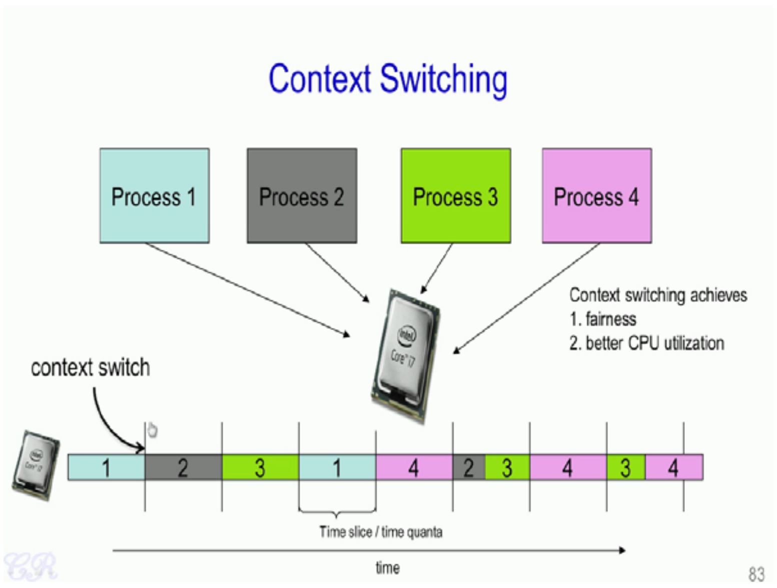
Introduction to Operating Systems

Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week - 04
Lecture – 17
CPU Context Switching

Hello. In this video we will look at CPU Context Switching. We will see how the operating system enables multiple processors or rather enables multiple processes to share a single CPU.

(Refer Slide Time: 00:30)

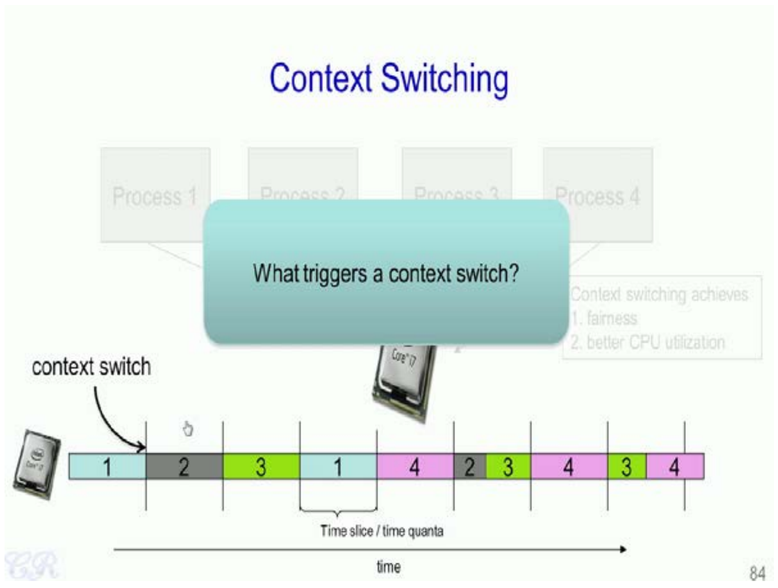


Context Switching

Let us consider this particular figure, where we have a single CPU in the system which is shared among multiple processes. So, the operating system by a feature known as multi tasking enables that this CPU is fairly shared among the various processes. So, in a multi tasking environment or rather in a multi tasking enabled operating system. The OS would allow one process to execute for some time, and then there is a context switch. So, during this context switch the process 1 would be stopped and a new process, in this case process 21 will begin to execute.

Now the operating system ensures that when process 1, stops executing its state is saved in such a way that after sometime it can be scheduled back into the CPU and can continue executing from where it had stopped. So, as a result in a multi tasking environment such as this, processes execute for brief interval of time known as time slices. So, for instance process 4 executes in this time slice then there is a period where it does not execute and then it continues executing in this time slice and so on. So, in this video what we are going to see is how the operating system ensures that a context switch occurs.

(Refer Slide Time: 02:00)

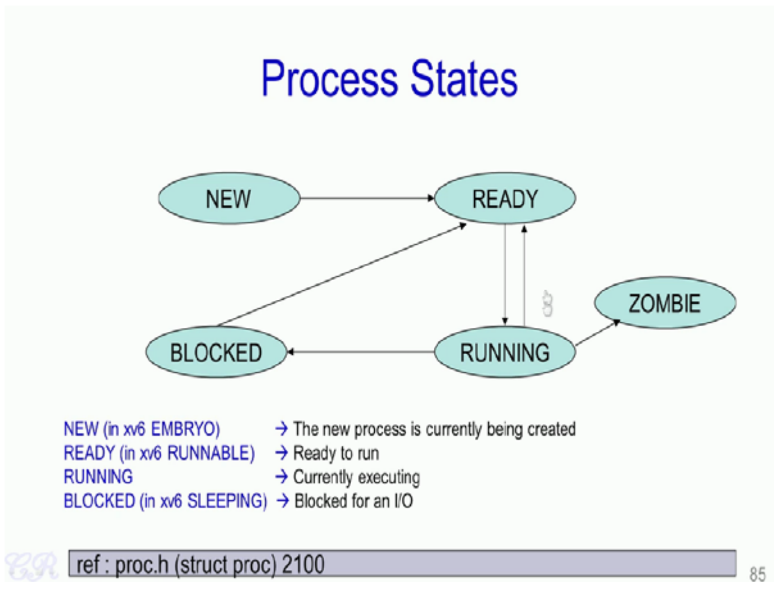


Context Switching

What triggers a context switch?

To begin with, we will see what triggers a context switch in an operating system.

(Refer Slide Time: 02:07)



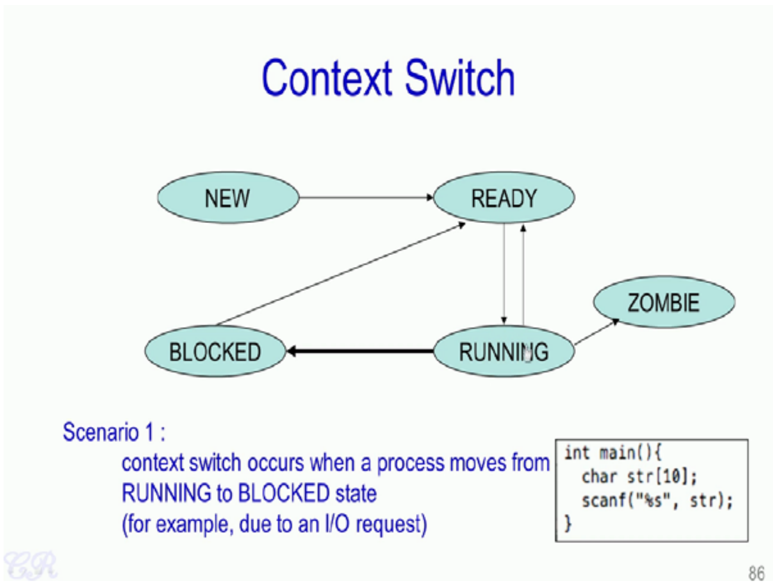
Process States

NEW (in xv6 EMBRYO)	→ The new process is currently being created
READY (in xv6 RUNNABLE)	→ Ready to run
RUNNING	→ Currently executing
BLOCKED (in xv6 SLEEPING)	→ Blocked for an I/O

ref : proc.h (struct proc) 2100

To answer this particular question, we need to go back to the process state diagram. So, in a previous video on the processes, we had seen that when a process gets created to the time it exits it goes through several difference states, and this is represented by this process state diagram. What we will see now is how this various states will trigger a context switch to occur.

(Refer Slide Time: 02:31)



Context Switch

Scenario 1 :

context switch occurs when a process moves from RUNNING to BLOCKED state

(for example, due to an I/O request)

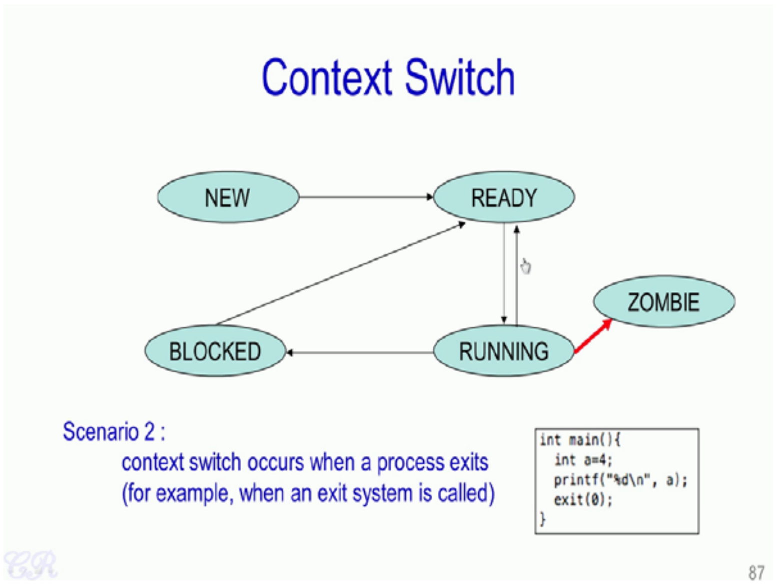
```
int main(){
  char str[10];
  scanf("%s", str);
}
```

Let us see this first scenario about how a context switch gets triggered. Let us consider this particular program which essentially invokes this scanf therefore, it requires a user to input something through the keyboard.

Let us say this function is executing as a process in the system, and it is currently in the running state, which means that it is currently holding the CPU and is currently executing in the CPU. When the scanf function gets invoked what would happen is that the process now requires to be blocked until a user inputs data through the keyboard. So, as a result the state of the process changes from the running state into the block state, now the process will remain in the block state until the user has input data into the console.

In order to utilize time effectively, what the operating system is going to do is that it is going to trigger context switch. So, that another process could then be executing in the running state and will have the processor.

(Refer Slide Time: 04:25)



Context Switch

Scenario 2 :

context switch occurs when a process exits

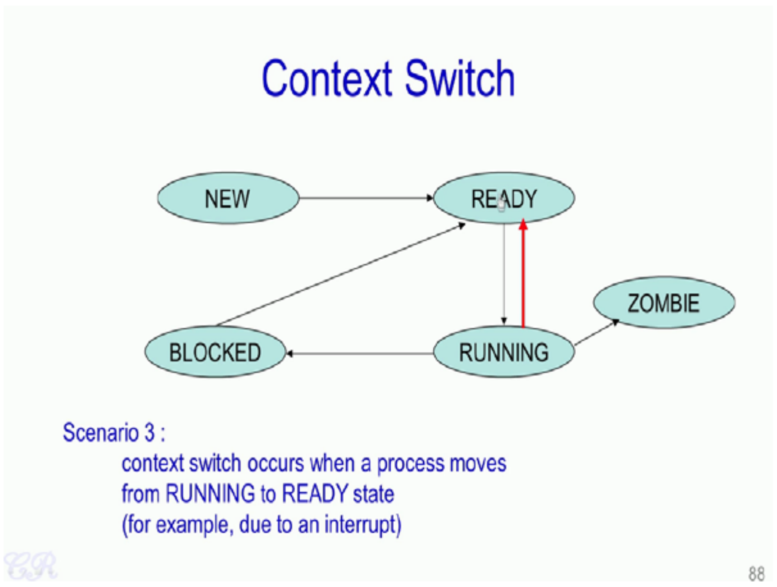
(for example, when an exit system is called)

```
int main(){
  int a=4;
  printf("%d\n", a);
  exit(0);
}
```

Let us take the second scenario where context switch can occur. Let us say that we are considering this particular program, where there is a printf and then there is an exit. As we know and we have seen before in a previous video that the exit is a system call which results in the process going into this zombie state.

At the end of the exit system call, the operating system will trigger a context switch. So, this will ensure that the current process which is just in exiting mode will not have the CPU resources anymore, but rather a new process will be assigned the CPU therefore, a new process will come from the ready state into the running state.

(Refer Slide Time: 04:31)



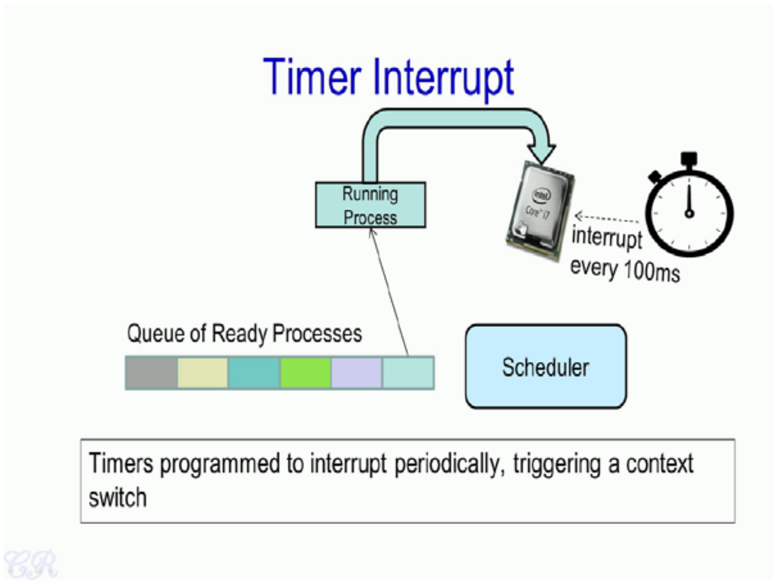
Context Switch

Scenario 3 :
context switch occurs when a process moves from RUNNING to READY state (for example, due to an interrupt)

88

Now, let us look at the third scenario that can occur. So, if a process is in a running state, and an event such as a hardware interrupt occurs, then it could lead to a context switch. So, for instance if a process is currently executing or currently holding the CPU and an interrupt occurs it moves from the running state into the ready state. As a result of the interrupt in would cause an interrupt handler to execute, and at the end of the handler a new process may be executed in the CPU, that is a new process will move from the ready state in to the running state.

(Refer Slide Time: 05:14)



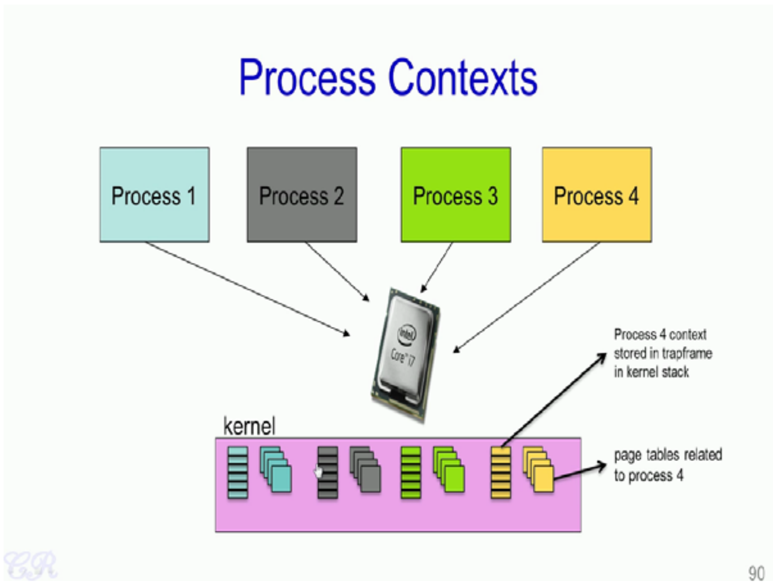
Timer Interrupt

Timers programmed to interrupt periodically, triggering a context switch

A very popular interrupt in this context is the timer interrupt. Now all systems have a timer within them. So, this timer is configured to send interrupts periodically to the CPU. The period could be anything from 10 milliseconds to 100 milliseconds, and may vary from system to system or when the timer interrupt occurs, the OS gets triggered and it causes a CPU scheduler to execute.

Now the CPU scheduler looks at the queue of processes which are in the ready state, and then based on some algorithm will chose a particular process. So, this process would then be moved from the ready state into the running state, and it would be this process that would execute in the CPU until the next interrupt that occurs. So, in this process every 100 milliseconds for instance the scheduler would pick a new process to execute and that process would then hold the CPU for the time slice of 100 milliseconds.

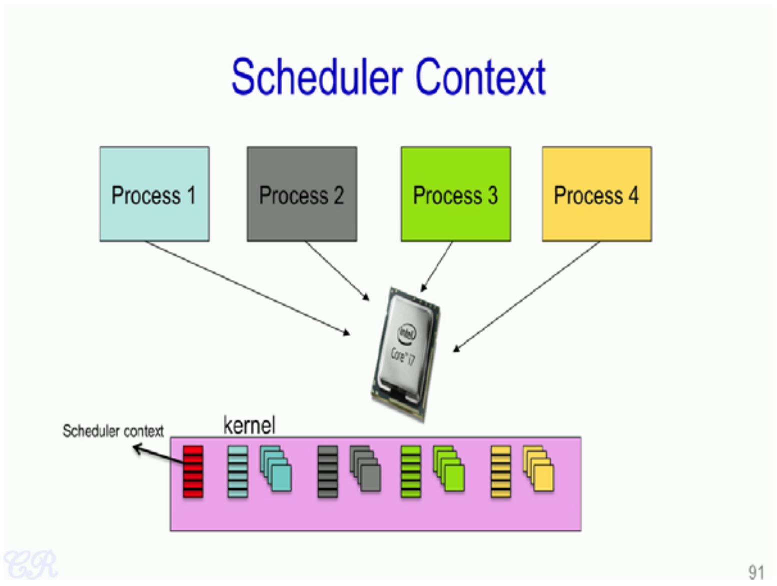
(Refer Slide Time: 06:22)


Process Contexts
Process 1
Process 2
Process 3
Process 4
Process 4 context stored in trapframe in kernel stack
kernel
page tables related to process 4
90

Now let us look more into detail about how a context switch occurs. So, in a previous video we had seen that corresponding to each process in the system, the kernel stores some meta data essentially there are three meta data for each process they are; the process control block, the kernel stack and the page tables corresponding to that process.

Now we have also seen in an earlier video, that when an interrupt occur this context of that process is stored in the kernel stack, in what is known as the trapframe. So, this context would allow the process to restart execution from where it had stopped. So, in this particular figure for instance, each process has its own trapframe or own kernel stack which was the trapfarme when an interrupt occurs, and its associated page tables. Similarly, process 3 has its own trapfarme and a page tables process 1 and process 2 as well.

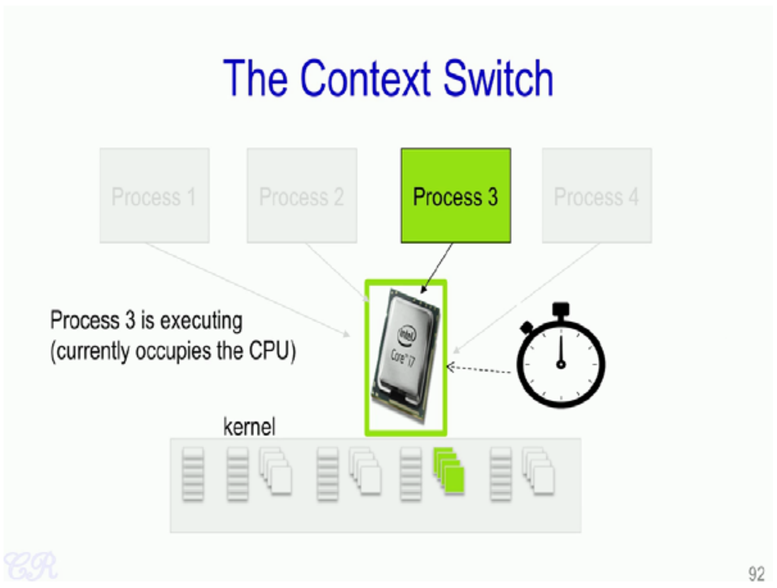
(Refer Slide Time: 07:31)



Scheduler Context

In addition to this, the CPU scheduler has a context which is also stored in a separate stack. So, this is known as a scheduler context and it is used while doing a context switch.

(Refer Slide Time: 07:45)

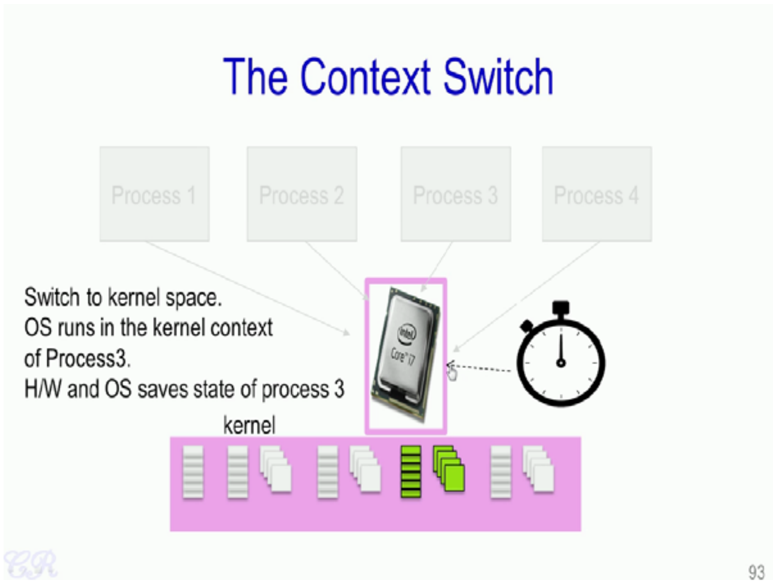


The Context Switch

Let us look more into detail about how a context switch occurs, let us assume that the user process 3 is currently executing in the CPU, that is the user process 3 currently holds the CPU, also as a result of this it is the process 3's page table which is currently active, that is for every instruction fetch every memory load or memory store it is the process threes page table which does the translation from the logical address into the physical address.

Now, when an interrupt occurs we have seen that there is a switch from the user mode into the kernel mode.

(Refer Slide Time: 08:26)

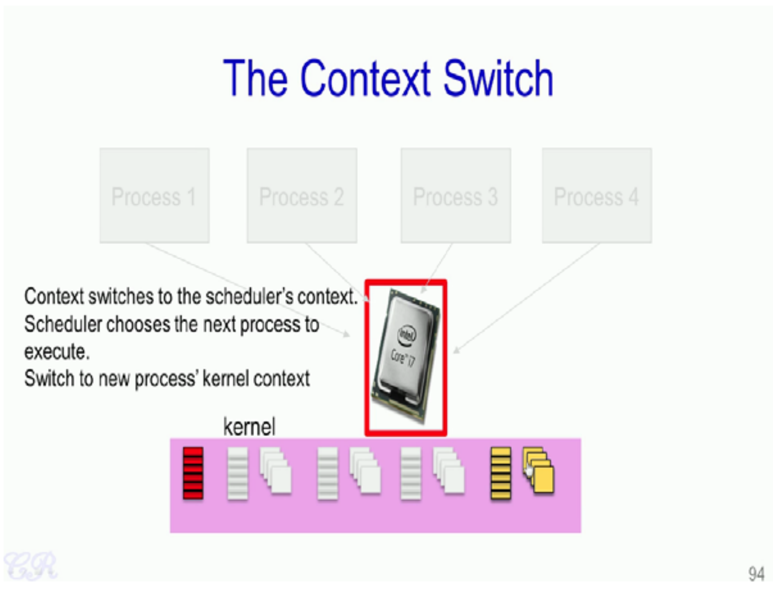


The Context Switch

Switch to kernel space.
OS runs in the kernel context of Process3.
H/W and OS saves state of process 3

Therefore the kernel begins to execute again, and it is the kernel which will now hold the CPU. All instruction that are being executed in the CPU, thus become belong to the kernel code also as a result of the interrupt what we have seen before, was that the entire context of this process 3 get stored in the kernel stack of that process. This context is stored in a structure known as the trapframe, and this trapframe as we had seen before has sufficient amount of information that would allow process 3 to restart executing from where it had stopped. The next things that happens is that the kernel determines that the interrupt occurred was due to the timer, and it would invoke the scheduler.

(Refer Slide Time: 09:19)

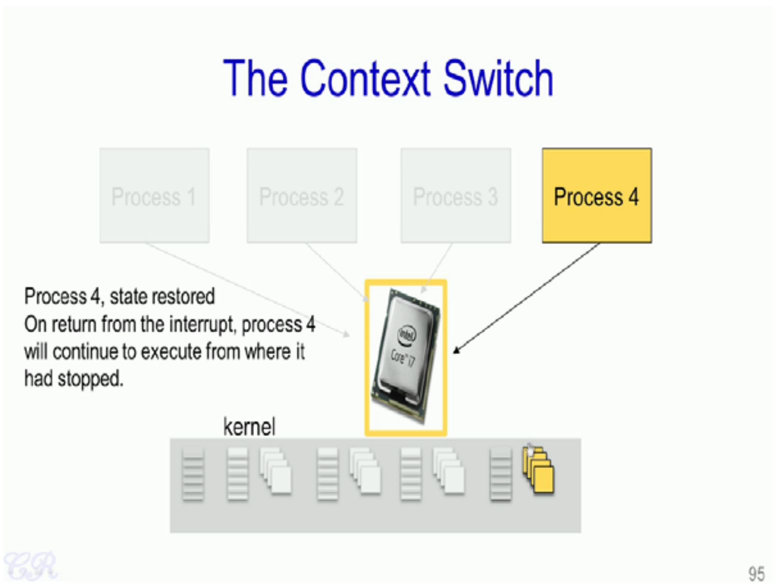

The Context Switch

Context switches to the scheduler's context.
Scheduler chooses the next process to execute.
Switch to new process' kernel context

The scheduler would then switch from the kernel stack of the user process, to its own stack or the scheduler stack. Therefore, it would obtain the scheduler context, the scheduler then chooses from the ready list the next processes to be executed in the CPU and then there is a switch to the new processes kernel context. Thus we are moving back from the scheduler context, back to the new processes in this case process 4 that has been selected by the scheduler, and we switch back to process fours page tables and process fours kernel stack.

Now, recall that each of these stacks have the trapframe. Therefore, even process 4 based on its previous execution, has a trapframe which contains the entire context of web process 4 had stop executing.

(Refer Slide Time: 10:23)



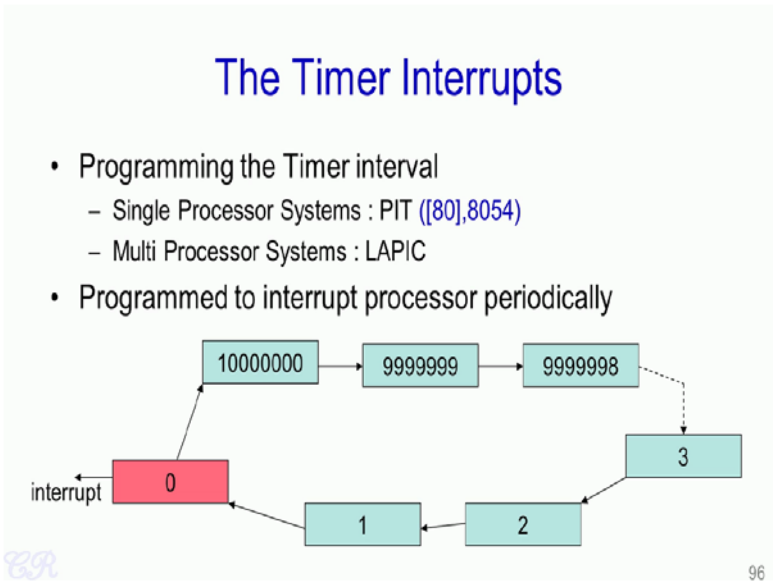
The Context Switch

Process 4, state restored
On return from the interrupt, process 4 will continue to execute from where it had stopped.

Thus, when the timer interrupts returns with the Irate call this contents of the process 4's trapframe gets restored. And as a result of this, processes four will continue to execute. Also what happens is that, we are switching to process fours page table. Now as process four executes instructions, its instruction are fetched a memory loads and memory stores are translated by process fours page tables.

In this way every interrupt that occurs would cause a switch to the kernel, and if it is a timer interrupt and requires a new process to be executing. The new process would be selected by the scheduler and its context would be restored. Also as we have seen the new processor page table would then be made active.

(Refer Slide Time: 11:21)



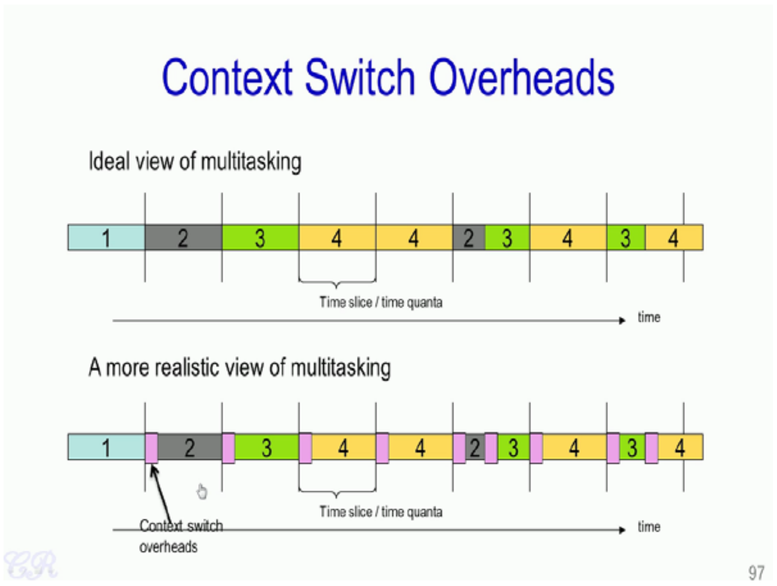
The Timer Interrupts

- Programming the Timer interval
  - Single Processor Systems : PIT ([80],8054)
  - Multi Processor Systems : LAPIC
- Programmed to interrupt processor periodically

Now, let us look how timers actually interrupt processes, as we have mentioned before all systems have a timer present in them. In legacy systems this is known as the PIT or programmable interval timer which is a dedicated chip present in the mother board. In modern systems or in current day systems, timers are present in the LAPIC. So, in both these devices there is a counter present internally. So, this counter can be programmed to start counting from a particular number.

For example, this large number, every clock cycle this counter is actually decremented until it reaches 0. At the end of the count that is when it reaches 0 the counter is load back loaded back with the original value and this process continues. So, essentially the counters keep going from this large value to 0 and then keep cycling across these values. Now when 0 is obtained by the counter and interrupt is assorted to the CPU. Thus at periodic intervals the CPU would receive interrupts from the timers. So, these interrupts are then used to decide on the next process that is going to execute in the CPU.

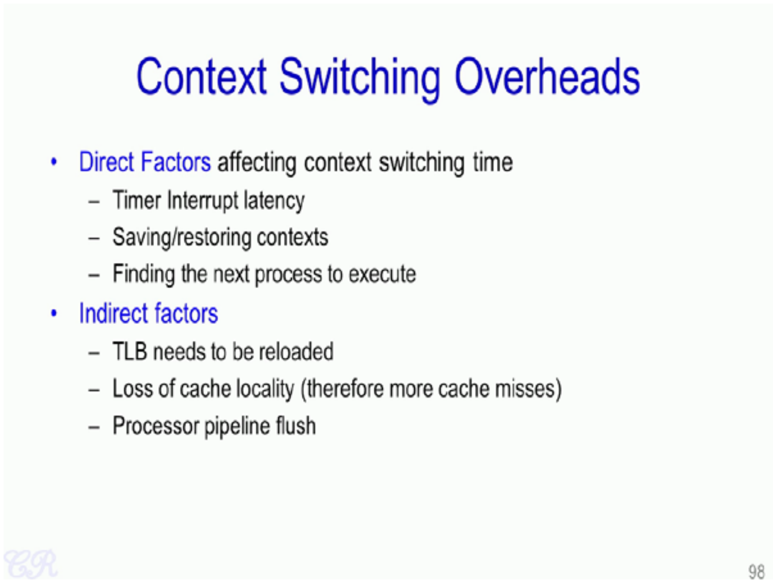
(Refer Slide Time: 12:42)


Context Switch Overheads

This figure over here depicts the ideal view of multiple tasking. So, in this particular ideal view, a process will continue to execute until its time slice completes. When its times slice completes a new process or another process will execute in the CPU. In a more realistic view of multi tasking, and also as we can infer from the previous slides is depicted in this particular figure at the bottom.

o, in this view or the more realistic view the process will continue to execute as usual until its time slice completes, and then instead of the next process immediately being context switched into the CPU, the kernel will execute, where in the kernel will do various things like, handling the interrupts doing the various jobs and also executing the scheduler where a new process is chosen. And only after this all this occurs will this next process execute in the CPU. So, this time difference between when the first process is preempted from the CPU to when the second process start to execute is the context which over heads; now, this context which over heads could be significant.

(Refer Slide Time: 14:04)



Context Switching Overheads

- Direct Factors affecting context switching time
  - Timer Interrupt latency
  - Saving/restoring contexts
  - Finding the next process to execute
- Indirect factors
  - TLB needs to be reloaded
  - Loss of cache locality (therefore more cache misses)
  - Processor pipeline flush

CR

98

The factors affecting the context switching over heads can be classified as either the direct factors or the indirect factors. Among the direct factors these are the three examples of direct factors, and they are essentially quite straight forward and easy to understand. For instance, the timer interrupts latency or for that matter any interrupt latency will add up to over heads in the contexts switching. Essentially there would be time taken for saving and restoring context for the various processes. In addition to this over heads are also caused by the scheduler, which needs to choose the next process to execute in the CPU.

The indirect factors are more settled and more difficult to understand. So, these are three examples of indirect factors. So, the first factor is that the TLB needs to be reloaded. So, TLB is the translational look a side buffer. So, it is a cache which stores recently use page mappings. So, when we switch from one process to another process, we know that the page table changes from the earlier process to the new process that is we have a new set of page tables which becomes active.

As a result the TLB needs to be flushed again. Now refilling the TLB will take time and result in over heads. So, another aspect is the loss in the cache locality. So, as we know cache memories work on the principle of locality. So, when there is a context switch this locality is lost. In the sense that we are no longer executing the same instructions has we had done before the context switch. So, as a result of this, the cache memories would need to be reloaded again and this could incur several cache misses and as a result add up to over heads.

Another aspect is that every time an interrupt occurs, the processor pipeline would need to be flushed. So, this also adds up to the over head. The context switching could incurs significant over heads and degrade performance quite significantly. As a result designer should very carefully decide upon how context switching is done and rather when it is done in order to achieve best performance of their system.

With this we will end this particular video on context switching in operating systems. So, we have seen various aspects of context switching essentially we have seen some details about how context switching occurs in operating systems and we also seen over heads that are incurred due to context switching.

Thank you.