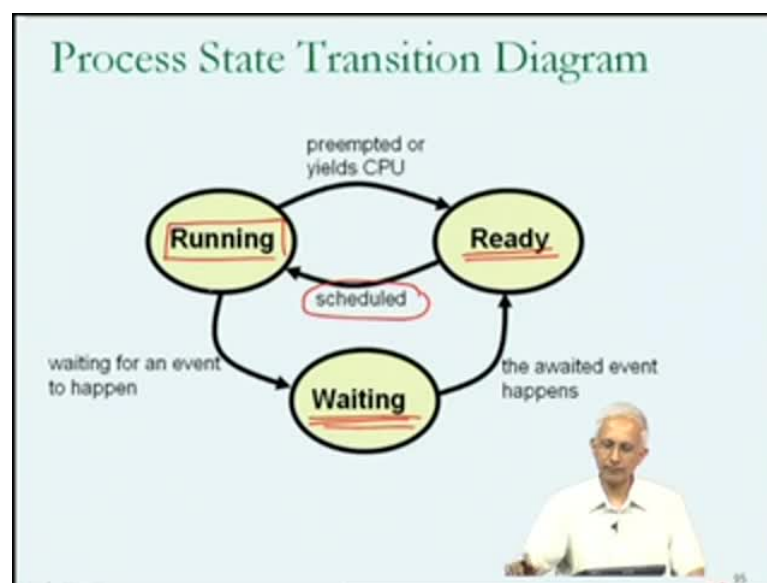


**High Performance Computing**  
**Prof. Matthew Jacob**  
**Department of Computer Science & Automation**  
**Indian Institute of Science, Bangalore**

**Module No. # 04**  
**Lecture No. # 18**

Welcome to lecture number 18 of the course on High Performance Computing. Just to remind you what happened in the previous lecture, we were trying to understand what happens when a program of ours runs on a computer system. We saw that the operating system is sharing the resources of the computer system among all the programs in execution. In particular, as far as the use of the one processor on the system is concerned, we looked at how the operating system, by actually viewing each process as a data structure, as a collection of data and some operations associated with it, can shift a tension from one process to another process. This was well illustrated by a very simple diagram that we came up with, called the phase operating systems perspective of how the process states transit or the process state transition diagram. So, I will just start off by reminding you about that.

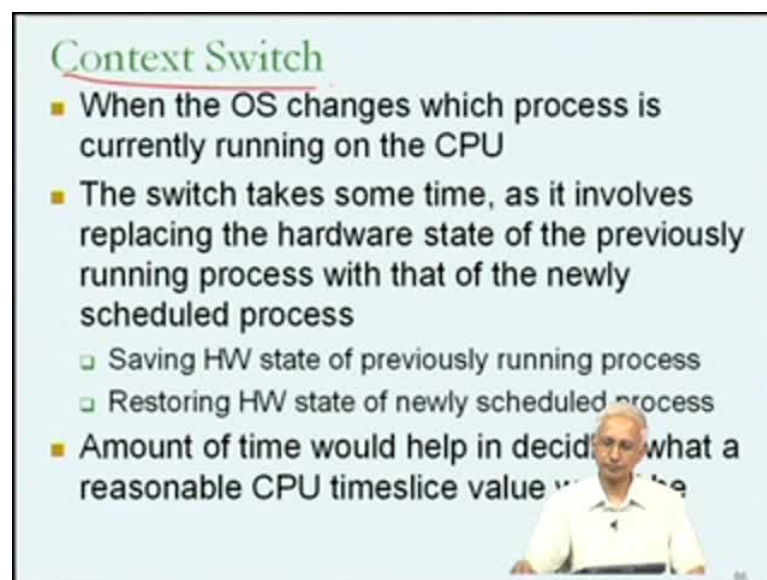
(Refer Slide Time: 01:09)



In the process state transition diagram, there is one state associated with each possible condition that a process could be in. At any given point in time, one process can be running on the CPU, several processes could be waiting for events of various kinds to happen. Once the events happen, each of those processes could become ready to be allowed by the operating system to run on the CPU.

Now, the interesting question became - when we have one program running on a system sharing the resources with hundreds of other processes, what do we need to know about what happens to our process? Now, we see that it may be useful to know something about how the operating system decides when a ready process should be allowed to run and that is what we are looking at.

(Refer Slide Time: 01:52)



**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

We are looking at various possible strategies the operating system could use. The event of switching from one process to another is described as a context switch. It involves a little bit of overhead, because the hardware state of the process which used to be running must be remembered because you will subsequently have to run at some point later in time. The hardware state of the process, which is to start running on the processor, must be put into the registers of the processor. So, this is what we call the context switch.

(Refer Slide Time: 02:24)

## 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


after  $P_1$  has been scheduled to run

front 

$P_2$	$P_5$		
-------	-------	--	--

 back

- Schedule next the process from the front of the ReadyQ




The operating systems process scheduler could be written to use many different kinds of policies depending on what the perspective of the person who designs the operating system is. We saw that the two possible higher level considerations could be either to try to come up with the policy that reduces the average amount of time to execute a program across many programs; alternatively, to try to be fair to all processes. Various possible process scheduling policies could be designed. We saw that if one takes a viewpoint that the operating system should not preempt a process. In other words, allow a running process to run as long as it wants. Then, there are one or two possible policies such as First Come First Served, which would be implemented using a queue of processes, but which would suffer from something like an infinite loop running in a process because that process would actually never yield the CPU.

(Refer Slide Time: 03:21)

## Non-Preemptive Scheduling Policies

1. First Come First Served (FCFS)
2. **Shortest Process Next**
  - 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?




The alternative is to use non-preemptive policy such as Shortest Process Next, which, once again will not do anything about the infinite loop process, but will be a little bit fairer among processes in order to reduce the average program execution time. Since neither of these policies is ultimately fair enough for the requirements of a real operating system like Linux or UNIX, we learnt that, in practice preemptive policies are what we should expect.

(Refer Slide Time: 03:48)

## Preemptive Scheduling Policies

1. **Round robin**
  - 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



A preemptive policy is one which periodically will switch from the running process to one of the ready processes in order to be fair to all processes. One simple idea in this direction was the Round Robin policy, which we described in this way. It maintains a First Come First Served ReadyQ, a newly arriving process, a newly created process will be added to the end of the ReadyQ.

A process, which is running at some point in time, will be preempted at the end of its CPU time slice in favor of the process, which is at the head of the ReadyQ. The process which was preempted itself will enter the ReadyQ at the end. So, in this way, processes periodically get some amount of CPU time and then go to the end of the ReadyQ, and so on until they finish their execution. So, in that sense, I described it as much fairer than any of the non-preemptive scheduling policies.

We also saw that this scheduling policy would not suffer from an infinite loop in a program because at the end of the CPU time slice, even if the process was running an infinite loop, it would be preempted from the CPU in favor of one of the ready processes.

(Refer Slide Time: 04:58)

**Preemptive Scheduling Policies**

1. Round robin *(FRONT) ReadyQ 4 (BACK)*
2. **Priority based** *4 3 2 1*
  - ❑ The readyQ need not be ordered on FCFS basis
  - ❑ It could be ordered on any other priority instead
  - ❑ For example: The process that has not run for the most time could get the highest priority
  - ❑ The scheduler could even assign a longer CPU timeslice for certain processes*1 sec 100 msec timeslice*

*108*

Next, we are going to look at the alternatives to the Round Robin scheduling, which would try to take other considerations into account in a sense to try to become fairer to processes in a commercial system like UNIX or Linux, we suspect that fairness might be an important consideration. In this direction, rather than just inserting processes into the

First Come First Served queue on the basis of their time of creation, some other consideration could be used. So, the idea of any priority based scheduling scheme is that the ReadyQ need not be ordered on a First Come First Served basis.

First Come First Served basis is in some sense arbitrary. It does not take into account any attributes of the process other than its time of creation. In order to be fair to processes, it might actually make sense to have other considerations, which could be described by what is called a priority. You have heard the word priority in connection with common everyday usage. The word priority would imply that all processes are not considered to be equal at all points in time. Some may have higher priority or higher privileges or higher preferences than the others. The scheduling policy could take a priority into account, instead of treating all processes as exactly equal.

Now, if one gives a priority based scheme, the ReadyQ rather than being First Come First Served could be ordered on priority. What do I mean by ordered on priority? Now, whenever we think of a ReadyQ, we think of a data structure in which there is a back (Refer Slide Time: 06:37) and a front. The idea is that any newly arriving process would typically enter at the back. When a process has to be scheduled; in other words, one of the ready processes in this ReadyQ has to be made into the running process, one would take the process from the front of the ReadyQ.

Now, if I am ordering the ReadyQ on priority what this means is that, processes will not enter the queue based on the time of creation or anything like that; rather, they will enter the ReadyQ at the appropriate point depending on their priority. For example, let us suppose that there is a process in the ReadyQ with priority 5. I am describing the priority of a process by an unsigned integer. We will assume that the higher the number, higher the priority. The next process might have a priority of 3, the next process might have a priority of 2. Now, let us suppose at this point in time, a new process comes into existence, you will notice that I have ordered the ReadyQ in terms of priority. The process with highest priorities at the front of the queue, the process with second highest priority is next, and so on.

Suppose at this point in time, a process with priority 4 comes into existence due to a fork or something, in the default FCFS, First Come First Served ReadyQ, the process with priority 4 would have entered at the back of the queue and appeared in the queue after

process with priority 2. However, if I order the ReadyQ in terms of priority, then the process with priority 4 will enter the ReadyQ here (Refer Slide Time: 08:10). The other processes, which had priority 3 and 2, would move back in the ReadyQ. So, in this way, we are giving slightly better treatment to the process, which has priority 4 because of its priority rating. **So, what could the priority be decided based on?**

There are many examples which one could think of. One possible idea is the one which I have written over here. The process that has not run for the most time could get the highest priority.

(Refer Slide Time: 04:58)

**Preemptive Scheduling Policies**

1. Round robin *FRONT* ReadyQ *4*
2. **Priority based** *BACK*
  - The readyQ need not be ordered on FCFS basis
  - It could be ordered on any other priority instead
  - For example: The process that has not run for the most time could get the highest priority
  - The scheduler could even assign a longer CPU timeslice for certain processes*1 sec*  
*100 msec* *timeslice*

Now, as the priority of processes are different, there is this possibility that a process with very low priority may in fact not get the chance to run in the CPU for a long duration of time. This could be used to actually enhance its priority. So, that is the idea, which is expressed in this example. A process that has not run for a very long amount of time could be given a higher priority.

Let us suppose, **there** we have an example shown in the diagram. It could well happen that the process with priority 2 does not get to execute for a long time. Maybe because after the process with priority 5 runs, the process with priority 4 runs, and the process with priority 3 runs, it is conceivable that new processes with priorities higher than 2 might get created and they would all enter the ReadyQ in front of the process with

priority 2. So, after this has happened for a significant amount of time, it is possible for the operating system to be designed with this kind of a consideration.

The process with priority 2 having been deprived of the CPU for a huge amount of time, could be given an enhanced priority, maybe a priority of 6, which would cause it to come to the head of the ReadyQ. So, this is a simple idea. It could be used by the operating system process scheduler to enhance the fairness as far as processes are concerned. So, by not treating all processes identically, processes which have been deprived could be given preferential treatment periodically. So, this is the idea of any priority based scheme.

As it happens that scheduler could even more... For example, we talked about the idea that associated with the operating systems scheduler, there could be duration of time called the CPU time slice. I gave you the example that one second or 100 milliseconds might be an appropriate CPU time slice. This may be decided based on properties of the processor, properties of the amount of time it takes to do a context switch, and so on. So, this is what we might think of as the CPU time slice (Refer Slide Time: 10:40). This is the maximum amount of time that a process would get to run on the CPU before it is preempted.

However, if there is a process which has not run on the CPU for a long time, such as, our process 2 in the example, which may ultimately get to run as a process with priority number 6, then it does get to run on the CPU and the scheduler could even give it an enhanced time slice. For example, if all the other processes when scheduled were getting 100 milliseconds, then this particular process, which had not got the CPU for a long time, might be given one second CPU time slice.

There are other dimensions... Not only is that the dimension of importance of a process as quantified by priority, this could also be used not only in the ordering of processes in the ReadyQ, but also in the amount of CPU time that they would get when they are scheduled for example. So, this actually opens up a whole possible design space of process scheduling policies. Designers of operating systems could use their discretion in deciding what setting of the different parameters of this designs space would be most appropriate to come up with a good fair efficient scheduling policy.



(Refer Slide Time: 11:46)

The slide is titled "Example: Multilevel Feedback Policies" in green text. The title "Multilevel Feedback" is circled in red. To the right of the title, the word "Policies" is written in red. Below the title, there are five bullet points in yellow. The first bullet point is "Used in some kinds of UNIX". The second is "A Preemptive, Priority-based policy". The third is "Multilevel: OS maintains one readyQ per priority level". The fourth is "Feedback: Priorities are not fixed". The fifth is "A process could be moved to a lower/higher priority queue for fairness". To the right of the text, there is a diagram showing five horizontal bars representing queues, labeled 1 through 5 from top to bottom. The top bar (priority 1) is the longest, and the bottom bar (priority 5) is the shortest. The word "FCFS" is written in red above the diagram. In the bottom right corner of the slide, there is a small video feed of a man in a white shirt speaking.

- Used in some kinds of UNIX
- A Preemptive, Priority-based policy
- Multilevel: OS maintains one readyQ per priority level
  - It schedules the process from the front of the highest priority non-empty queue
- Feedback: Priorities are not fixed
  - A process could be moved to a lower/higher priority queue for fairness

I am going to talk about one class of scheduling policies a little bit, which falls in the category of priority based. Generally, one hears them refer to as Multilevel Feedback policies, a family of policies. Policies of this kind were used in some variants of UNIX. So, ideas along these lines might be what you would see in a UNIX or Linux system. Now, just you describe briefly attributes of a multilevel feedback policy. In general, these policies are preemptive and they use priorities.

Now, let me try to explain such a policy by looking at each of these two terms: the term multilevel and the term feedback. Now, when you hear the term multilevel, you realize that there are many levels. The suspicion is that, there are many levels of queues; in other words, the operating system is actually maintaining one ReadyQ per priority level. So, for example, in the previous slide, I had talked about priorities in terms of numbers. I had used the numbers 6 5 4 3 2 to talk about the different possible priorities. The process with priority 6 was a very high priority or important process from the perspective of the scheduler. A process with priority 2 was a low priority process.

Now, the idea of a multilevel scheduling policy would be that, rather than maintaining just one ReadyQ, there would actually be one ReadyQ per priority level. So, one ReadyQ for priority level 6, one for priority 5, one for priority 4, one for priority 3, etcetera. There could be a number of processes in each of these queues. So, if there are two processes with priority 2, they would be the lowest in this diagram ReadyQ (Refer Slide

Time: 13:34). If there are three processes with priority 6, they would be in the top-most ReadyQ. So, in a sense, the operating system might choose to keep separate ReadyQs for each of the priority levels. Then, there is no question of reordering the elements in the queue because if you look at the queue for priority level 6, all the processes have priority level 6. Therefore, the question arises in what order should they be included in the queue. Then, the answer, which one could see clearly is that the queues could then be maintained as First Come First Served queues. All the processes in a particular queue have equal priority. Therefore, the queues by definition could be First Come First Served queues, which is what we normally understand by the term queue anyway, from a data structures course.

Now, the question becomes the following: If there are many ReadyQs, when the time comes to identify a process to schedule... in other words, to make as one running process from among all the processes and all the queues, which one should the operating system process scheduler pick?

Obviously, if there are any processes in the highest level priority queue, the process from the front... For example, if there are processes in the ReadyQ for priority level 6, then the process from the head or front of that queue could be scheduled. However, what if the ReadyQ for priority level 6 is empty? Then, it makes sense to have the policy design so that it then looks into the priority level 5 ReadyQ. If that too is empty, it could look into the priority level 4 ReadyQ and so on.

(Refer Slide Time: 11:46)

The slide is titled "Example: Multilevel Feedback Policies". It contains the following text:

- Used in some kinds of UNIX
- A Preemptive, Priority-based policy
- Multilevel: OS maintains one readyQ per priority level
  - It schedules the process from the front of the highest priority non-empty queue
- Feedback: Priorities are not fixed
  - A process could be moved to a lower/higher priority queue for fairness

Handwritten annotations include "Policies" in red at the top right, "FCFS" in red next to the first bullet, and a vertical list of numbers 2, 3, 4, 5 on the right side, each next to a small horizontal bar representing a queue. A video feed of a man in a white shirt is visible in the bottom right corner of the slide.

In short, one could describe the scheduling policy as schedule the process, which is at the front of the highest priority non-empty queue. So, start looking at the highest priority queue. If it is not empty, take the process from the front of that queue. If it is empty, go down to the next priority queue and so on. So, this gives a well-defined mechanism for deciding which process to schedule next.

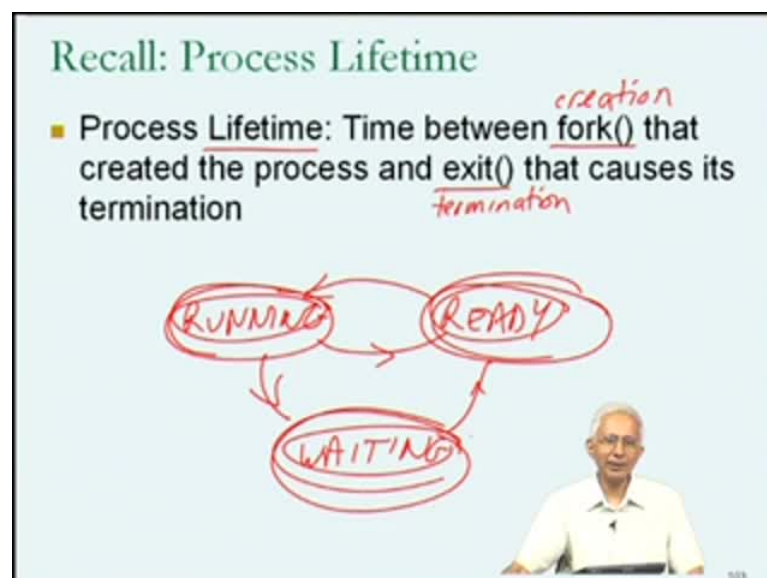
Now, the danger of this kind of a scheduling policy with only multiple levels is that, it is no different from what we had seen before unless there is some mechanism by which the priority of a process can change. So, if a process of priority level 2 can become a process of priority level 6, then this can be used by the operating systems scheduler to create a level of fairness to processes. So, that is where the term feedback comes into the picture.

When you hear the term feedback, you imagine some kind of a system in which some attribute of the system is used to change the properties of the system either through enhancing or reducing some attribute of the system, negative or positive feedback. So, the idea here in terms of scheduling might be that based on what has happened to a process up to now. One could decide whether to enhance or to reduce the priority of the process or possibly to keep the priority of the process the same as it is right now. So, that is what the term feedback is going to imply.

In this class of scheduling policies, **which** I have described this multilevel feedback, the word feedback is going to mean that the priority of a process is not fixed. During the lifetime of a process, its priority could increase; whereas, priority could decrease depending on how the operating system process scheduler is designed to deal with fairness.

We have seen an example, where it might make sense to increase the priority of a process. Suppose for example, that a particular process has been at priority level 2 for a long time and has not got CPU time, then that could be used as an indication that the priority of that process should be increased. On the other hand, if there is a process, which has received a lot of CPU time because it has been at level 6, then the fact that it is at priority level 6 and has received a lot of CPU time, could be used to reduce its priority level; maybe, bring it down to priority level 5. So, this combination of having multiple levels; **in other words, priorities and feedback;** in other words, the priority is not being fixed, but the priority of a process potentially increasing or reducing, could end up with policies that are not only dynamic in terms of how the **treat** processes, but also as fair and as efficient as could be designed. So, these are quite flexible process, scheduling policies compared to the simple alternative of something like Round Robin, which would have put all the processes into a single First Come First Served ReadyQ. After scheduling a process **state**, bring it to the back of the ReadyQ. So, the multilevel feedback provides a lot more flexibility to the designer of the operating system.

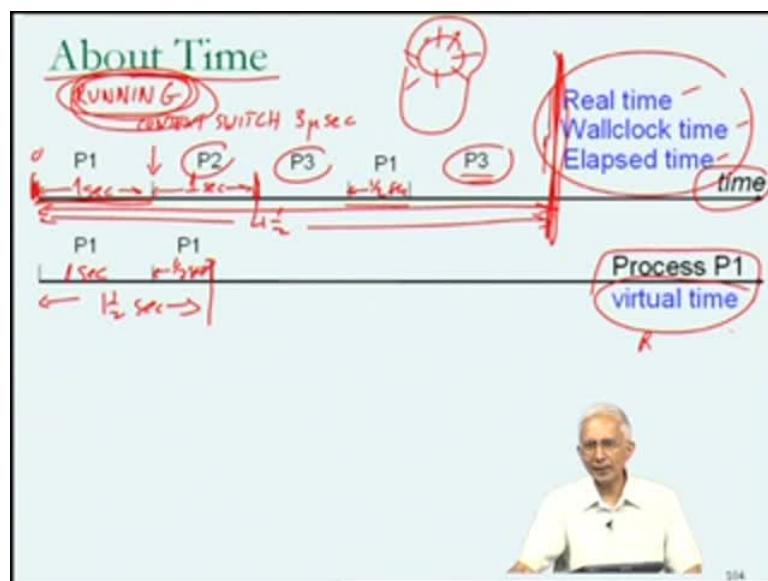
(Refer Slide Time: 18:09)



Now, given this rough introduction to how the operating system might be doing the scheduling of processes, we get the idea that when we submit a program for execution, we can think about its lifetime. I have used this term before. Let me just formally say what I mean by lifetime of a process, is the time between its creation which happens due to fork system call and its termination, which happens typically due to execution of the exit system call. So, this is the standard use of the term lifetime, that we have been throughout the course. I talked about the lifetime of a piece of data, the time interval between its deletion and its creation. We now talk about the lifetime of a process.

Now, if we think about the lifetime of a process and we want to quantify it in terms of time, we need to look at timelines. We have this clear picture in our minds that from the process state transition diagram, where we have at least three states: running, ready and waiting, we have this clear idea that process P1 could be running right now. Then, it might be in a waiting state because it does a I/O operation. When the I/O operation completes for example, it could end up in the ready state, then from there, it could go back into the running state, and then after being in the running state for some time, it might get preempted for which reason it goes to the ready state, and so on. So, a process during its lifetime is going to spend sometime in the running state, sometime in the ready state, and sometime in the waiting state. Therefore, we need to look at timeline to get a better idea of what lifetime might mean.

(Refer Slide Time: 19:53)



Let me just draw a timeline. As usual, I draw a timeline as a line which starts at some point in time and goes on indefinitely into the future. Now, let us consider what would be the various kinds of things we would see on the timeline. If there were many processes contending for the CPU; in other words, let suppose I have five or six programs running on a computer system, this might result in five or six processes. All of these five or six processes at some point or the other will hopefully end up being running, waiting, and ready. However, I am going to draw on this timeline is a history of the order in which the different processes actually get to run. Since it is only when a process is running that it is making use of the CPU and that its instructions are getting executed, its data values are getting modified, and so on. So, on this timeline, we are basically going to show intervals of time, when a particular process is running. That is the important state from our perspective. So, it might happen that the first process, which gets to run, is process P1. So, I show that by indicating an interval on the timeline and labeling that interval as P1, the suggestion is process P1 is running in that interval.

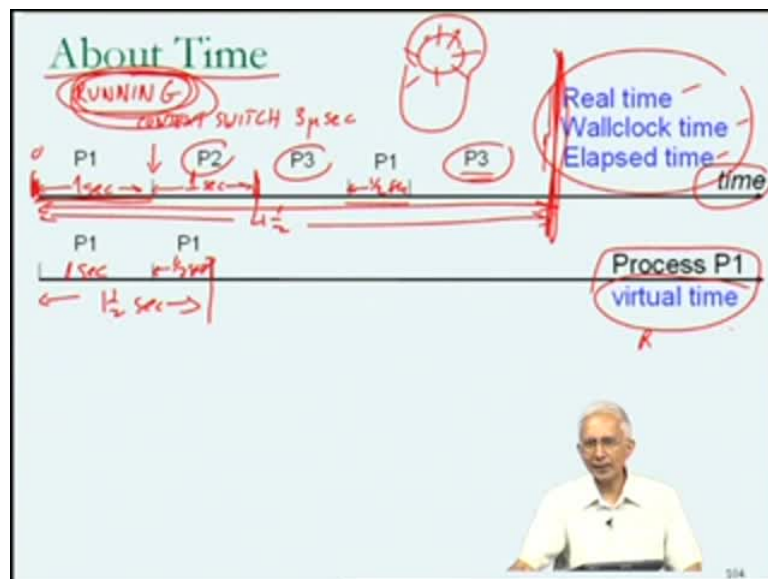
At this point in time, something happens – possibly process P1 gets preempted if this is a preemptive scheduling policy, which is why there is an end of the interval and conceivably process P2. Some other process, which was in the ReadyQ ends up being the running process. Just remember that we have this clear notion that, in order to switch from process P1 to process P2, there was a little bit of overhead; there was this thing called the context switch. So, the event that happened over here was the context switch from process P1 to process P2 (Refer Slide Time: 21:35).

You note that if the CPU time slice was one second, then it is conceivable that the duration the process P1 was executing was one second. It is conceivable that the duration for which process P2 will execute is one second and that the context switch itself is not instantaneous, but it is likely to take a small percentage of one second. For example, I had used the example last time of a few microseconds. So, it is something which takes time, but it is such a small amount of time compared to the **other events of interest, the intervals on the timeline**, that I will just show it by that very thin line. But do remember that the context switch does take some small amount of time. The amount of time that it takes to remember in memory the state of process P1 and to put into the hardware registers, the beginning state of process P2 as it should have been based on where it stopped execution the last time it was running.

The history as we have it now process P1 was the first process to be scheduled. This was followed by process P2. After process P2, was either preempted or did a long duration operation like a file I/O or a page fault or something like that. It is possible that process P3 executed and so on. So, there is going to be a history of the various processes as time went on. Do you notice that after P3, I show process P1 executing once again. So, P1, which had got its first chance to run on the CPU over here, (Refer Slide Time: 23:07) gets a second chance to run on the CPU over here.

Now, what are we showing on this timeline? We are showing the events of interest as far as the operating system is concerned. In fact, we are showing what you might describe as the Real timeline. I continue to show process P3. The implication from this diagram might be either the process P2 had finished execution, the program had terminated, or it could be that after process P1 had executed, the priority of process P2 became less in the priority of process P3, which is why process P3 followed process P1 the second time around. So, that is a vagary of the process scheduling policy. We will not worry too much about that.

(Refer Slide Time: 19:53)



The important thing to notice is that on this timeline, we are putting our information about what happened as far as the use of the CPU is concerned. This is the history of processes running on the CPU. It does not contain information about the history of

processes waiting or the history of processes spending in the ReadyQ and so on. Those could be separate kinds of timelines if that information was important to us.

Now, we do need a term to refer to this kind of time because we are actually going to talk about other versions of time. Since the timeline that we have drawn over here is showing the sequence of events that happened as far the CPU is concerned in reality, we could refer to this as Real time (Refer Slide Time: 24:29). This is not some kind of a fictitious time or an imaginary time such as when we talked about memory, each process had its imaginary or a virtual perspective on what addresses were. Here, the events that are happening are what actually happened on the CPU. Process P1 did run on the CPU first, then there was a small duration of time and the context switch occurred, then process P2 did run on the CPU, and so on. So, in that sense, one could refer to this time, the time that we have been drawing on this time line as the Real time. What do I mean by Real time?

What I mean by Real time is what you may also call Wallclock time. The term Wallclock time is used just to disambiguate from any other possible kind of Real time that you might have in mind. The picture that you should have in mind when you think of the wall clock is that on the wall behind you, there was a clock and you know that clock ticks, every second hand moves forward by one. If it is digital clock, it might be a little bit different, but you know the rate at which time moves for that clock. So, we are essentially saying that the time moves along this timeline at the same rate the time moves along the wall clock. In other words, we are talking about the same physical time that we deal with in our daily lives, the time as I see advancing along my watch.

In general, we think of these two terms: Real time and Wallclock time as being synonymous. In some books you may hear people talking about Real time; in other books, you may see people talking about Wallclock time. In fact, another term which is sometimes used to describe this concept of time is to refer to it as Elapsed time or the amount of time that has elapsed since the beginning of the timeline. In this case, it might be when the system was booted up, when I turned on the system. That might be the 0 that I see over here. So, these three terms: Real time, Wallclock time, and Elapsed time are to be viewed as being synonymous. In general, you will find all three of these terms used depending on the preference of the person, who is describing the time. Whenever you



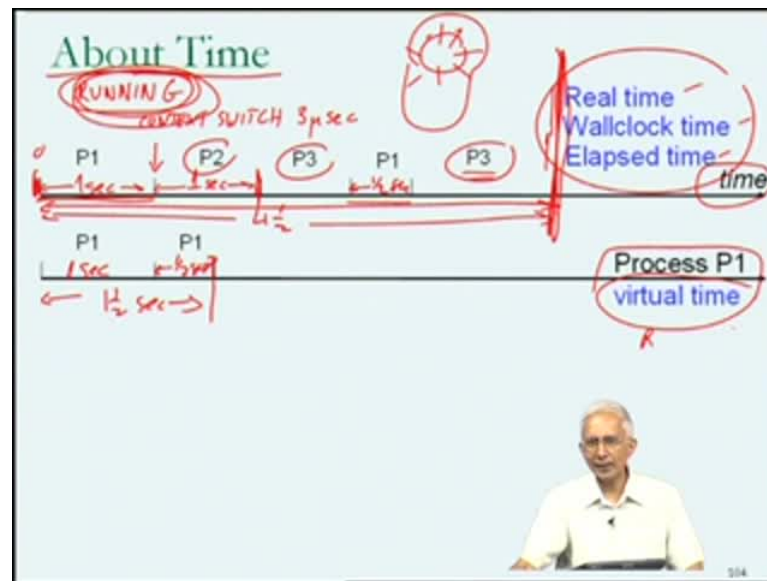
hear any of these terms, think about your watch or the clock on the wall, a correctly running clock on the wall.

Now, the question which has been in your mind ever since I put up this slide when I titled the slide about time would have been... Obviously, what are the kind of time could there be?

Now, thing to notice is that when we talked about main memory and the CPU at the beginning, it was fairly clear to us that whenever we talked about a memory address, we were talking about a main memory address until we introduced a notion of protecting one process from another and the notion of having virtual addresses, where we actually had separate addresses for each process. Since the process is the important entity as far as the execution of programs is concerned, the process is the unit of resource management by the CPU. It stands to reason that we might have another perspective of time on a computer system.

If there is another perspective of time, it could be a per process notion of time. In other words, I could have another timeline specific. Let us say to process P1. If I was to label this timeline, I could not just label the time since the word time is now ambiguous. It could mean the kinds of time that we had in the upper timeline or could mean the kind of time that we have on the lower timeline. Therefore, in future, whenever I talk about time, I will have to qualify by explaining whether I mean Real, Wallclock, or Elapsed time or whether I mean virtual time. If I am talking about virtual time, I have to qualify by which process I am referring to (Refer Slide Time: 27:59) in that timeline. Now, in this particular timeline very clearly, I am drawing a timeline, which is the virtual time of process P1. What we mean by the virtual time of process P1 is going to be related to its intervals of time when it is running.

(Refer Slide Time: 19:53)



Once again, our emphasis is on running as the timelines are going to show us running times **since the time that the processes** in the waiting state or in the ready state are not onto the direct control of the programmer; whereas, the times which are on this line are under the direct control of the programmer and therefore, are of interest to the programmer.

Here (Refer Slide Time: 28:36) we are trying to get a timeline in which all that is shown are events relating to process 1 and events relating to process 1 being running. If I had to fill up this timeline, it will show... If I am drawing the timeline up to this point in Real time, then the process 1 virtual timeline would end over here because we know that process 1 was running on the CPU for one second and then later on it ran on the CPU for maybe half a second. That looks like a smaller interval. So, it ran on the CPU for total of 1 and half seconds in this Real time interval (Refer Slide Time: 29:14). So, in the Real time interval, which might be about 7 or 8 seconds, process P1 ran for 1 and half seconds.

Let us say this Real time interval is 4 and half seconds. In this Real time interval of 4 and half seconds, process P1 ran for an interval of 1 and half seconds. What I am showing in this diagram is (Refer Slide Time: 29:35) the process 1's perspective of time. **This is** because you will note that the other intervals of time do not really exist for process 1 in the sense that it was not active on the CPU when process P2 was running. It was not

active on the CPU when process P3 was running. Therefore, this is a complete picture of time as far as process P1 is concerned in terms of its variables changing or its instructions being executed.

(Refer Slide Time: 30:07)

**Recall: Process Lifetime**

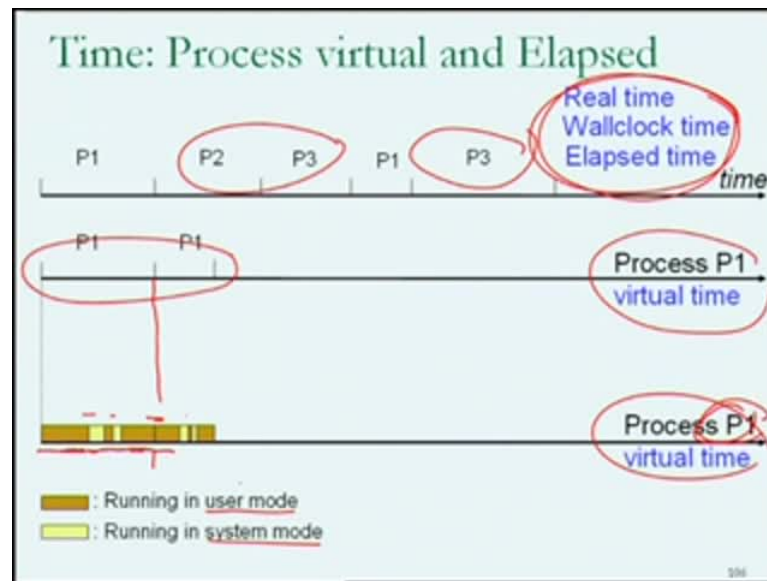
- Process Lifetime: Time between `fork()` that created the process and `exit()` that causes its termination
- At any given point in time, a running process is executing either in user mode or in system mode (when it makes a system call) *virtual*
- Can find out the total CPU time used by a process, as well as CPU time in user mode, CPU time in system mode

305

Now, we can **add** one more complication into such a diagram given our exposure to process lifetime. You will remember that when we talked about system calls, I introduced the idea that at any given point in time, a process could be running, but further it could be running either in user mode or in system mode. Up to now, we understood that a process would be running in system mode when it makes a system call. If it is just running in its ordinary mode, where it is executing instructions of its own program not the very special instructions within the system call, which is part of the operating system, then it is actually just running in user mode.

We had this idea that the operating system actually makes it possible for us to find out how much time our process was running in user mode and how much time our process was running in system mode. Over here, we actually had used the term CPU time. However, we now understand that what we are talking about here is this concept of virtual time, the time specific to this process. So, I could not use the term virtual time earlier, which is why we are talking about CPU time. However, now, we know that the CPU time is shared among many processes. In order to disambiguate what we mean by CPU time, we could more profitably use the term virtual time.

(Refer Slide Time: 31:24)



We could go back to the diagram we had just seen, where we had both the full picture of the CPU from the perspective of how it is time is being used among all the processes to the picture of time in terms of one specific process. We realized that the same the process P1 got 1 and half seconds of CPU time could be further refined to saying that if I looked at the virtual timeline of process P1, there would be some intervals of time when it is running in user mode and some intervals of time when it is running in system mode. So, I could actually color code.

In this particular diagram, I am going to show in brown the intervals of time when the process P1 is running in user mode and in yellow the intervals of time when it is running in system mode. We would then there expect to see a breakup of what is happening in those two time intervals, maybe along the lines of what you see over here. So, you will notice that process P1 starts by running in user mode. Then, possibly it makes a system call. So, it is running in the yellow system mode for some time. Then, it comes back to the user mode. Then, once again, it goes into the system mode possibly because of another system call. Then, it ends up running in user mode. At that point, apparently it gets context switch doubts. So, there is a small interval of time for the context switch, few microseconds. When it starts executing again, it is executing in user mode, etcetera.

When I told you that the operating system actually keeps track of how much time that your process is running in user mode and how much time it is running in system mode,

the implication was that this book keeping of - what is the sum total of the intervals of time which are brown as far as process P1 is concerned and what is the sum total of all the intervals of time which are yellow as far as process P1 is concerned is done by the operating system. It is something which you can get access to. So, there are mechanism, system calls through which you could request the operating system for this information. As we saw, this might be useful for you in improving your program. In using mechanisms that we will talk about later. So, the notion of the process virtual timeline... In other words, trying to restrict once the tension, what is happening to your process rather than the other process is which happen to be running on the system and have nothing to do with you.

In other words, concentrating on process virtual time rather than on what is really happening in terms of the movement of time as the world knows it may be of relevance to us in improving the qualities of our programs understanding, how our program is doing things. Further, the breakup between doing things in user mode and system mode.

(Refer Slide Time: 33:53)

The slide is titled "How is a Running Process Preempted?". It contains two main bullet points:

- OS preemption code must run on the CPU
  - How does OS get control of CPU from running P1 process to run its preemption code?
- Hardware timer interrupt (1 msec)
  - Hardware generated periodic event ←
  - When it occurs, hardware automatically transfers control to OS code (timer interrupt handler)

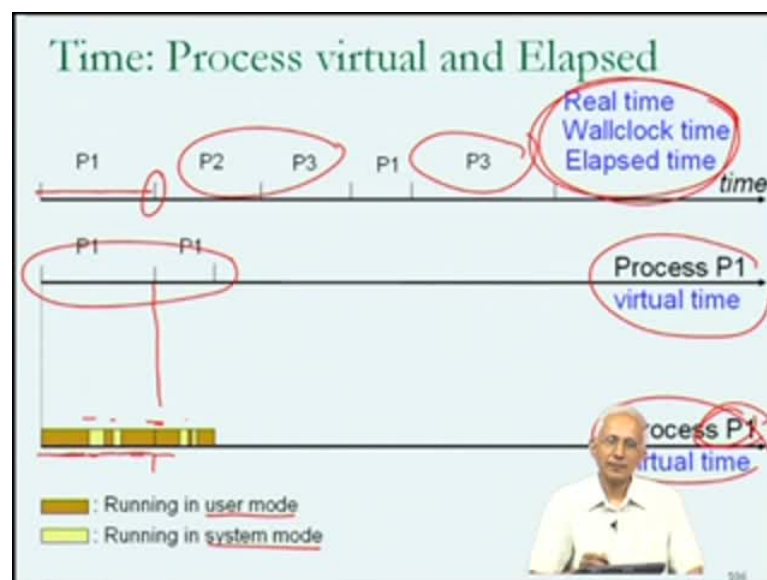
A presenter is visible in the bottom right corner of the slide.

Now, before we move forward, I will fill in a few gaps, which remain from our discussion of how the operating system manages CPU time. One interesting question, which would arise if one thinks about it deeply is the one on the screen - how is a running process preempted?

Now, you will recall that I mentioned that if one is using a preemptive process scheduling policy, the idea is that the operating system process scheduler will not allow a process to run indefinitely, but periodically. Let us say after it finishes its CPU time slice, it will preempt it. Now, if you bear in mind that preempted and switched to one of the ready processes rather than continuing execution of the process in question.

Now, if you think about this a little bit, there is a problem because the operating system itself is a piece of software. In order to preempt a process, the instructions within the operating system, which do the scheduling policy, which I will refer to as the operating system's scheduling code, must run on the CPU. What does the operating system's scheduling code do? Basically, the operating system's scheduling code will include instructions, which do the scheduling policy, which look into the ReadyQs, etcetera. It will include instructions, which will do the saving of the hardware context of the running process; it will include instructions, which do the restoring of the hardware context of the process, which is going to be scheduled. So, fairly large number of instructions, all of which I am describing as the operating system's scheduling code.

(Refer Slide Time: 35:32)



For this to happen, the instructions of the operating system must be running, but unfortunately, process P1 is running. Process P1 will continue to run until it is preempted. Hence, the problem (Refer Slide Time: 35:50). How come the operating system runs its scheduling code before process P1 has been preempted? Until process P1

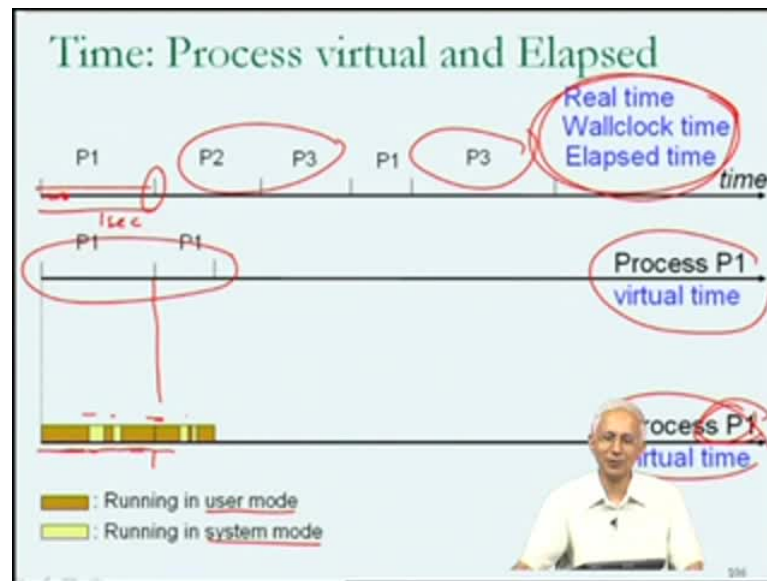
is preempted, process P1 occupies the CPU. So, this is the nature of the problem. So, very clearly, this is going to require some kind of a hardware support. This is not something with the operating system can manage as a software entity. This is just a rewording of the question, how does the operating system get control of the CPU from a running process P1 for example? P1 was a running process in our example in order to run its preemption code.

As I had indicated, this may actually require hardware support (Refer Slide Time: 36:28). The form that the hardware support takes is a small piece of hardware called the hardware timer. Essentially, when you hear the word timer you normally think of something which counts time. That is basically what this piece of hardware is doing. Basically, it is something like a clock, which periodically will try to grab control of the CPU. The way that it grabs control of the CPU is using a mechanism called an interrupt.

Now, what is the hardware timer, what does the hardware timer do? It periodically generates an event. The period of this event could be a few milliseconds. What I mean by period is the frequency with which the event occurs. If the hardware timer is such that it generates this hardware event once every millisecond (Refer Slide Time: 37:21), then the period would be one millisecond. So, it is a hardware generated event. So, it is a piece of hardware which is doing this part of the CPU.

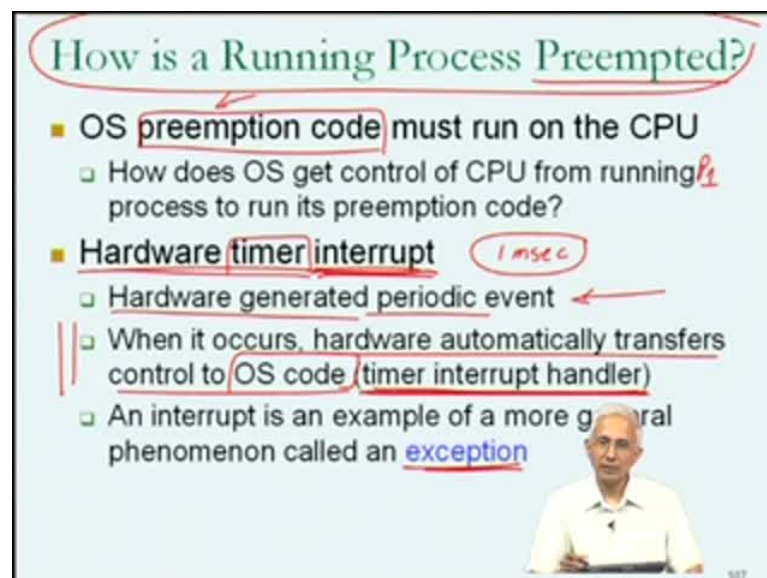
Now, the mechanism that is going to have to be provided within the CPU is whenever this kind of an event occurs, the hardware must automatically transfer control to the operating system. So, the hardware has to be built, the processor has to be built so that whenever an event, the kind mentioned in the first bullet occurs, automatically without the need for intervention from any form of software, control gets transferred to the specific operating system code, which is called the timer interrupt handler. So, this gives the operating system a mechanism through which it can grab control of the CPU. **This is** because we now have a guarantee that if the period of the hardware timer is 1 millisecond, once every 10 milliseconds, the small piece of code, part of the operating system, will get executed again.

(Refer Slide Time: 38:21)



Going back our timeline, we do the timeline like this. However, we now need to understand that once every millisecond, which is very frequently in this timeline... Remember: This interval is one second. Therefore, if the timer interrupt interval is 1 millisecond, then thousand times the operating is actually getting control in this interval.

(Refer Slide Time: 38:41)



We realize that with this mechanism, the operating system gets control whenever it wants to as far as the control of the CPU is concerned. So, very clearly, in those 1000 times that the operating system timer interrupt handler gets controlled, 999 times we do



not execute the preemption code and last time it will execute the preemption code. As a result of which process P1 gets replaced by process P2 through the context switch.

Now, the concept that we have over here, (Refer Slide Time: 39:15) where there are events in which the hardware automatically causes control to be transferred to a piece of operating system code, will require that the hardware be designed to do this. So, it must have a very well-defined interface for operating systems and hardware to interact with each other in this way. So, a very well-defined sequence of events must happen for this kind of a commitment to be made by the designer of the hardware.

Very clearly, we are talking about a slightly more general phenomenon than just to solve the problem, which is listed on the screen. In fact, the interrupt which I talked about over here, the hardware timer interrupt is merely an example of a more general phenomenon, which is known as an exception. So, I would like to spend a few minutes talking about exceptions before we move forward.

What we need to understand here is that there is something called an exception and today's processors are designed so that whenever an exception occurs... There could be many different kinds of exceptions. However, whenever an exception occurs, whatever happen to be running on the CPU before, control automatically gets transferred to a special piece of operating system code. Depending on what the exception is, a different piece of operating system code might be beneficial. For example, if the exception just occurred was a timer interrupt of the kind that we were talking about over here, (Refer Slide Time: 40:44) then control automatically gets transferred to a piece of operating system code called the timer interrupt handler, which would contain the appropriate activity as for as handling or dealing with timer interrupt is concerned.

(Refer Slide Time: 41:01)

**Exceptions**: TRAPS INTERRUPTS  
rare, unusual

- Certain exceptional events that occur during program execution, handled by the processor HW
- There are two kinds of exceptions

1. **Traps**: Synchronous, software generated
  - Page fault, Divide by zero, System call
2. **Interrupts**: Asynchronous, hardware generated
  - Timer, keyboard, disk

Apparently, there could be other kinds of exceptions. Therefore, we will need to learn a little bit more about this. So, let me first talk about exceptions. By its very name, we understand that exceptions are some kind of rare or unusual. So, the word exception here means rare, unusual, not in the common sequence of events, exceptional events. These are exceptional events that occur during program execution and automatically handled by the processor hardware in that. Depending on what the event is, a specific piece of operating system code would transfer control to automatically.

Now, in general, one could talk about two different classes of events. I will use the word classes rather than kinds of events. There are two different classes of exceptions. These two classes of exceptions are referred to as traps and interrupts. We had seen the word interrupt before; I talked about hardware timer interrupt. In other words, an interrupt or an exception generated by the hardware timer. So, in general, when we talk about exceptions, we need to remember that they are two kinds: traps and interrupts.

Now, a trap and an interrupt are different in that the trap is a software generated event and interrupt is the hardware generated event. So, we saw that the hardware timer is a piece of hardware, which periodically possibly once every millisecond generates an event. In that case, we call that an interrupt. Now, the suggestion from this classification is that there are also some kinds of exceptions, which are generated not by specially designed pieces of hardware, but by software. You will be wondering... I want to

actually emphasize what the terms synchronous and asynchronous mean here. It is not important for us in our current context. So, the question of what kind of exceptional situations might be generated by pieces of software?

In fact, we have already seen at least one. That is the page fault. If you think about it, a page fault occurs because... Let us say a load word instruction is executed. The effect of the load word instruction might be that an address is generated. It may be found out that the address is not an address, which currently has a page table entry in the TLB; or, it may not be an address, which has a valid page table entry. In that, the page table entry associated with that virtual address might not have the valid bit set. In some sense, one could say that this is a software generated exception. I view it as an exception because we know that automatically when this page fault occurs, control got transferred to the operating system page fault handler. That is the common term. You will recall that we talked about an exception being handled. The piece of operating system code, which handles the exception, is what we refer to as handler.

We had previously seen what happens as far as the page fault handler is concerned. Now, we can think of the page fault as being this kind of an exceptional event; it is relatively rare. Hopefully, it does not happen on every memory reference. However, when it does happen, the processor has to be diverted from what it is currently doing to actually executing the operating system's piece of code called the page fault handler.

(Refer Slide Time: 41:01)

**Exceptions**: TRAPS, INTERRUPTS  
core, unusual

- Certain exceptional events that occur during program execution handled by the processor HW
- There are two kinds of exceptions classes LW R, (8/R29) SYS CALL

1. **Traps**: Synchronous, software generated handler
  - Page fault, Divide by zero, System call
2. **Interrupts**: Asynchronous, hardware generated
  - Timer, keyboard, disk file read

We have also seen that the system call is something which would be caused by the execution of an instruction. In other words, the sys call instruction in the MIPS instruction set. This call instruction is an ordinary user mode instruction; it is not a privileged instruction. However, when this instruction is executed, control automatically gets transferred to the operating system region of code, which executes that particular system call **and through system call handler of some kind conceivably**. Once again, it was a situation, where control automatically got transferred to a piece of the operating system code. We have seen both of those before.

Let me just mention that it is also possible that there are other events of this kind such as, if there is a program, which executes a divide instruction, which results in a divide by zero, things could be set up so that this is handled by a divide by zero trap handler. So, different kinds of traps for different kinds of exceptional situations could be caused by the execution of instructions.

Interrupts on the other hand are exceptional situations which are explicitly generated by the hardware. We have seen one example of an interrupt. That was the hardware timer interrupt. We saw that this was very important from the perspective of the design of the operating system since it is only through having a periodic timer interrupt that the operating system could periodically get control of the CPU in order to do its management of the resources; otherwise, the operating system would lose control of the CPU to infinite loop process and never be able to recover it.

Now, we could also think about other pieces of hardware in the system and ask the question, for example, when I hit a key, how does that mechanical device on the keyboard cause any information to enter into the computer, which is as far as we know made out of circuitry? We can now look at this from the perspective of the following sequence of events. When you hit a key, mechanically, something is happening inside the keyboard. The keyboard itself contains circuitry. The pressure on the key is going to call some of that circuitry to generate a keyboard event or a keyboard interrupt.

Essentially, what is going to happen when you press the key is that the keyboard interrupt gets generated; control gets transferred to a small piece of operating system code called the keyboard interrupt handler. What the keyboard interrupt handler does is to transfer the character, which you had typed in. So, if you had pressed the A key, then

within a buffer within the keyboard, the character A is going to be generated and the character A is going to get transferred by the keyboard interrupt handler into somewhere in memory. So, you may have typed the key and then you may have included in your program some kind of a scanf to read input. Ultimately, from the keyboard, the keyboard interrupt causes the piece of character to get transferred and then the data will go to your buffer, which you have mentioned in your scanf call. So, all of this is triggered by an interrupt as far as the keyboard is concerned.

Similarly, operations on disks also may have interrupts associated with them. For example, we talked about the fact that when a process does a file read operation, this is going to cause a read operation to start on the disk. Now, this is an operation, which could take conceivably a large number of milliseconds to complete. So, the process itself is going to go into a waiting state. It was in the ready state when it executed the file operation, but the process may go into the waiting state at the discretion of an operating system designer. Subsequently, the disk I/O operation will complete and the data will be available somewhere in the disk ready to be transferred from the processor to memory. In doing this, it is possible that the disk can indicate that the I/O operation has completed possibly by an interrupt to grab the attention of the processor, to execute a disk I/O interrupt handler routine, and to do the transfer of the data. So, it is conceivable that things could have been setup that way.

Associated with different input and output devices or with different pieces of hardware in the computer system, there might be generation of interrupts in order for the interaction to happen. Therefore, traps are important from the perspective of the events that you generate within your program. Interrupts are very important from the perspective of interaction with the outside world and in particular the I/O devices. We have now seen the special case of the timer interrupt handler important for the operating system design. The very fundamental operating system design may require this existence of a timer interrupt handler as we had described it.

(Refer Slide Time: 49:29)

### What Happens on an Exception

*PC = 0x1000*

- 1. Hardware**
  - Saves processor state
  - Transfers control to corresponding piece of OS code, called the **exception handler**
- 2. Software (exception handler)**
  - Takes care of the situation as appropriate
  - Ends with **return from exception** instruction
- 3. Hardware (execution of RFE instruction)**
  - Restores the **saved processor state**
  - Transfers control back to the **saved PC value**

*R1*

*R29*

Now with this idea that the hardware must be designed to handle exceptional events and that the exceptional events could either be traps or interrupts, I will sketch out in more detail what happens when an exceptional event actually does occur. We have seen that when an exceptional event occurs, at the beginning, an interrupt or a trap gets generated. I will lay the blame for this at the hardware.

(Refer Slide Time: 49:45)

### Exceptions

*TRAPS: rare, unusual*

*INTERRUPTS*

- Certain exceptional events that occur during program execution **handled by the processor HW**
- There are two **kinds** of exceptions *classes*
  - 1. **Traps: Synchronous, software generated**
    - **Page fault, Divide by zero, System call**
  - 2. **Interrupts: Asynchronous, hardware generated**
    - **Timer, keyboard, disk, file read**

*LW R1, -8(R29)*

*SYSCALL*

Now, if you think about the traps that we talked about, if there is a page fault, the question of how is the page fault identified will arise. You will realize that the page fault

occurred because of an instruction in your program and hence, we think of it as a trap. However, the page fault will actually be recognized by the memory management unit when it tries to translate the address. Similarly, the system call is an instruction in your program, but the fact that this is a system call is identified by the hardware, which executes the instruction, the instruction fetch, instruction decode hardware. So, ultimately we could lay the blame for handling the exception initially at the hardware.

(Refer Slide Time: 50:23)

**What Happens on an Exception**

*P1 PC = 0x1000*

- 1. Hardware**
  - Saves processor state
  - Transfers control to corresponding piece of OS code, called the **exception handler**
- 2. Software (exception handler)**
  - Takes care of the situation as appropriate
  - Ends with **return from exception** instruction
- 3. Hardware (execution of RFE instruction)**
  - Restores the saved processor state
  - Transfers control back to the saved PC value

308

I will start the description of what happens on an exception with the hardware. If it is a trap, there is some prior history relating to the software, which we have already studied adequately. So, what should the hardware do when it realizes that an exception has occurred? It could be a trap; it could be an interrupt.

Now, the first thing that it will have to do is, the hardware will have to be designed to bear in mind that in the very near future, control is going to have to be transferred to an operating system routine called the handler. It could be a page fault handler, timer interrupt handler depending on what this particular exception is. However, control will have to be transferred to the handler, which is a special piece of operating system code. This is going to mean that whatever is currently executing on the processor, which could be process P1 executing in user mode, is going to have to be stopped for the moment.

If one is going to stop the execution of process P1 for the moment, then its current state will have to be saved because later on the state will have to be restored. Therefore, the first thing that the hardware will do when an exception occurs, it cannot just immediately transfer control to the handler; it will have to save the processor state in memory. After it has saved the processor state, since the state will be necessary when one is coming back from the exception, it can transfer control to the corresponding operating system exception handler. As I said, it could be the page fault handler, timer interrupt handler.

Now, the control transfers to the software, which is the exception handler. What does the software do? Again, depending on what the exception is, the software would have been written to do certain action. For example, we saw exactly what action the page fault handler would be designed to do. So, the page fault handler does that. So, I will just describe this for saying that the appropriate action is taken as per the situation. Since control was transferred to the correct exception handler, correct instructions will be executed for the handling of that particular exception whatever it may be. What about after the exception has being handled?

All the codes in the exception handler have been successfully executed, what should be done next? Then, it is time to transfer control back to the process, which was running when the exception occurred. Now, one will need an instruction to do this. One cannot just do this using the jump registers in the type of instruction that we use to return from a function call for various reasons. So, this would be done with a special instruction, which might be called the return from exception instruction.

Every instruction set is going to contain such an instruction. We have not talked about this instruction before in talking about the MIPS one instruction set because this is not the kind of instruction that you or I would use in our ordinary programs. This is an example of a privilege instruction; it is only going to be used at the end of an exception handler. An exception handler is a piece of operating system code. Therefore, it was not necessary for us to know about it. Since we are worried about ordinary user programs, none of us at the moment as far as we know is going to have to worry about writing an operating system. So, exceptional handler ends with execution of special instruction, which would cause control to be transferred back to wherever it should.



(Refer Slide Time: 50:23)

The slide is titled "What Happens on an Exception" in green text. At the top left, there are handwritten notes in red: "P1" with a circle around it and "PC = 0x1000". The slide content is organized into three numbered steps:

- 1. Hardware**
  - Saves processor state
  - Transfers control to corresponding piece of OS code, called the exception handler
- 2. Software (exception handler)**
  - Takes care of the situation as appropriate
  - Ends with return from exception instruction
- 3. Hardware (execution of RFE instruction)**
  - Restores the saved processor state
  - Transfers control back to the saved PC value

Red circles highlight "Saves processor state" and "Restores the saved processor state". A red arrow points from "return from exception instruction" down to "execution of RFE instruction". A small number "508" is in the bottom right corner.

We are now back to hardware, the execution of the RFE or return from exception instruction. What should happen in this instruction or as an effect of this instruction? The answer is, this instruction should cause the saved processor state to get restored into the processor; then, it should cause control to be transferred back to the appropriate PC value. So, whatever process happened to be executing... It might have been process P1 executing at the PC value, X 1 0 0 0. That is where control should be transferred back to process P1. We are back to process P1 because we have restored its saved state and resuming from the correct PC value, which we knew because that was part of the saved processor state.

With this, we have some idea about what an exception is and we have some idea about how an exception is handled. Just to sum up what we have seen in today's lecture: In today's lecture, we saw how operating system process scheduling policies might be made little bit more flexible than what we had seen in the previous lecture. Largely, from the perspective of allowing the scheduling to be done as fairly and efficiently as possible, we saw that it is important for the operating system to have some way to grab the CPU from the currently running process; otherwise, the very idea of preemption would not be possible. I mentioned how this is typically done through a mechanism called an interrupt by having special piece of hardware called a hardware timer, which periodically generates an exceptional event called an interrupt.

Interrupts are important events in a computer system. They are examples of more general events called exceptions. There are two kinds of exceptions: traps, which are software generated such as page faults or system calls and interrupts, which are hardware generated typically by I/O devices or a piece of hardware like a timer. The hardware must be designed if it is going to use operating systems of the kind that we have seen. To handle these kinds of exceptional events by saving the state of the processor, transferring control to an appropriate operating system handler, and then returning control with the restored context of the originally running process.

With this, we have a fairly good idea about what is happening behind the scenes when our programs execute from the perspective of operating system process scheduling. We will continue with a few more closing comments in this area in the next lecture.

Thank you.