblocks, the next run is separate "job", added to queue again
- Problem: short processes can get stuck behind big processes
 - Response time of interactive processes may be poor
- Example schedule: P_1 (1-5), P_2 (6-8), P_3 (9 to 10)

Process	CPU time needed (units)	Arrives at end of time unit
P1	5	0
P2	3	1
P3	2	3

The diagram shows a horizontal timeline with arrows indicating the execution of processes P1, P2, and P3 in sequence. P1 starts at time 0 and runs for 5 units. P2 starts at time 1 and runs for 3 units. P3 starts at time 3 and runs for 2 units. The timeline ends at time 10.

So, with this in mind, let us start understanding some simple scheduling policies. So, the simplest policy that you can think of is simple, first in first out or FIFO policy, what is this policy? Basically all the processes that are arriving in your system, put them in some sort of a queue. Whenever a process comes stick it into the queue and the scheduler will start picking processes one by one in the queue.

It will take this first process run it till it completes or terminates or blocks, then move on to the next process and the next process and so on. There is a simple queue in the order in which the processes arrive you will run them. And if a process runs for some time and blocks of course the next time it comes in it will be treated like a separate job. It will be added back to the queue again and once again it will get it start.

So, this is a simple process it is non pre-emptive, it is it let a process run for as long as it wants to and it is very easy to implement, but the problem is that sometimes short processes can stuck can get stuck behind some big processes suppose your P_1 is a very large process that takes a long time to run, then the other processes that come later will have to wait their turn for a very long time. This is not ideal, especially if you have interactive processes.

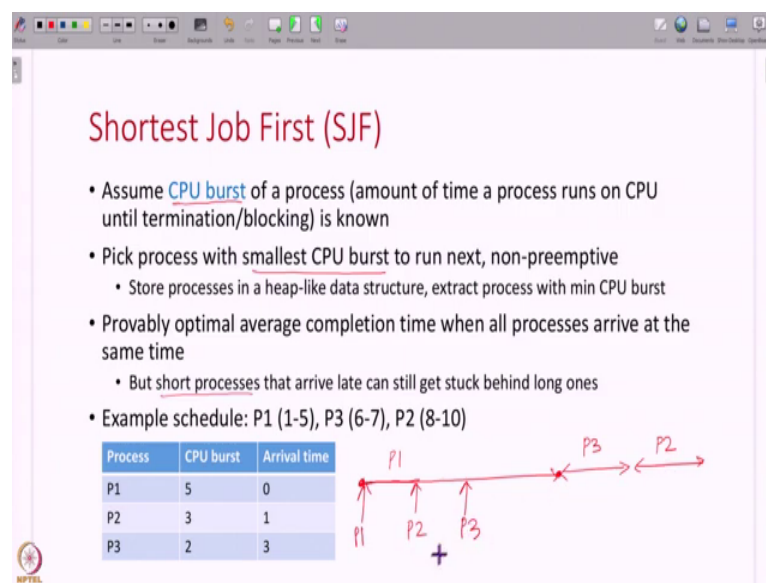
So, that is the inefficiency of this FIFO policy. So, let us just see a small example of FIFO. Suppose, you have 3 processes process, P_1 that runs for 5 time units and arrives that at the end of time interval 0 that is at just at the beginning of time slot 1. So, process P_1 arrives, and then

shortly after some time another process P2 arrives, and again at the end of time unit 3 we have process P3 arriving and this is our timeline.

So, now, what is the schedule that the FIFO scheduler will generate? So, when P1 comes FIFO scheduler of course, starts to run P1, P1 will run for its entire duration of 5 seconds, even though P2, P3 have come and they are shorter processes, it does not matter this is a FIFO scheduler P1 has come first therefore, it will run first.

And then after some time, after P1 ends, then P2 runs for the next 3 units P2 runs and then after some time towards the end P3 runs, this is the schedule. So, given the arrival times of processes and how long they will run given this information, you can create a schedule like this. This is an example schedule of the FIFO scheduler and this is fairly easy to understand.

(Refer Slide Time: 12:00)



Shortest Job First (SJF)

- Assume CPU burst of a process (amount of time a process runs on CPU until termination/blocking) is known
- Pick process with smallest CPU burst to run next, non-preemptive
 - Store processes in a heap-like data structure, extract process with min CPU burst
- Provably optimal average completion time when all processes arrive at the same time
 - But short processes that arrive late can still get stuck behind long ones
- Example schedule: P1 (1-5), P3 (6-7), P2 (8-10)

Process	CPU burst	Arrival time
P1	5	0
P2	3	1
P3	2	3

The Gantt chart shows a horizontal timeline. At time 0, P1 starts and runs until time 5. At time 1, P2 arrives but does not run. At time 3, P3 arrives but does not run. At time 5, P1 completes. At time 6, P3 starts and runs until time 7. At time 8, P2 starts and runs until time 10. The processes are labeled above the timeline: P1, P2, P3, P3, P2. A small '+' sign is marked below the timeline at time 3.

Now, let us move on to slightly more complex scheduling policies. Another popular though a very theoretical scheduling policy is what is called the shortest job first. In this scheduling policy, you assume that you know how long a process runs. In FIFO you need not know how long the process had to run. You just let it run as long as it wants to.

But here in shortest job first you assume that what is called the CPU burst of a process that is the amount of time a process runs in one instance, when it is given the CPU until it terminates or blocks that is called the CPU burst of a process and you assume that the CPU burst of processes known. And this scheduler will pick the process with the smallest CPU burst to run.

Amongst all the processes that are there in the system right now, you will pick the process with the smallest CPU burst. Maybe you store all the processes in a heap like data structure and you extract the process with the minimum CPU burst. You can decide what data structure to use. So, you will always pick the process with the smallest CPU burst, but this is non pre-emptive.

Of course, when a process is running, if another process with a shorter CPU burst comes in, it will not pre-empt the running process. It is a non-pre-emptive policy. And this policy you can prove, if you take a course, theory course, you can actually prove that this is optimal and this will actually minimise the average completion time of all processes when all the processes arrive at the same time, under certain conditions, this policy is actually going to work very well.

But you still have the problem that a short process can get stuck behind a long process, if it arrives slightly late. You have ideally you want to let the shorter processes run first, but if the short process comes slightly late, and you have already started a long process, then this is a non-pre-emptive policy, it will not pre-empt the running process.

Once again, let us take our same example of P1, P2, P3 that we have seen in the previous slide. And let us try to work out what the schedule will be with the shortest job first. So, you have process P1 has arrived and it has started to run and it runs for the entire 5 units. Even though P2 P3 are arriving and they are shorter than P1.

Once P1 has started, when P1 started it was the shortest job and just because these other processes arrived, we are not pre-empting P1 we are not stopping P1 therefore P1 will continue to run for its entire duration of 5 time units. Now, at this point after P1 finishes, then, what do you do? At this point, you have both P2 and P3, but P3 is the shorter process.

Therefore here is the difference from FIFO. FIFO would have run P2, but the shortest job first will now run P3 and then it will run process P2 at the end. This is the schedule for shortest job first. But as you see, we wanted to prioritise short processes get them done quickly, but that may not always happen, especially if these processes the short processes arrive a little late.

(Refer Slide Time: 15:11)

Shortest Remaining Time First (SRTF)

- Preemptive version of SJF
- A newly arrived process can preempt a running process, if CPU burst of new process is shorter than remaining time of running process
 - Avoids problem of short process getting stuck behind long one
- Example schedule: P1 runs for 1 unit, P2 (2-4), P3 (5-6), P1 (7-10)

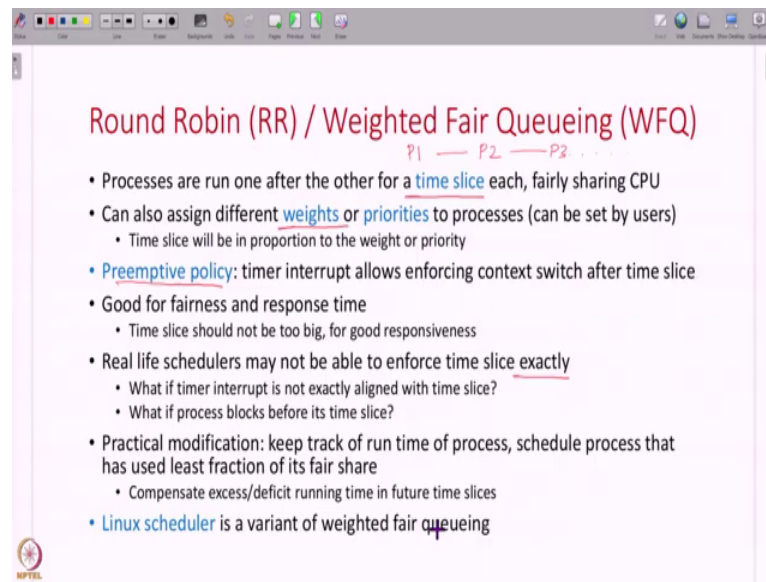
Process	CPU burst	Arrival time
P1	5	0
P2	3	1
P3	2	3

So, therefore, the improvement on this is a pre-emptive version of shortest job first that is called shortest remaining time first. That is when a process arrives and if CPU burst is shorter than the remaining time of the current process, then it will pre-empt the current process. If a shorter process comes, it can pre-empt the currently running longer process thereby giving priority to shorter processes.

And this avoids the problem of short interactive processes getting stuck behind long processes. So, once again the same example let us see and let us see what happens with this pre-emptive shortest remaining time first. So, process P1 has run for 1 time unit, it has arrived and it has run. Now, after 1 time unit we have process P2 is arriving. Now, at this point P2 has a CPU burst of 3 units and P1 has run for 1 unit and it has 4 more units left. So, therefore, this is shorter than this therefore, P1 is pre-empted and you will run P2.

Now, while P2 has run for 2 units, after 2 more units after P2 we have P3 arriving. But when P3 is arriving, P2 only has 1 unit of work left and P3 has 2 units of work. This 2 units of P2 are done, there is only 1 unit left and P3 has 2 units of work. Therefore, P3 will not pre-empt P2 we will let P2 continue after P2 finishes. Now, P1 has 4 units left and P3 has 2 units left. Therefore P3 is the shorter one therefore, we will run P3 and finally at the end, this long P1 process will complete. So, this is the schedule that you will have with shortest remaining time first.

(Refer Slide Time: 17:08)



Round Robin (RR) / Weighted Fair Queueing (WFQ)

$P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow \dots$

- Processes are run one after the other for a time slice each, fairly sharing CPU
- Can also assign different weights or priorities to processes (can be set by users)
 - Time slice will be in proportion to the weight or priority
- Preemptive policy: timer interrupt allows enforcing context switch after time slice
- Good for fairness and response time
 - Time slice should not be too big, for good responsiveness
- Real life schedulers may not be able to enforce time slice exactly
 - What if timer interrupt is not exactly aligned with time slice?
 - What if process blocks before its time slice?
- Practical modification: keep track of run time of process, schedule process that has used least fraction of its fair share
 - Compensate excess/deficit running time in future time slices
- Linux scheduler is a variant of weighted fair queueing

Now, all of these are process scheduling policies that assume that you know the runtime of a process and so on. But in real life, this may not be possible when you start when you fork a process when you run your A dot out programme you do not know how long it is going to run for. So, therefore, real life operating systems cannot make this assumption.

So, now we will discuss a few scheduling policies that real schedulers can use. Another simple policy that is without all of these assumptions of knowing the CPU burst is what is called a Round Robin or a fair queueing and there is also a variant of it called weighted fair queueing. So, what is Round Robin? It is simple. You run the processes in a Round Robin fashion one after the other for a time slice each.

You run process P1 for some duration, a few time units, then you run P2, then you run P3, you go through all your processes in a Round Robin fashion, once you reach the end of your list, then you come back again run P1 again for another time slice, maybe you run P1 for 10 milliseconds, go to P2 10 milliseconds, 10 milliseconds go do this all the time, then come back again to P1.

So, this is a simple Round Robin or a fair queueing policy that is fairly sharing the CPU across all the processes in the system. And you can also have a weighted version you can have a weighted Round Robin or a weighted fair queueing where you can assign weights or priorities to these processes. You can say process P1 is twice as important as process P2, therefore, I will run P1 for 20 milliseconds and I will run P2 for 10 millisecond, P3 for 10 millisecond and so on.

You can do this weighted fair share across different processes and where the time slice will be in proportion to the weight or the priority. And this is a pre-emptive policy, obviously, at the end of a time slice, you are going to pre-empt this process, you can use the timer interrupt to go off at the end of the time slice and you are going to stop this process move on to the other process, even if this process is still ready to run. Therefore, this is a pre-emptive policy.

And this is good for fairness and response time, every process will get its turn pretty soon. You are not waiting for a process to finish or and keep other processes waiting for a long time you are not doing that. And especially if your time slice is small enough, this scheduling policy has a very good response time the processes will respond will run at least for a short period of time very quickly.

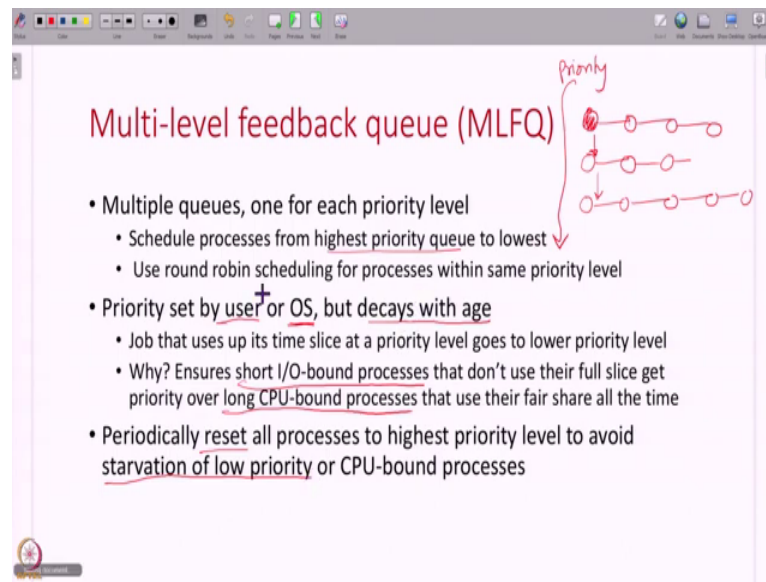
And so if you have a real life scheduler, you may not be able to enforce this time slice very exactly. If you say 10 millisecond, maybe the timer interrupt when it goes off already 11 milliseconds have passed or maybe just before the time slice ends at 7 milliseconds the process has made a blocking system call. Real life processes, you may not be able to exactly enforce this time slice.

So, what real life schedulers will do is they will just adjust this excess or shortfall of the running time in future time slices, so you will simply keep track of how long the process has run, you will try to enforce a time slice approximately. But you will simply keep track of how long a process has run, and you will schedule the process that has used the least fraction of its fair share.

So, suppose a process has run over its fair share and slightly use more than its fair share in this round. In the next round, you will deprioritize it slightly. On the other hand, if a process did not use its fair share in this round in its next round, it might get priority. So, you will pick a process that has used the smallest fraction of its fair share so far.

In this way you can compensate for slight overshooting undershooting of the time slice. So, this is a practical modification given that in real life you may not exactly be able to enforce the time slice and the Linux scheduler policy in the latest Linux kernel is a variant of this weighted fair queuing, of course, it is very complex, it has many other features in it, but at the very basic level, this is the simple idea behind the Linux scheduler.

(Refer Slide Time: 21:27)



Multi-level feedback queue (MLFQ)

- Multiple queues, one for each priority level
 - Schedule processes from highest priority queue to lowest
 - Use round robin scheduling for processes within same priority level
- Priority set by user or OS, but decays with age
 - Job that uses up its time slice at a priority level goes to lower priority level
 - Why? Ensures short I/O-bound processes that don't use their full slice get priority over long CPU-bound processes that use their fair share all the time
- Periodically reset all processes to highest priority level to avoid starvation of low priority or CPU-bound processes

So, one other policy once again which is suited for real life implementation is what is called a multi-level feedback queue. So, here instead of Round Robin where you just had one list of processes in multi level feedback, you what you do is you maintain multiple queues of processes. You can have this as one queue of high priority process, then all the medium priority process you have another queue and the low priority process you have another queue in this way you have multiple priority levels and at each priority level, you will keep a separate queue of processes.

And you will schedule processes always starting from the highest priority level. If the scheduler has to run it first go to the highest priority level run this process, run this process and at the same priority level processes of same priority you can use something like Round Robin. So, the scheduler always starts at the highest priority level, runs all these processes in a Round Robin fashion, then it moves on to the next priority level, then to the next priority level and so on.

So, this is a different way of running processes from the Round Robin. And what is this priority? It can be set by the user or it can be adjusted by the scheduler itself. For example, the scheduler can do something like it can decays the priority with age, that is if a process has run at a high priority level for some time, and it has fully run for its time slice at this high priority level, then you can the next round, push it to a lower priority level.

Why do you want to do that? This is to ensure that short processes, if you have short IO bound processes that is if a process just runs for a small time goes waits for some IO disk operation to

happen again runs for a short time again goes away. So, you want to prioritise such processes, this guy is only coming to you for a very short time. So, you might as well just run him quickly.

Then again, he will go back wait for some IO operation. So, such processes, you want to prioritise them over processes that run on the CPU for a very long time, over long CPU bound processes, these will always run on the CPU for a very long time. Therefore, whenever you have a short IO bound process come in, you want to quickly run that guy, because anyway, he has to wait for IO later on.

So, how do you prioritise such processes, what you do is if any process did not use up its full time slice, It only ran for a short period of time, then you let it remain at this priority level. On the other hand, if it used up its entire time slice you move it to a lower priority level. So, with time processes that are hogging the CPU, that are taking up a large share of the CPU all the time, these processes will move down the priority level.

Whereas processes that are running for a very short period of time IO bound processes will run at a high priority level. So, this is an small heuristic that is used to ensure that IO bound processes, which only run for short periods of time they are given preference. And of course, you do not want to give always preference to these IO bound processes.

You do not want a CPU bound process to always be stuck at the lowest level and never complete. You know if a CPU bound process is at the lowest priority level and IO bound processes are always coming in, the scheduler is always just running one of these processes and never coming here you do not want that.

So, periodically in such algorithms which maintains strict priority levels. Periodically what you will do is, you will reset all the processes to the highest priority levels. So, everybody gets a fair chance to compete once in a while. So, this is to avoid starvation of low priority processes. So, this is another example of a slightly realistic complex scheduler algorithm.

(Refer Slide Time: 25:14)

Multicore scheduling

- Scheduling decision needs to be made separately for each CPU core
- Do we bind a process to a particular CPU core always, or do we let a process run on any CPU core that is free?
 - Is the queue of ready processes common to all cores, or maintained per core?
- Ensuring a process runs on the same core as far as possible is better
 - Cache locality: process-related memory is likely to be in CPU caches of the core
 - In NUMA systems, better to run process on core that is close to the RAM region that has process memory
 - Per-core queue of ready processes avoids synchronization across cores
- But, we must be flexible too
 - If CPU core overloaded, some of its processes must move to another core
 - Load balancing across cores to ensure uniform workload distribution

Handwritten notes: 'Cores' with arrows pointing to core icons; 'Core L1 L2 L3' with arrows pointing to cache levels; 'Per-core queue' underlined.

So, now finally, the last thing I want to talk to you about with scheduler is multi core scheduling. So, all of these scheduler policies pick a process to run on one CPU core. Now, what if you have multiple CPU cores? You have to schedule processes on the multiple CPU cores independently. So, there are different ways of doing it. If you have these multiple CPU cores, what you can do is you can just maintain a common queue of processes P1, P2 and so on.

And whenever a CPU core becomes free from this common queue, you can schedule processes okay process P1 you go here, this is like you are waiting at some counter in a single file and whichever counter becomes free you will go there. So, the core is free you run here oh no the core has become free the next process will go here. You can keep doing this.

This is any process basically can run at any CPU core that is free, this is one way of doing it. The other way of doing it is you somehow assign processes to cores. You have separate queues, these processes will run on the core, these processes will run on the core and so on. You have separate queues waiting for each core.

And whenever, so, first, this process will run on the core when its turn is done the next process the next process and so on. So, you can bind the process to a particular CPU core and run it on the core always or you can let any process run on any CPU core. Accordingly, you will maintain this data structures of, this list this queue of ready processes, it can be a common queue or it can be a per-core queue.

So, what are the pros and cons of both these approaches both these are possible, but they have their own advantages and disadvantages. So, ensuring that a process runs on the same core as far as possible, this policy is better for the following reasons. Why? You will have cache locality, that is when a process is running on a CPU core recollect that a CPU core has multiple layers of cache that are private to itself and you only have the last level cache that is common across different CPU cores.

Therefore, if a process is running on a CPU core, all of its some of its core data could be in the private caches of a CPU core. So, therefore, it is simply more efficient and faster to let a process also just resume on the same core again in the future. Because then you will get good cache locality you will get good hit rate in the cache.

The other reason is that you have some kind of NUMA systems, which is in some systems we have discussed this along back in some systems, some memory is closer to some cores. You have a large number of cores and you have different RAM and for these cores, it is faster to access this RAM for the CPU cores it is faster to access this RAM.

So, in such systems, what do you do, if the memory image for processes in this RAM, it is better to schedule the process in the cores, if the memory image of the processes in this RAM it is better to schedule the process on the cores. So, this is called NUMA aware scheduling. In such cases, a process that is there in this area of the RAM, even if the CPU core is free, you may not want to run it there because accessing this memory is very slow.

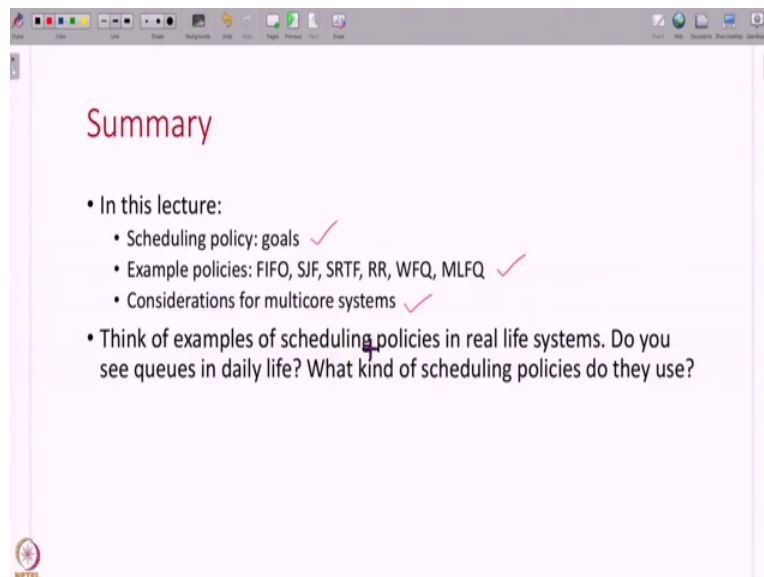
So, this is the other reason why you might want to just pin processes to one CPU core or a set of CPU cores. And another reason is that if you have these per-core queues, it avoids synchronisation. So, any time different CPU cores have to access a common data structure, there will be overheads like acquiring locks, there will be cache coherence, we have studied all of these things.

In general, if a CPU core is accessing one item of data, it is better if other CPU cores also do not access the same item of data, to a wide this synchronisation across CPU cores also, having per core queues is better. So, for all of these reasons, it might seem like a good idea to just allocate processes to CPU cores and CPU whenever it is free, it will pick from its own queue. But the disadvantage of doing that is this is not flexible.

If one core is overloaded this CPU has many processes that are taking a lot of time and some other CPU is just free, that is not sensible, that does not make sense. So, in such cases, what do you want to do? You want the CPU core to take some of the work from the overloaded core. Therefore, if you are doing per core queues of ready processes, then you must have some way for load balancing across these cores, to ensure some uniform distribution of workload.

So, these are all the things to keep in mind when you are thinking of a multi core scheduling algorithm. So, most of the schedulers and real operating systems have to consider all of these points, because most systems today are multi core systems and all scheduling algorithms have to solve these issues around multi core scheduling. So, that is all we have for today's lecture.

(Refer Slide Time: 30:31)



To summarise, in this lecture, we have studied, what is the scheduling policy, what are its goals. We have seen some examples scheduling policies starting from very simple ones on to more complex realistic ones and we have also seen what should a scheduler do in case of multi core systems. So, a small exercise for you just think of examples of real life schedulers.

In real life, if you have a counter and you have some queues and you know, any time any system in real life, where you have some scheduling going on, just think of what is the kind of scheduling policies that are used in real life.

For example, if you submit some applications to your office in a college, do they look at the applications and pick the easiest applications that take the smallest amount of time, do they

process them first, do the process them in the order in which they came in. So, just look around in real life and try to find examples of the scheduling policies we have studied today. Thank you all. That is all I have for this lecture and we will meet back again in the next lecture. Thanks.