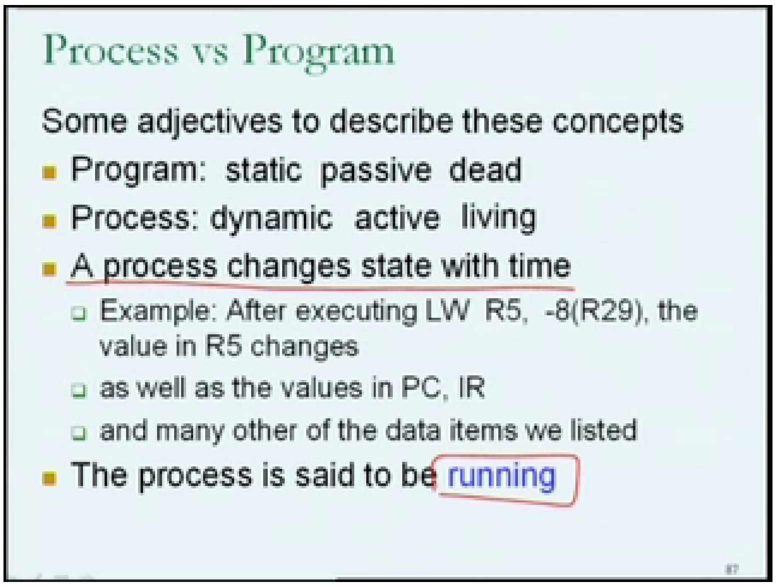
# High Performance Computing

**Prof. Matthew Jacob**
**Department of Computer Science and Automation**
**Indian Institute of Science, Bangalore**

**Module No. # 04**
**Lecture No. # 17**

Welcome to the lecture number seventeen, of our course on, “high performance computing”. We are currently looking, at the functionality, of the operating system, which we are going to call as process management. It relates to how the operating system, shares the C.P.U time, of a computer system, among the many programs in execution.

(Refer Slide Time: 00:38)



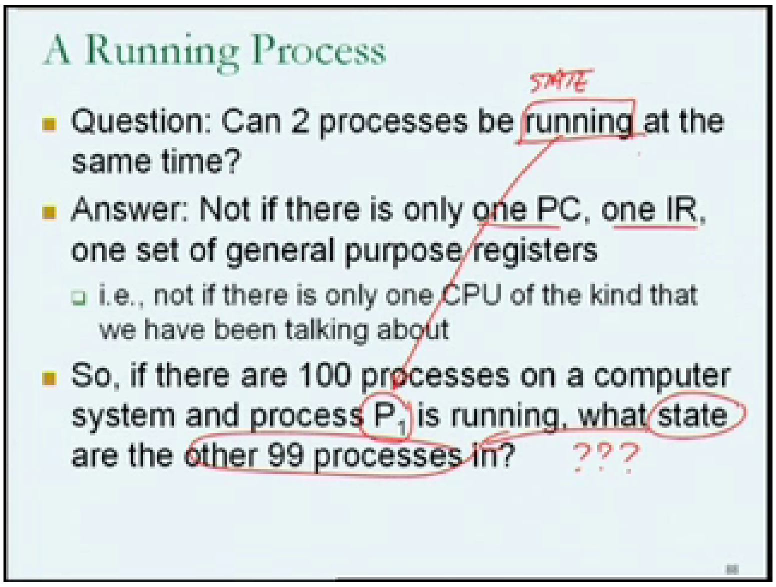
## Process vs Program

Some adjectives to describe these concepts

- Program: static passive dead
- Process: dynamic active living
- A process changes state with time
  - Example: After executing LW R5, -8(R29), the value in R5 changes
  - as well as the values in PC, IR
  - and many other of the data items we listed
- The process is said to be running

Now, in the previous lecture, we had ended the lecture, by trying to put down our thoughts, about what a good perspective on how to view what a process is. We did this by using adjectives, to contrast a process from the program that it may have come from. And we came up with the idea that processes are active entities, which change state with time, and that when a process is in a condition of being on the C.P.U, and executing instructions, we could refer to that process as being running. Right, this has in a sense that and we look into this concept a little bit more.

(Refer Slide Time: 01:13)



## A Running Process

- Question: Can 2 processes be running at the same time?
- Answer: Not if there is only one PC, one IR, one set of general purpose registers
  - i.e., not if there is only one CPU of the kind that we have been talking about
- So, if there are 100 processes on a computer system and process $P_1$ is running, what state are the other 99 processes in?

So, this concept of, “A Running Process”, so you recall that what we have said is, “When a process is in the CPU and its instructions are being executed, then it is changing its state”. It is changing the state of its variables; it is changing the state of text, data, stack, and heap; may be not text, but data, stack and heap, registers, page tables etc.

Now, a question which immediately arises is, “Can two processes, be in that condition at the same time?” In other words, can two processes be running at the same time? I am using the word running, in the sense we have defined it, in the previous slide (Refer Slide Time: 00:55). It is a technical term. A processes, is in the state running, if it is in the CPU, and it is causing the execution of instructions through which its different data; different data associated with it, makes it modified.

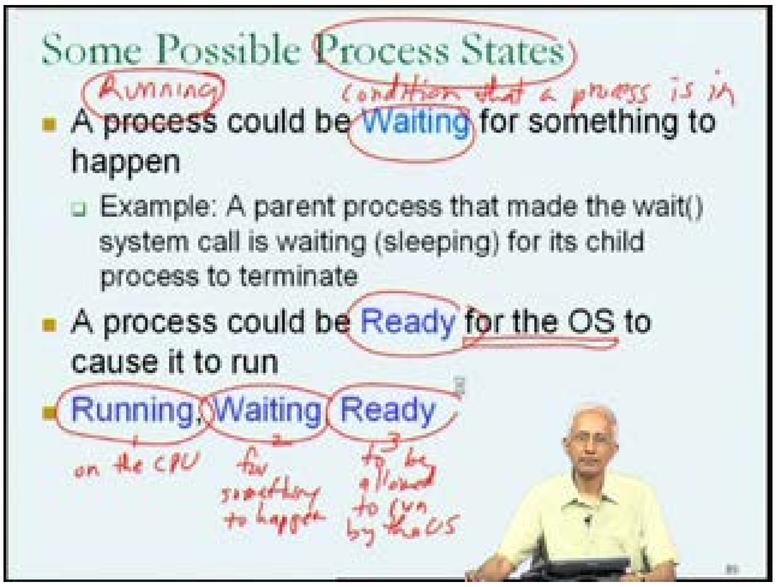
Now, the answer to this question should be very obvious, as we have talked about this before. The answer is no, and clearly it cannot be the case. You cannot have two processes running, if your system has only one program counter; one instruction register and one set of general-purpose registers. Because, for a program, for a process to be running, it has to be in a condition where its instructions are being executed.

And if my computer system has only one program counter, one instruction register, and one set of general-purpose registers, then only one process can be running at a time. Later on, we may learn about computer systems, which have more than one program counter, etcetera. For the moment, our picture of a computer system is one, where there is one CPU and in that CPU, there is one set of special purpose registers and one set of general purpose registers. For the answer to the question is definitely no. I cannot have two processes running at the same time. They could be utmost one process running at a time.

So, the answer is that for the kinds of CPU's that we have been talking about so far, they can be at most, only one process running at the time. In other words, going back to that hypothetical situation, which I often use, that there are 100 processes, on a computer system, then if only one of them can be running, let us assume the process that is running is process P1(Refer Slide Time: 03:21). That raises the question of what are the other 99 processes are doing. What is their condition? What is their state?

So, at this point, we seem to be talking about running, as being the condition of process P1, we can think of running as being the state, or the condition in which process P1 is. And we are asking the question that what is the state of the other 99 processes? Given that there are 100 processes on the system. After this point in time, we have seen only one state; a process could be in, the state of being in the C.P.U, its instructions executing, which data changing. So, we need to learn about other states that a process could be in.

(Refer Slide Time: 04:02)



## Some Possible Process States

condition that a process is in

Running

- A process could be Waiting for something to happen
  - Example: A parent process that made the wait() system call is waiting (sleeping) for its child process to terminate
- A process could be Ready for the OS to cause it to run
- Running, Waiting, Ready

1 on the CPU

2 for something to happen

3 to be allowed to run by the OS

So, what are some other possible process states? This is what we need to get our handle on next. One thing, which we have seen, is that the process could be waiting for something to happen. We saw this for example, in the case of the parent process, which could be sleeping, waiting for its child to terminate. And it could come into this state by executing the wait system call. So, this seems to be another condition that a processor could be in.

Remember! We are going to use the word, "Process State", to refer to have a high level the condition that a process could be in. So, one possible condition was that the process could be running, but at any given point in time, at most, one process can be running. What condition could the other processes in wait could be in? One answer is, some of them could be waiting for something to happen, such as, a parent process waiting for its child to terminate and there could be other such situations. So, we will assume that this is a common problem with a common condition the processes could be in. So the example of a parent process that made the wait system calls, and is waiting or sleeping for its child process to terminate (Refer Slide Time: 05:07).

Now, after the child process terminates, what happens? The answer is that the parent process may not actually be able to start running right away, because there could be some other process running. And therefore, we need to talk about, another state that a process could be in. In other words, another condition that the process could be in where it is not waiting for anything to happen. On the other hand, it is not running either, so there is a third possibility and I will refer to that third possibility. Second possibility was, “Waiting”, the first possibility was “Running”, and the third possibility is what I shall refer to as “Ready” (Refer Slide Time: 05:51).

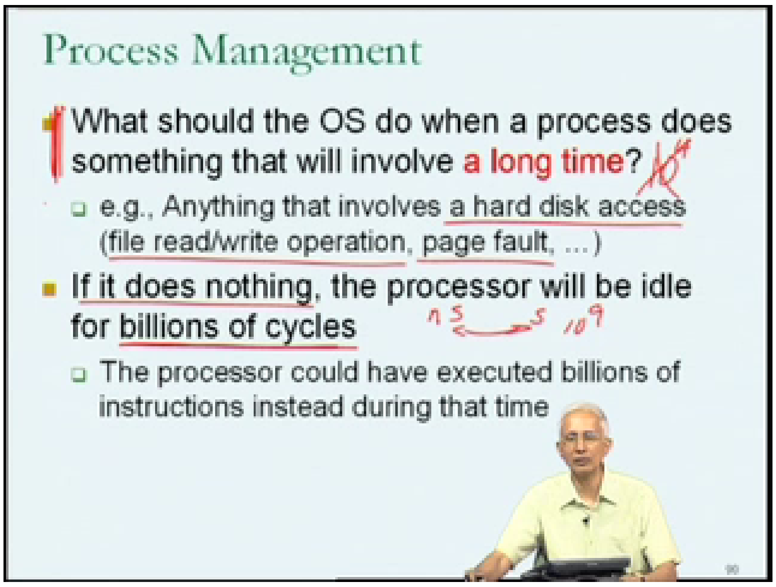
So, a process, which is not waiting for anything but is not running, would be described as being ready. What it is ready for? It is ready to be allowed (Refer Time: 06:00) to run, or its ready for the operating system to cause it to run. I am laying the blame at the door of the operating system, since currently, we are talking about the different jobs an operating system does for processes. And we are actually talking about the job of sharing CPU time or managing processes and therefore, we understand that this is a responsibility of the operating system, not of the individual user program or of the user.

So, in short this is a possible third state. A state where a process is not waiting, it is not running, but it is essentially ready to be running and is just the question of the operating system allowing it, or causing it to run.

In short, we could talk about the three possible states or conditions that a process could be in and I will refer to these three possible conditions as, “Running, Waiting and Ready”, (Refer Slide Time: 06:45) ready to run, waiting for something to happen, and running on the CPU that may be a more complete way to look at it, ready to be, allowed to run, by the operating system.

So, there are these three possible states we could think of. Now, before I proceed, I have been talking about the idea that at most one process could be running at a time. Therefore, it is possible that at some points in time, there are no processes running in which case that is something that we need to also think about later, but for the moment, we will assume that they would be one process running, some number of processes waiting and some number of processes ready.

(Refer Slide Time: 07:42)



## Process Management

- What should the OS do when a process does something that will involve a long time? $10^4$
  - e.g., Anything that involves a hard disk access (file read/write operation, page fault, ...)
- If it does nothing, the processor will be idle for billions of cycles $ns \rightarrow s$ $10^9$
  - The processor could have executed billions of instructions instead during that time

So, the operating system job of managing all these processes is what we know as, Process management, and from the perspective of the operating system developer, the one kind of question which might arise, is this question. This is an operating system design question, but we will ask the question and look at the kinds of ways that operating systems answer this question, in order to understand, what may happen to the programs that we execute. A question is, "what should the operating system do when a process does something that will involve a long time?" Now, I do need to explain what I mean by a long time, so let me give you an example.

Now, let us suppose that the operating the process does something, which results in a hard disk access (Refer Slide Time: 08:33). What are the different things we have seen that a process could do that would result in a hard disk access? The answer is, it might be read or write operation on a file, if the hard disk is implemented, on a… if that file is implemented on a hard disk.

We have also seen that process may make a memory reference which results in a page fault, which may also result in a hard disk access, and why do I associate a long time with the hard disk access, because in the previous lecture when we were talking about, the speed disparity between the main memory and the hard disk, we actually caught this ballpark, this approximate idea that the hard disk could be 10 to the power of 4 times slower than memory.

So, if memory is operating in terms of 100 of nanoseconds or 100 nanoseconds, the hard disk is operating, in terms of milliseconds, or their about, possible even larger than milliseconds. Therefore, the hard disk is substantially slower than the main memory, and if you want to compare the hard disk with CPU, the situation is much worse.

So, the question is a relevant question, because if a process does some operation, which results in a disk operation, a disk access being required, and if the processor waits for the disk operation to complete, in other words if the processor does nothing, while the disk operation is going on, then from the processor perspective this is going to take billions of cycles. Remember, the processor is to what is of magnitude faster, I mean substantially faster than the… Let me just explain, where this billions came from? (Refer Slide Time: 10:18)

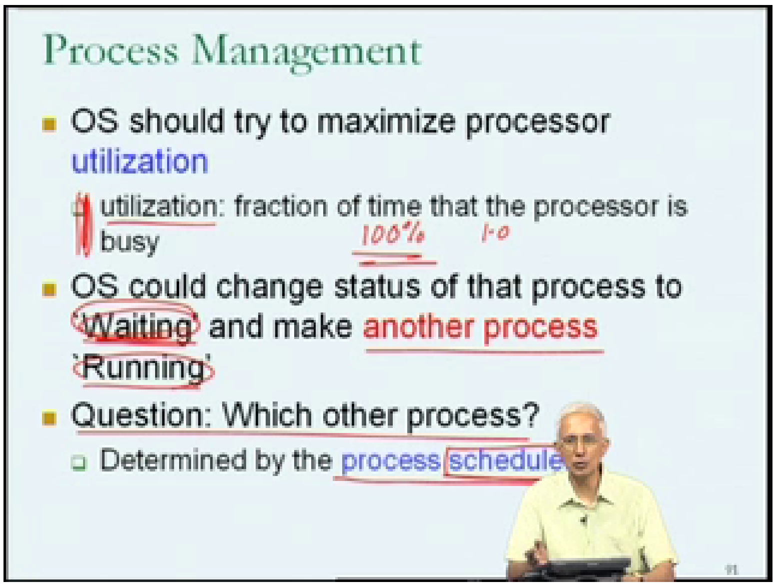
We know that the processors operating on a nanosecond time scale, and we know that the hard disk, rather than being milliseconds could even be operating on a scale of seconds.

I said there in the worse case, a disk operation could take a significant part of a second and this is where the billions of cycles come from, right. There is a difference of 10 to the power of 9 between these two numbers, (Refer Slide Time: 10:38), the 10 to the power of 4, was coming from a different calculation.

So, when a process executes an operation like a load word, which results in a page fault, the net result could be a disk operation, and if the operating system just allows the process to wait, for the disk operation to complete then that would take the equivalent of billions of cycles of processor time. In other words, if the operating system does nothing then the processor will be ideal for billions of cycles; billion is 10 to the power of 9. Because a processor is operating on a time scale of nanosecond, and might be able to finish the execution of one instruction in a nanosecond.

So, in effect, by doing nothing, by just allowing the process which is waiting, for the disk operation to complete, the operating system is causing the processor to idle instead of executing what may have been billions of instructions. Right, this seems to be a wasteful perspective, on how to answer the question. If the operating system had done something different, then conceivably the processor could have been busy doing something useful, rather than just waiting for the disk I/O, disk operation to complete.

(Refer Slide Time: 11:54)



## Process Management

- OS should try to maximize processor utilization
  - utilization: fraction of time that the processor is busy
- OS could change status of that process to 'Waiting' and make another process 'Running'
- Question: Which other process?
  - Determined by the process schedule

So, from the perspective of this problem, of that question, which I have just raised, the question was, what should the operating system do, if a process does an operation that takes a long time, such as a disk operation? The operating system should probably be designed, so that it tries to maximize processor utilization. This is one consideration that could be taken into account, in designing the different strategies, inside the operating system.

What do you mean by utilization? By utilization, I mean the fraction of time that the processor is busy. So, if I am able to…, if the operating system is able to make sure that the processor is fetching, and decoding, and executing instructions, a 100 percent of the time, then conceivably it will be much more effective, in reducing the delays that uses C, and making programs run much faster. Therefore, maximizing the processor utilization, percentage of time that the processor is busy, trying to get it as close to 100 percent or a fraction of 1.0. Utilization will often be described as a percentage (Refer Slide Time: 12:55), by multiplying the fraction by 100.

This would be a good objective for an operating system to have, but it is possible that other objectives might also come into play. Now, before we think about that. How could the operating system cause the utilization of the processor, to increase in the event that we have just talked about, where one process does an operation, which results in a long operation, like a disk access. The answer is the operating system could actually, change the status of that process, which was running into the state waiting, and you could make some other process, the running process.

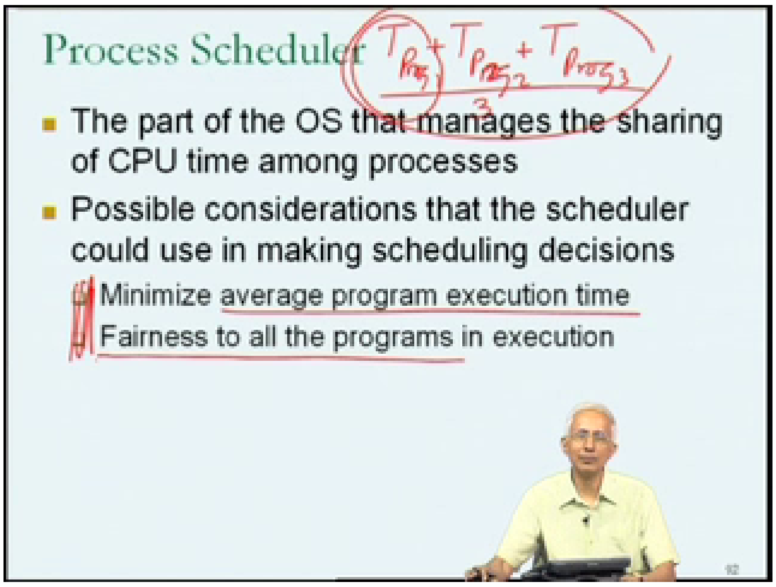
So, as soon as that disk operation happened, if the operating system had changed the status of the then running process to waiting, and had made one of the ready processes running, then the running process could have executed, potentially billions of instructions, in the time that the disk operation took place, and the net result would have been that the utilization of the processor, fraction of the time that the processor is busy, would have been close to this hundred percent objective.

So, this seems like good thing for an operating system to try to do. Try to maximize utilization, possibly by, switching from a process, which should be waiting, to a process that is ready, so that the process which was ready can now run and progress. Execute instructions, their by progressing towards its own termination, and also improving the processor utilization. So the processor utilization is some kind of an indication of how successful programs are as a whole. Now, in this particular suggestion, it was indicated that the operating system, could make another process into the running process. And that raises the question of, which other process might the operating system pick? (Refer Slide Time: 14:43)

So, when a process is running, and does a long operation; it would be made into, a waiting process, which is really clear. But the question is which process should be made into a running process, into the running process. Now, this decision is something, which is built into a part of the operating system call the process scheduler.

Since, the process scheduler, does the activity of scheduling processes to the CPU, it make decisions, about when a particular process should be allowed to run on the CPU. Hence, the name scheduler, just like you have a scheduler, in the bus station, to decide when buses should depart, you have a scheduler in the operating system, to decide when processes should be allowed to run.

(Refer Slide Time: 15:31)



## Process Scheduler

$\frac{T_{Prog_1} + T_{Prog_2} + T_{Prog_3}}{3}$

- The part of the OS that manages the sharing of CPU time among processes
- Possible considerations that the scheduler could use in making scheduling decisions
  - Minimize average program execution time
  - Fairness to all the programs in execution

So, let us talk about the specifics of process management, in terms of what a process scheduler might do. So, I will talk about the process scheduler, is being the part of the operating system that manages the sharing of CPU time, among the processes, which are on the system.

Now, what these possible considerations could the scheduler make in making scheduling decisions. This scheduling decisions relate to the decision of which other process should be allowed to run, that is the essentially the scheduling decision. Now, one consideration which might be used, is trying to minimize the average program execution time. So, on a particular system if I have, let us say three programs (Refer Slide Time: 16:18), in execution, and they may be running as six processes. Then the idea here is, each of this programs, started at some point in time and it is going to finish at some point in time.

So, I have some idea about how much time that program ultimately takes, similarly, some idea about how much time the second program ultimately takes, and some time about, some information about how much time, the third program actually takes. I will know all these times, after the programs of all finished execution.

So, then I could add them up and divide by three, and that is what I mean by the average program execution time (Refer Slide Time: 16:51). So, this would seem to be a reasonable idea, in the sense that if there are many programs in execution, if the alternative would have been that the operating system might try to do things makes scheduling decisions, based on the well being of one program. Whereas, here we talking about making scheduling decisions based on the average well being, of all the programs. We are trying to minimize the average program execution time.

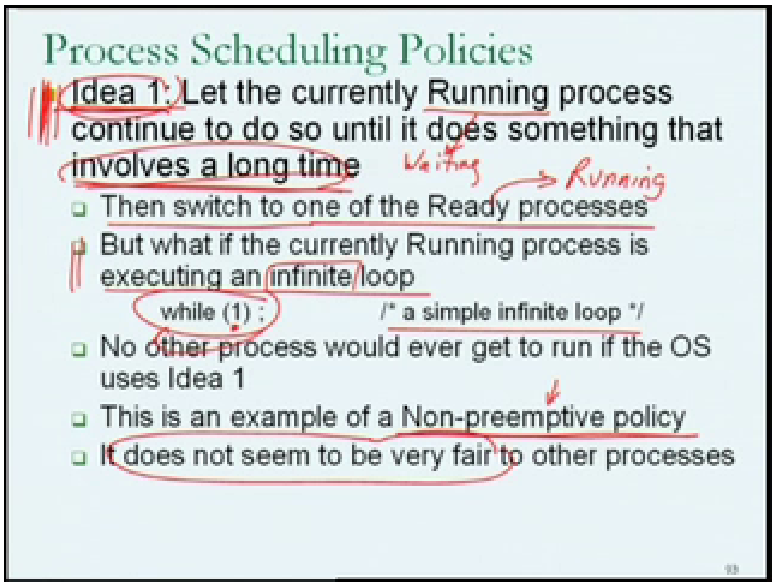
So, this has not seem to be, to bias towards any one program, and thus make sense in terms of a strategy the basis for building strategies they could be incorporated into the operating system scheduler. Now, unfortunately if one does this there is no guarantee that one program may not get treated extremely badly, at the expense of some other programs getting treated very well, and the average coming down. Therefore, an additional consideration that might be used, is to also ensure that theirs fairness to all the programs (Refer Slide Time: 17:49), in execution. One does not want to sacrifice one or two processes, in the interest of the average value.

So, minimizing average program execution time, does not actually guarantee fairness. So, this turns out to be another consideration that could be taken, into account by a process scheduler and we will relate to this in a few seconds.

So, in a sense, we are talking about part of the operating system, the person designs the operating system is going to have to design strategies and these are considerations, which might be used in designing strategies. We talked about, when we talked about memory management for example, we talked about the problem of how to how they how the page fault handler could determine, which page to replace.

So, that was the problem. It had to make decisions about which page to replace, and the strategy in that case, was built upon the consideration of, “principle of locality of reference”, and ended up in a policy called least recently used. So, we are going to run along the same lines. Here we have a problem; we have some kind of a strategy, which hopefully will boil down to some kind of policies, which can then be implemented inside the process scheduler.

(Refer Slide Time: 19:01)



## Process Scheduling Policies

Idea 1: Let the currently Running process continue to do so until it does something that involves a long time

- Then switch to one of the Ready processes
- But what if the currently Running process is executing an infinite loop

```
while (1) ;            /* a simple infinite loop */
```

- No other process would ever get to run if the OS uses Idea 1
- This is an example of a Non-preemptive policy
- It does not seem to be very fair to other processes

Now, there are many possible scheduling policies that have been considered. Let me start with some basic concepts, so that you'll realize in different systems that you might encounter, some of the other may show up. Now, one possible idea that could be used is we could let the currently running process, continue to run. Remember, running is a condition that a process is in. Running process is the one, which is in the CPU, its instructions are being executed, and it is changing in its state.

So, one overall strategy that an operating system process scheduler could have is, forget about waiting for a disk event, you could just have the strategy of, let the currently running process continue to do so, until it does something that involves a long time. So, until the thing like the disk access happens, let the currently running process continue to do so. This is one over higher level strategy that an operating system process scheduler might use. So, it is taking it is going to take into account the problem with disk access that we talked about, but until the problem arises, it just lets the currently running process continue.

Now, any process scheduling policy, which does, which uses this idea, when the scheduling, when the long event does happen, it would switch to it would switch the currently running process, into the state waiting, as we saw, and would switch one of the ready processes. In other words, among all the processes, which are ready to run, on the processor, it would take one of them and make that into the running process.

That is what I mean by switching to (Refer Slide Time: 20:37) one of the ready processes, but this means is, that it would make one of the ready processes, into the new running process. So, the old running process would become the waiting process, and one of the ready processes, would become the new running process, in the event of the long wait time. Now, what is the currently running process, does not execute a long wait delay. What if it (Refer Time: 21:00) does not execute instruction, which causes a page fault or it does not do a disk I/O, then there is a danger, that the currently running process could keep running indefinitely, and this may not be good for the other processes.

So, that is the one weakness of this idea, idea one (Refer Slide Time: 21:16). Just to highlight, how dangerous idea one could be. Consider the situation, which I outlined over here. So, I have a currently running process, and it is in a situation, where it is going to be guarantee there, it can continued to be currently running as long as it does not do a operation, which takes a long time.

In other words, it does not do a disk operation, or it does not suffer a page fault. Now, what if the currently running process is process, which is executing an infinite loop (Refer Slide Time: 21:43). First of all, what do I mean by infinite loop? By infinite loop, I mean a loop, which continues indefinitely. For example, in here, I am showing you a very simple, the simplest possible c infinite loop. It is a loop, in which it has a condition which is always true, and there are no statements inside the loop, so it just continues looping, forever and ever.

This kind of a loop is not going to suffer from page faults. It does not, since it does not make memory accesses. It is not doing any file IO, therefore, it is not going to involve any a long operation. It is just going to execute forever and ever and ever, which is why it is called an infinite loop and over here, I comment that is a simple infinite loop (Refer Slide Time: 22:25).

So, this is clearly a malevolent program. It was written with the intent of this running in definitely, but if there was such a program running as a process, on the system, and it became the running process then this process under idea one, would never give up the CPU and therefore, no other process would have any opportunity of running on the system.

No other process, would have any opportunity of running on the system, if they process scheduler used this idea. Now, what this really means is that for systems, operating systems, which are going to be used in situations, where real users are involved, cannot use this idea. This is an extremely dangerous idea, because any user who wanted to, could just write a program, which has an infinite loop, just for the fun of it, may be just you see if you can cause a whole system to become useless, and the net effect would be the no other process running on the system, would make any progress. No other programs would complete. His infinite loop could run indefinitely. So, that would be malevolent behavior an attempt to hepatize the system.

But there are situations, where this kind of an infinite loop, may happen without malevolent intentions. For example, when you are debugging a program, it is conceivable that before the program reaches its correct state, it may contain situations which you are not anticipated, as a result of which they may be infinite loops, within the program, and if you left your program over night to run, because for some reason of the other, during the time that you are away, no other process would be able to make any progress, if the operating system had used idea one. So, even it is no malevolent, this kind of situation could arise during, for example, process debugging or something of that kind program debugging.

Now this kind of idea, is clearly not good for, as I said for operating systems, that are going to be used by multiple users, as no other process would ever get to run, if the operating system uses idea one. However, this idea is used in certain other scenarios, and in general, an operating system page process scheduler, which uses the idea one, would be referring to as a Non-preemptive policy (Refer Slide Time: 24:40). The alternative that we are going to see is, where there is something called preemption, and the operating systems that you typically used use preemptive scheduling policies.

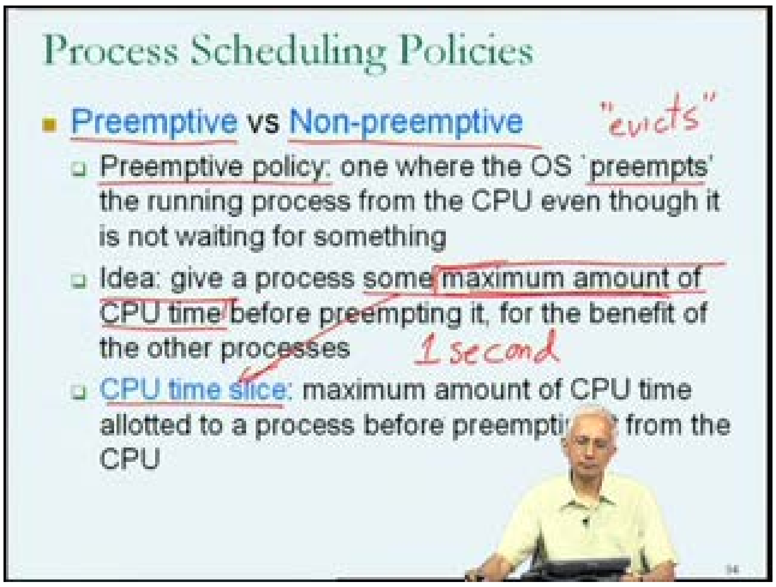
But the start off with, it would be useful to get an idea about, how non-preemptive policies might work, and just put your mind at rest, a non-preemptive policy would not be used for an operating systems are Unix or Linux, but could be used in some restricted environment, where it is known that there are no infinite loops, and where the programs have written very carefully, so that they give up the CPU, by doing possibly, by doing long duration events, every once in a while.

So, that other process can get access to the CPU. For example, in certain embedded systems, such as within a cell phone, where different programs may be in execution, this kind of scheduling policy, may in fact make sense.

So, we should not rule out the idea of a non-preemptive policy. But from the perspective of Unix like situations, where there are multiple users, and it looks like a single user, if he knew that the operating system was using a non-preemptive policy, and that by writing a program with an infinite loop, he could cause havoc, prevent other processes making any progress, this is seems to be a very good example of not being fair. (Refer Slide Time: 26:00). I had used the term fairness, and this would clearly be a situation where a lot of processes are being treated unfairly.

They have absolutely no chance of getting CPU time and therefore, one does not associate fairness with the scheduling policy. The fairness would have to be incorporated into systems that use non-preemptive policies, by careful encoding of the programs. The programs would have to be responsibly written.

(Refer Slide Time: 26:19)



## Process Scheduling Policies

- Preemptive vs Non-preemptive "evicts"
  - Preemptive policy: one where the OS 'preempts' the running process from the CPU even though it is not waiting for something
  - Idea: give a process some maximum amount of CPU time before preempting it, for the benefit of the other processes 1 second
  - CPU time slice: maximum amount of CPU time allotted to a process before preempting it from the CPU

Now, the alternative as I said, is that an operating systems process scheduler, could use something called preemption, as supposed to, not doing preemption, and the idea of preemption is, that the operating system, every once in a while, preempts the running process, from the CPU even though it is not waiting for something. And what we mean by preempts is evicts. We had used a word evicts, in connection with page replacement, replacing one page with another page in main memory.

Here, we have a similar kind of a concept. One process is being evicted from the CPU, in favor of another process and that is what the word preempt is being used for here. So, in a preemptive process scheduling policy, the operating system will preempt or evict the running process, from the CPU, even though it is not waiting for something, even though it is no doing a long duration event. For example, and this is clearly being done the interest of either improving the average program execution time or in terms of improving the fairness to the other processes on the system. So, we expect that Unix and Linux systems, we are going to find this kind of preemptive policy.

Now, the question which may arises is when does the operating system preempt a process (Refer Slide Time: 27:37), and the ideas that an idea which is often used, is that in a preemptive scenario, the operating system might give a process, some maximum amount of CPU time, and then preempt it, and the reason for preempting, is to give some benefit to the other processes, which are ready on the system.

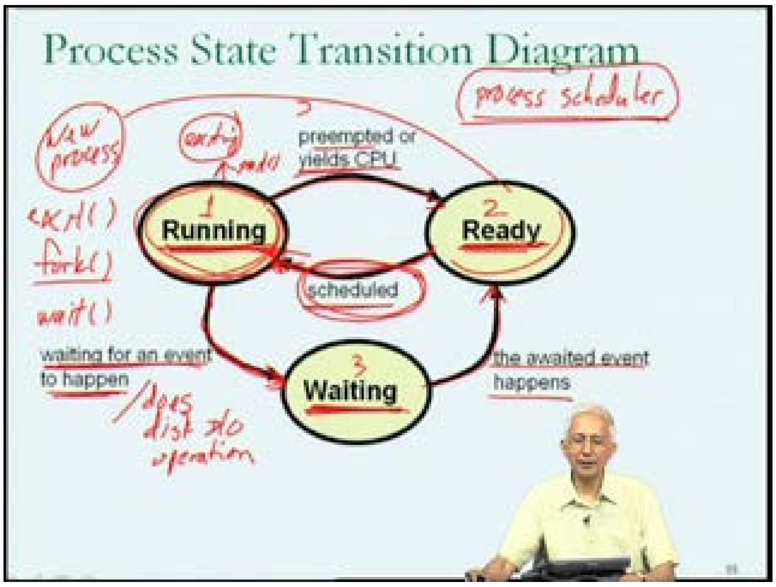
Now, the term CPU time slice is often used for that maximum amount of CPU time. So for example, if you are told that the CPU time slice on your Linux system is one second (Refer Slide Time: 28:07), when you take that to mean that after a process, has been running for one second, the operating system will preempt it. It will move that process into the ready state, and cause one of the ready processes, to become the currently running process, and by doing this other processes are getting some CPU time, and this will keep the behavior of the sharing of the CPU time resource little bit fairer.

So, this concept of the CPU time slice (Refer Slide Time: 28:36), which is the maximum amount of time that is a process will get before it is preemptive or evicted, from the CPU by operating system.

Why do I use to term maximum amount of time? I use the term maximum amount of time, because it is possible that before this one second has elapsed, the process may do an operation, which causes it to delay, may do an operation such as a file I/O, or may suffer a page fault, as a result of which, the operating system may have switched it out, because it was waiting for a long duration event, anyway.

So, that is why I use the term, maximum amount of time. This one second is the maximum amount of time, a process could get, and you will get this one second only if I does not cause the operating system to remove it, from the CPU for some other reason, such as, doing a long duration event possibly relating to a disk operation.

(Refer Slide Time: 29:28)



Now, with this a quick introduction to how a process scheduler might work, we can get an overall picture of what is happening behind the scenes, from the perspective of the process, by something call the Process State Transition Diagram. So, we know that there are three process states that we have been talking about. They are “Running”, “Ready” and “Waiting”, and in this diagram we are going to try to show the conditions, under which a process, which is in one of these states, could move into another one the states.

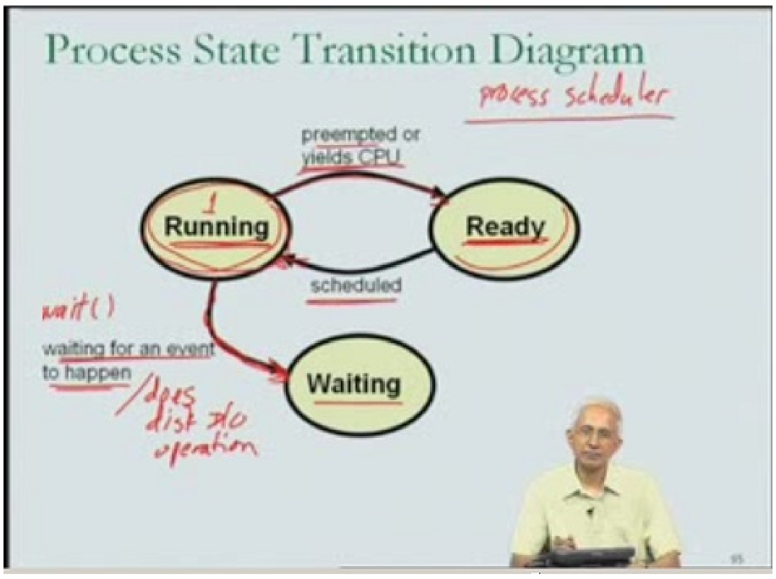
And this kind of process state transition diagrams are useful to, get an overall perspective of what the operating system is doing. So, when you look at a process state transition diagram, you will see many arcs. So, this is an arc going from the running state to the ready state, and it is labeled. And this particular arc is labeled, preempted or yields the CPU and this is going from the running state to the ready state.

So what this means is, if there was a process, which was in the running state, and it was preempted (Refer Slide Time: 30:28). In other words, the operating system forcibly evicts from the CPU, then that process would move into the ready state. Now, it is also possible that there are mechanisms, which a process can voluntarily yield the CPU, or give up the CPU. We have not talked about any such mechanism yet, will suppose that there are. Then if there was a process, which was in the ready running state and it voluntarily yields the CPU, by using mechanisms available for that purpose. Then that process would move into the ready state. It is not waiting for anything to happen; it is ready to actually start running once again. That is why I talk about and that is why we see this arc going from the running state to the ready state.

Now, under what conditions could a process move from the ready state to the running state, and the answer is, we look at an arc going from the ready state to the running state, we see that the label on that is scheduled (Refer Slide Time: 31:25). We have seen the term scheduler. I talked about the operating system process scheduler, and I said that the operating systems schedules processes on to the CPU, and therefore, this word scheduled must mean the operating system operation through which a process which is run ready is selected to become the running process.

Hence, the arc going from ready to running is label scheduled. A process will move from the ready state to the running state, because it is selected by the process scheduler to be the one process, from among all the processes, which are in ready state, to move to the running state, and become the one process that is running.

(Refer Slide Time: 32:08)



## Process State Transition Diagram

Now, if you look at the arc going from the running state to the waiting state, we see that it is labeled waiting for an event to happen. So for example, if there is a process which is running, and it executes the wait system call, then we know that it is basically wants to sleep until something happens to its child, and in this activity, it is waiting for an event to happen, which the event, it is waiting to happen is the possibly the exit of its child. And therefore, that is why this is labeled as waiting, for an event, to happen. The result to which it goes into the waiting state, it does not go into the ready state, because it s not actually ready to run right now. It will be ready to run only, when the event that it is waiting for, has happened.

There could be a similar arc, or similar label, which could be put here, for a process which does a disk I/O operation, right, the long duration event that we had talked about earlier. So, if there is a process, which is in the running state and it does a disk I/O operation, then we now learn, it will going to the waiting state, and it will be in the waiting state, until the disk I/O operation completes. What should happen to the process which is waiting for the disk I/O, to complete? (Refer Slide Time: 33:24) Remember, in this case, the even that it is waiting for, is the completion of the disk input or output operation.

So, what should happen to the process which is waiting for the disk input or output operation to complete, when the disk I/O, actually does complete? The answer is that we expect that it should move into the ready state, and therefore we expect to see an arc going from waiting to ready and labeled the waited event happens (Refer Slide Time: 33:50), from waiting to ready. Subsequently that process, in the ready state will soon or later, gets scheduled, at the discretion of the process scheduler.

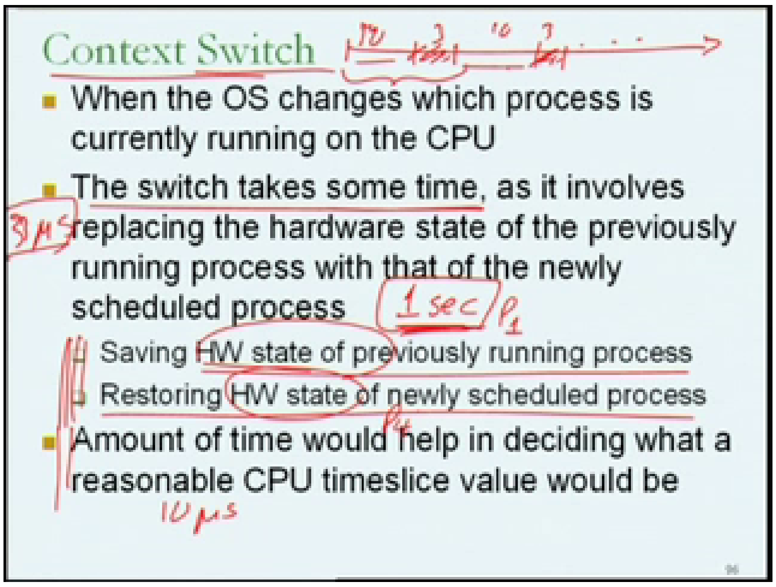
So, a diagram like this (Refer Slide Time: 34:04), is very useful. It gives us a good idea about the behavior from the perspective of processes, behavior that processes can expect, in terms of how the operating system process scheduler is, sharing the CPU time among all the processes on the system. In practice, you may see process state transition diagrams with many more states. For example, if you looked at the Linux process state transition diagram, and may have a large number of states, and that may relate to certain other conditions, which may have to be taken into account or simply certain special cases that occur.

For example, you know that when a process is running, when it executes a fork system call, the effect is that a new process comes into existence, and that conceivably it will take some time, for that new process to actually be created, because a lot of activity has to happen to create a new process. Its text, data, stack, heap, have to be created, and that involve a lot of memory operations, and possibly some disk operations. Therefore, a fork operation is likely to take a fair amount of time. So, if the process which is running is doing a fork operation, the new process, which comes into existence may come into existence (Refer Slide Time: 35:14), in another state, from which it might later on move, into the ready state. So, they could well be a state called new process, and it might get entered into, because of a fork system call.

Similarly, a process which is running and executes the exit system call, we know essentially, wants to terminate gracefully. So, it is conceivable, that one might find another state, which is labeled, which is known as exiting, and that is entered into from the running state, only if the process executes the exit system call, so this might be labeled exit (Refer Slide Time: 35:51).

So similarly, they could be other states, in a more…, in a real world process state transition diagram, but this three state, process state transition diagram, that we have discussed, gives us an adequate idea, about what is happening behind the scenes in terms of a process, being changed from waiting for something to happen, to ready, to be allow to run on the CPU, to actually being the one process that is running on the CPU, with a sequence of events that we went through in this in this slide. Now, let us think a little bit, about what is involved in the transition from ready to running, which happens, when a process which was ready is scheduled to run.

(Refer Slide Time: 36:35)



## Context Switch

- When the OS changes which process is currently running on the CPU
- The switch takes some time, as it involves replacing the hardware state of the previously running process with that of the newly scheduled process
  - Saving HW state of previously running process
  - Restoring HW state of newly scheduled process
- Amount of time would help in deciding what a reasonable CPU timeslice value would be

Now, this particular event is often referred to, as a, “Context Switch”. And it is called a context switch, because it is what happens, when the operating system changes, which process is currently running on the CPU. So, I will go back to the previous slide (Refer Slide Time: 36:55), previous slide. So, if process P 1, was currently running, and many processes were currently ready, then when the operating system evicts process P 1, process P 1 may end up as a ready process and process P 4 may end up, as a new running process. So, there is a switch from process, P 1 to P 4 (Refer Slide Time: 37:12), as far as the running on the CPU is concerned. That is why this is called a switch. So, when the operating system changes or switches, which process is running, currently running on the CPU. This event is known as a context switch.

Now, the thing to notice that context switch may be, an event that takes, a reasonable amount of time, because what is going to have to happen, is that whatever state was present, in the hardware, in other words, the contents of register, the contents of program counter, the contents of instruction register, prior to the context switch, all related to the old process, which used to be running. In other words process P1, and before process P4 can start running, on the process, its picture of what should be in the registers and what value should be in the program counter register.

And what instruction should be the instruction register must be present, in the hardware. therefore, the context switch must involve, replacing the hardware state of the previously running process, with that of the newly scheduled process, and only after that has been done, can the newly scheduled process, has actually start changing the values of the registers.

Therefore, associated with a context switch (Refer Slide Time: 38:24), there must be some saving. We must, the operating system must remember or save the hardware state of the process that use to be running, in other words process P 1, and it must put into the hardware, into the register. In other words, it must store the hardware state, of the newly scheduled process, in other words, P 4.

So, the context which will involve these two operations: saving the hardware state of the previously running process, and restoring the hardware state of the newly schedule process. What is this mean? This means, that somewhere, the operating system must be remembering the hardware context. In other words, the hardware state, the contents of all the hardware registers, as far as, each process is concerned. The perspectives of process P1, about what the contents of the registers are, as suppose to the perspective of process P4 as far as, the contents of the hardware registers are concerned.

When process P1 is evicted, from the CPU, it must remember what its perspective of the registers was, and when process P4 is to enter the CPU, it must put into the hardware process P4 perspective of what the hardware registers are supposed to contain. So, where is the operating system supposed to save all this information the hardware state?

You will recall that we had talked about the hardware state. The hardware registers as being part of the data, associated with the process and therefore, the operating system is going to have certain space, associated with remembering the hardware state, of each process, and this is going to be part of the context or the hardware state of each particular process.

So, the operating system is going to manage this in memory, and the saving of the whole state, and the restoring of a new state, are the essential components of what happens during a context switch. And the reason that I say that this could take a reasonable amount of time, is that this is clearly going to involve many memory operations, because for in the case of a risc instruction set, each of the register values will have to be loaded from wherever it was stored, by the operating system, into the corresponding register.

And if there are 32 registers, and several 32 general purpose, 32 general purpose integer registers, and 32 general purpose floating point registers, and 20 or 30 special purpose registers. The size of that hardware context could easily be a few hundred words and therefore, it is going to take a few hundred instructions, for this context switch to happen. This is not unrealistic for us to expect, which is why I said, that the context switch could take some amount of time.

So, there is a little bit of overhead associated with this aspect of how the operating system is managing the CPU time, but it is something which may be necessary, in order to allow a single CPU to be shared by a large number of processes.

So, (Refer Slide Time: 41:13) this amount of time that it takes, may also be useful for the person designing the operating system, in designing what would be a reasonable CPU time slice value. Remember, we talked about CPU time slice in a previous slide (Refer Slide Time: 41:29), it was the maximum amount of CPU time that could be allotted to a process, before it is preempted from the CPU.

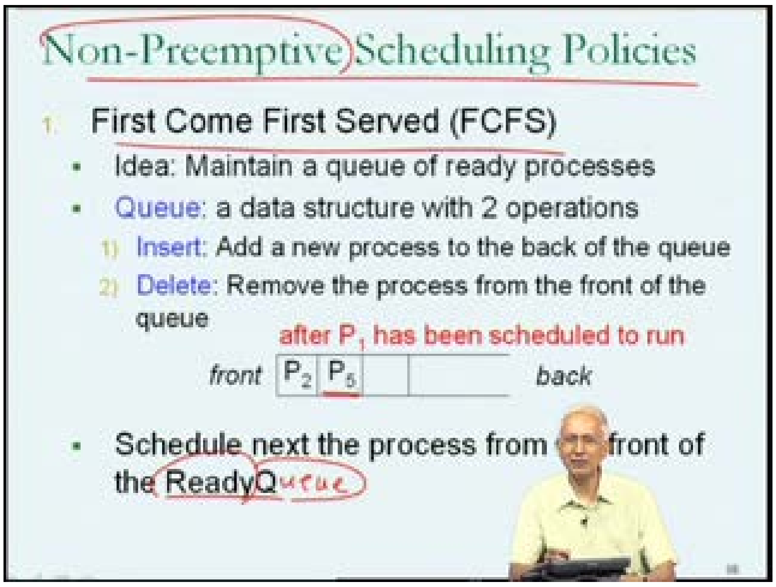
So, when you come back to this slide (Refer Slide Time: 41:38), if I told you that the amount of time to do a context switch, is let us say three microseconds, and then you are asked to decide, how big to make the CPU time slice, we could greatly understand that you would clearly not want to make the CPU time slice, just 10 or 15 microseconds.

Because, if we make the from the perspective of the CPU, 10 or 15 microseconds is actually a large amount of time, because in 10 or 15 microseconds, we can execute thousands, if not more of instructions. However, from the perspective of over head, having a CPU time slice of only 10 microseconds would mean that if I looked at the CPU time line, initially it runs a process for 10 microseconds, then it evicts the process, it takes 3 microseconds to context switch, and then it runs in next process for 10 microseconds, and then it takes 3 microseconds to context switch, and so on. Which means that out of every 13 microseconds, 3 microseconds are wasted, or are not used for the are not used for the useful execution of programs.

In that sense, I might view it as a waste. It is not actually contributing towards the execution of programs, what is part of the over head of what the operating system is doing, and could be viewed in some sense as a waste.

Therefore, this is a fairly high overhead to pay in terms of three microseconds, out of every thirteen microseconds, being used in this operating system over head. On the other hand, if I actually used CPU time slice of one second, then we will say that 3 micro seconds, is fairly small almost insignificant part of one second and therefore, the over head of context switching becomes small enough relating to the CPU time slice. And therefore, the operating system designer will clearly use the CPU time slice, the context which time over head idea, how much it is, in making a decision, about what the reasonable CPU time slice would be. This number of one micron one second may not be that unrealistic for systems today, as is this rough idea, about how long it takes to switch from one process to another.

(Refer Slide Time: 43:45)



## Non-Preemptive Scheduling Policies

1. First Come First Served (FCFS)
   - Idea: Maintain a queue of ready processes
   - Queue: a data structure with 2 operations
     1) Insert: Add a new process to the back of the queue
     2) Delete: Remove the process from the front of the queue
   - Schedule next the process from the front of the ReadyQueue

Now with this, we can actually look at some specific of some specific ideas that are talked about for scheduling policies, just like we talked about LRU, FiFFO, Random, a specific ideas, for how one could do page replacement. I will give you some ideas about some of the specific process scheduling policies people talk about. For completeness, we start off with some non-preemptive scheduling policies. A one idea for non-preemptive scheduling policy is this idea of First Come First Served. The idea here, the process which came into existence first, could be the one that is allow to run in the CPU first, hence the name first come first served.

So, to implement this kind of a policy, the operating system would clearly maintain the operating system scheduler, would maintain a queue of ready processes. It is obvious to those of you know the queue is. Let me explain, what I mean by queue? A Queue is a data structure, we learned about the stack as a data structure. Now, I am telling about the queue, as a data structure. It is a very well define term.

The queue has a data structure, other than the creation, in deletion, has two well define operations. The two operations are insert, which in our context here would be, adding a new process to the back of the queue, and delete which relates to removing the process from the front of the queue. Hence, when you think of a queue (Refer Slide Time: 45:10), you think of the front and the back. In the way that the first come first served policy, scheduling policy would use a queue, is that when the time comes where it has to decide, which process to schedule next. In other words, which process to execute on the CPU next? It would schedule the process, which is at the front of the queue, hence the name front.

So, when you insert a new process, it goes to the back of the queue. And when you want to take a process out of the queue, you take it from the front of the queue. So, queue is typically denoted like this (Refer Slide Time: 45:40). It has a front, and it has a back. So, the things go in to the back and they come out from the front. Just like a queue in principle, when you waiting at a queue in a bus stop or something like that, this is the idea, which you would have.

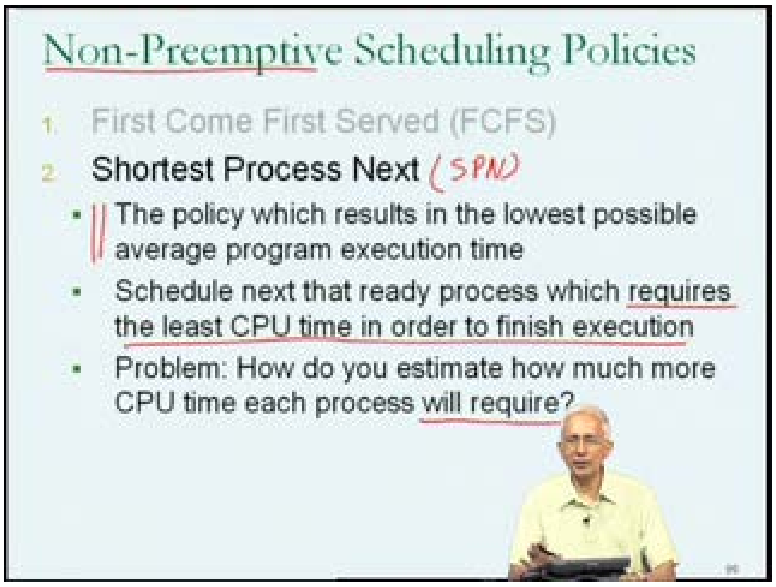
So, if there is a process P 1 in the queue, then a process P 2 comes into existence. P 2 will be added at the back, behind process P 1, after this P 3, P 5 comes into existence, and it gets added into the queue after P 2. At this point of the currently running process is evicted than the scheduler if choosing the first come first served policy, would take process P 1 from the front of the queue, and make at the running process. So, after P1 has been scheduled to run, the queue would like this (Refer Slide Time: 46:25). P 2 and P 5 would be there and P 1 would be the running process.

So, here, I would use the term, ReadyQ (Refer Slide Time: 46:33), because the well I am abbreviating it here. Technically, I should write this as ReadyQueue, but the operating system is maintaining the information about all the ready process in the system, in a queue, because it is using a first come first served policy, and this abbreviation of ReadyQueue, is often used.

So, this is a very simple idea, but we have seen that non-preemptive scheduling policies are not too good anyway. So, you may ask, what is the problem with this particular scheduling policy? And the answer is that just imagine what happens if process P 1 takes a long time to execute, and process P 2 has also huge amount of execution time, but process P 5, requires only a small amount of time to execute.

Then under the first come first served policy, the average execution time, would actually be fairly high, because the execution time for P 5 would include the amount of time that it waited in the queue for P 2 to finish execution. Therefore, first come first served is not a very good policy, even from the let long it is a non-preemptive policy. So, it is not very good from the perspective of being fair. It is not even a very good policy from the perspective of average execution time, because it is does not seem to do anything special about processes, which have short requirement or small requirement of the CPU.

(Refer Slide Time: 47:53)



## Non-Preemptive Scheduling Policies

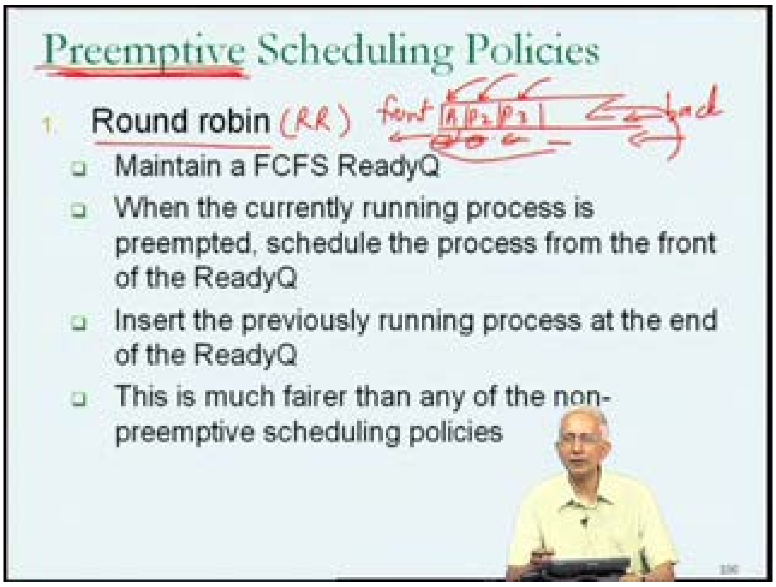
1. First Come First Served (FCFS)
2. Shortest Process Next (SPN)
   - The policy which results in the lowest possible average program execution time
   - Schedule next that ready process which requires the least CPU time in order to finish execution
   - Problem: How do you estimate how much more CPU time each process will require?

Therefore, another process scheduling policy, which is talked about in the context of non-preemptive situations, is this scheduling policy which actually tries explicitly, to minimize the average program execution time, by always scheduling the process, which requires the least amount of CPU time.

So, among all the processes in the ReadyQ, among all the ready processes, it tries to identify the process, which has the lowest requirement as far as execution time is concerned. In other words, that identifies the among all the processes, the one which requires the least CPU time, in order to finish execution, and it schedules that process and the net effect is going to be that is going to have lowest.

This particular policy which is called as, Shortest Process Next, abbreviated as SPN probably has lowest possible average program execution time. It does of course, suffer from a minor problem, in that one has to be able to get good estimate, about how much CPU time each process will require, which often makes it in feasible. But if one is able to get good estimates of how much CPU time, each process will require, notice the problem here, is that this requires looking to the future. Therefore, always problematic, but if one can overcome that problem, then shortest process next is a good policy, because it has some good properties. But since, we are not overly concerned about non-preemptive policies, let us quickly move forward and look at some preemptive scheduling policies.

(Refer Slide Time: 49:20)



## Preemptive Scheduling Policies

1. Round robin (RR)
   - Maintain a FCFS ReadyQ
   - When the currently running process is preempted, schedule the process from the front of the ReadyQ
   - Insert the previously running process at the end of the ReadyQ
   - This is much fairer than any of the non-preemptive scheduling policies

Now, one of the older and easy to understand preemptive scheduling policies, this scheduling policy called as Round Robin. This is often abbreviated as RR. You will remember that what I mean by preemptive scheduling policy is that after the running process has received the maximum permissible amount of CPU time, which is defined as the CPU time slice, may be one second, then the operating system will automatically evicted from the CPU, in favor one of the ready processes.

So, that is the class of scheduling policies we are talking about now called preemptive scheduling policies. How does a Round Robin work? Now what Round Robin does, is like the first come first served policy, it maintains a first come first served ReadyQ.

So, it maintains a queue along lines of what we saw for the… a queue is something with a data structure, which has a front and a back, new processes (Refer Slide Time: 50:21) enter at the back, and a process exist from this only from the front. So, the round robin policy, maintains a first come first served ReadyQ, so all the ready processes would be present inside a data structure of this kind, and operating system data structure of this kind. Now, when remember this is a preemptive policy. Soon or later, the currently running process will reach the end up its time slice, and will have to be evicted on a context switch.

So, when the currently running process is evicted, what will have happen is, scheduler is schedule the process from the front of the ReadyQ, and what it will do with the currently running process, process which was just running, until the contact, which occurred. It will put that previously running process to the end of the ReadyQ, to the back of the ReadyQ.

So, essentially the processes which are in the ReadyQ, one after the other, will get scheduled, and therefore, in some sense we view this as being fair. So, it is definitely much fairer the any of the non- preemptive scheduling policies, and we see the beginnings of fairness here, because the general idea is, that each process will get one CPU time slice, before the next process gets its own one CPU time slice, and a process after having got its CPU time slice, enters the queue at the back, and therefore, soon or later, it too will later on get it second CPU time slice, and so on. Therefore, all the processes one by one will get will make progress when they get CPU time, as I mean the schedule, under this round robin strategy.

So, definitely much fair than anything that we have seen up to now, and seems a good basis for the management of processes, on something like a Linux or Unix system. You notice that this particular policy does not suffer from the problem that we saw with the shortest, with the first come first served policy, because even if process P 1 requires hours and hours of CPU time, even if process P 1 contains an infinite loop, and even if process P 2 requires of huge amount of CPU time, and process P 3 requires very small CPU time. The most amount of time that process P 3 will have to wait, before gets scheduled would be two CPU time slices. Therefore, there is an element of fairness. No particular, no one process is going to get overly, unfairly treated, due to the scheduling policy.

So this turns out to be a very interesting kind of an idea. And this is probably a good time to break without discussion of preemptive scheduling policies for today. We have not yet reach a point where we see a policy that is actually adequate, for the real needs of an actual operating system, such as Linux or Unix, but we are almost there. In next lecture, we actually start by looking at something very similar, to the scheduling policy, used in a real Unix or Linux system.

Today, we have actually seen a great deal about the difference states, that a process could be in. We understand that the operating system manages the CPU time, by switching currently running process, at any given point in time, if there is only one CPU, only one process can be running. But by keeping track, of the other processes that are in a condition where they could run, and by periodically switching from the process, which is running, among the processes which are ready, using a process scheduling policy, the operating system is able to keep all the processes in some state of progress, and allowing successful completion of processes. We stop here, for today and move forward to a real word kind of scheduling policy, in the next class.

Thank you.