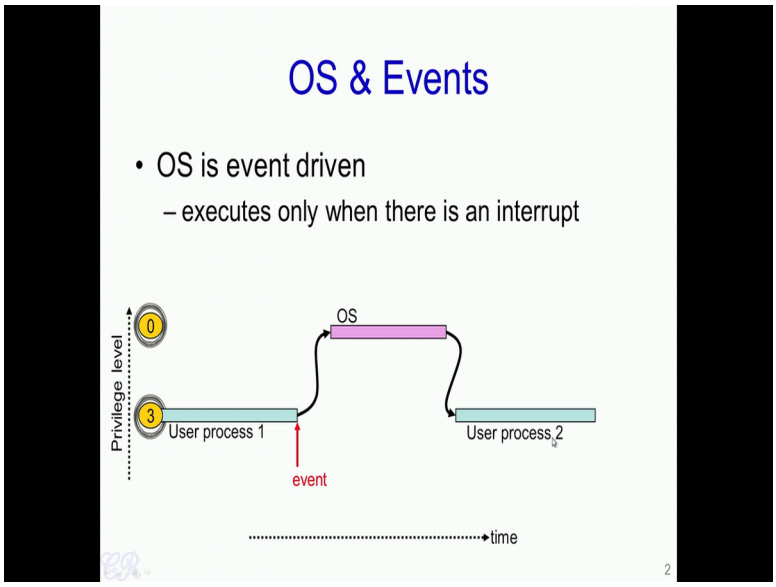
**Introduction to Operating Systems**
**Prof. Chester Rebeiro**
**Department of Computer Science and Engineering**
**Indian Institute of Technology, Madras**

**Week – 04**
**Lecture – 14**
**Interrupts**

Hello and welcome to this video. In today's video we will look at interrupts, which forms a crucial part in all modern day operating systems. So, unlike normal software or normal programs that we write, operating systems are event based that is whenever an event occurs only then the operating system executes. To see what this means, let us look at the slides.

(Refer Slide Time: 00:43)

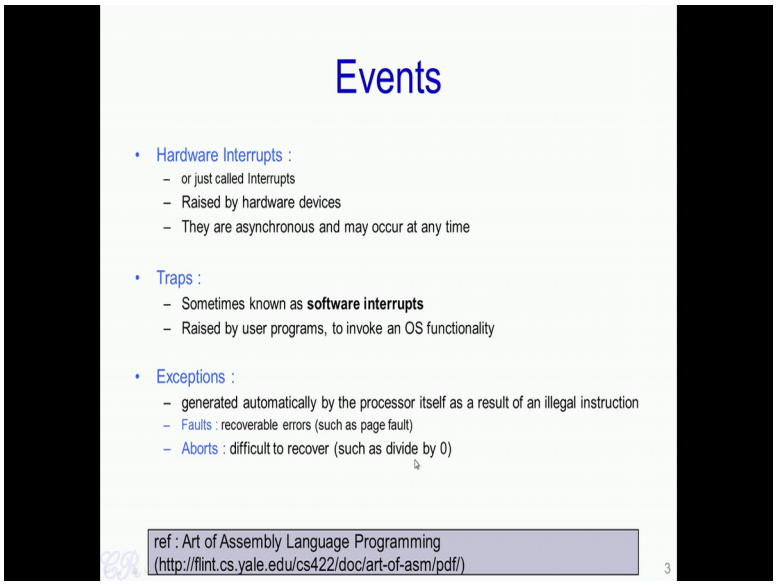



So suppose you have a user process that is running, so as we have seen before user process runs in user space and in the Intel nomenclature this is in ring 3 (mentioned in above image). Now this user process continues to execute on the processor until an event occurs. So, when this event occurs it would trigger the operating system to execute. So, this triggering of the operating system will also result in a change in the privilege level. So the system would no longer be executing in the user space, but rather it will be in privilege level 0 or that is executing in the Kernel space.

So the operating system would essentially execute and service this particular event (refer

above image) and at the end of that execution, the control is fed back to user space and the process will continue to execute. So the User space process which would execute after the OS completes could be the same process that is User process 1 or some other User Process, in this example it is User process 2 (mentioned in above image).

(Refer Slide Time: 02:01)




Let us look at how events are classified. So, various literature categorizes events in different ways, but what we will do is we will follow the categorization of events based on this particular book called the Art of Assembly Language Programming which can be downloaded from this website (mentioned in above image). So, in this book events are classified into 3 different types; these are Hardware Interrupts, Traps and Exceptions.

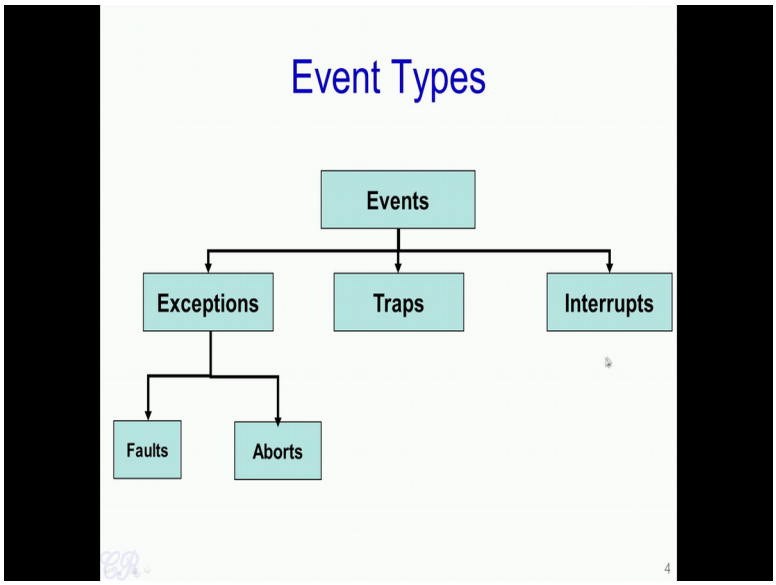
So Hardware Interrupts or sometimes just called as Interrupts are raised by external hardware devices. For example, the network card when it receives a packet could possibly raise an interrupt or other devices such as the keyboard, mouse or a USB device when plugged in could raise in hardware interrupt. So, these hardware interrupts are asynchronous and can occur at any time.

Besides hardware interrupts there are Traps and Exceptions. Traps are sometimes known as software interrupts, they are raised by user programs in order to invoke some operating system functionality. For instance, if a user program wants to print something on the monitor it would invoke a trap, which essentially would be a system call to the operating system and the OS will then take care of writing the particular text or writing

the particular data on to the screen.

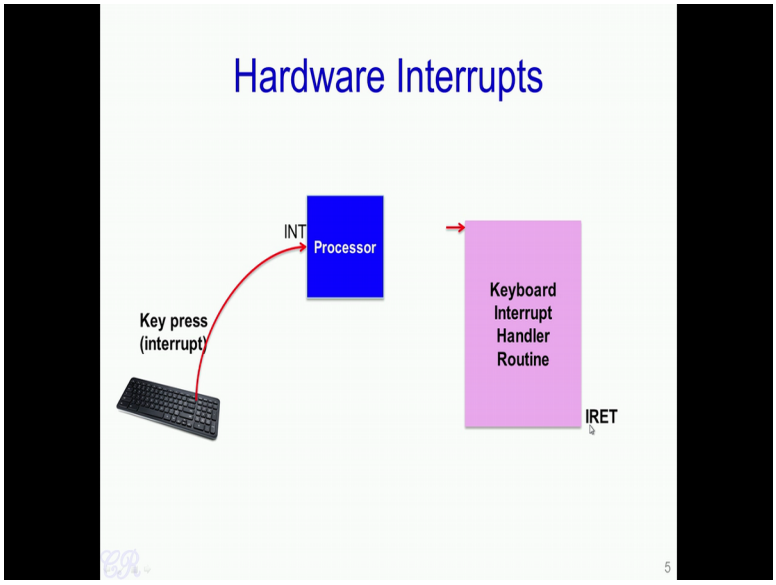
The third type of event is known as Exceptions. These events are generated automatically by the processor itself as a result of an illegal instruction. So, there are 2 types of exceptions these are Faults and Aborts. A very common example of a fault is a page fault. So, faults are essentially exceptions from which the processor could recover. For instance, when there is a page fault that occurs when a process is executing, it would result in the OS or in the operating system executing and loading the required page from the swap space into the RAM. On the other hand, an exception which is of the form abort would be very difficult to recover, such as a divide by 0 exception.

(Refer Slide Time: 04:38)




So when a divide by 0 exception occurs in a program, typically the program would be terminated. Essentially the operating system has no way to recover from such a divide by 0 exception. This particular slide (mentioned in above image) shows the various classification of events into Exceptions, Traps and Interrupts. Exceptions are further classified into Faults and Aborts. So, we will now take a specific case of interrupts that is of Hardware Interrupts.

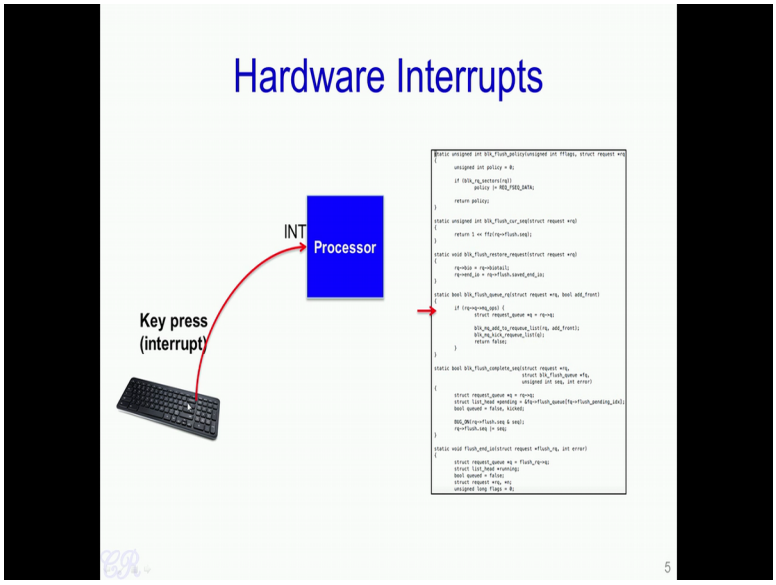
(Refer Slide Time: 04:58)




So, let us look at Hardware Interrupts. In general, processors today have a dedicated pin on the IC known as the Interrupt pin. So, this pin is often shortened or just called by the INT pin or in some processors as the INTR pin. So, devices such as the keyboard will be connected to the processor through the INT pin. When a particular key is pressed on the keyboard it would result in an interrupt being generated to the processor.

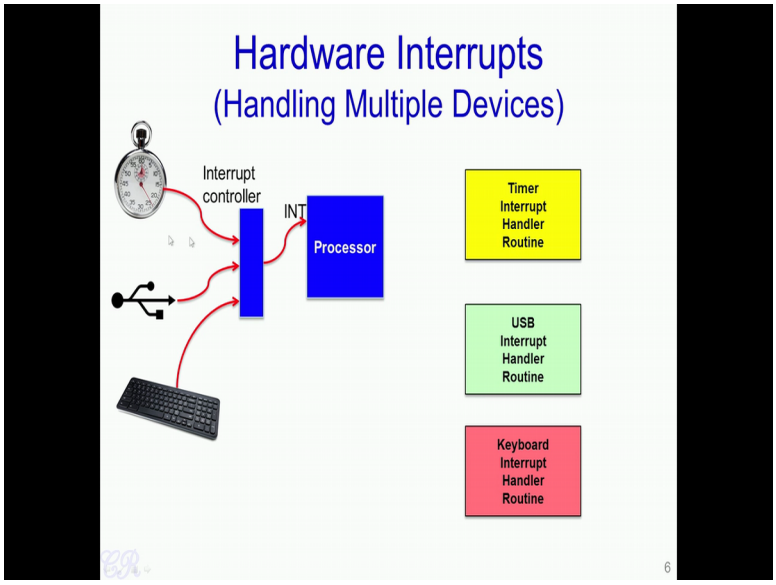
Now, let us see how this particular interrupt takes place (mentioned in above image) and what happens in the processor. So, the processor typically would be executing a program and would be executing some instructions. Now, when a key is pressed, an interrupt is generated to the processor and that would result in a switch in the processor to what is known as the Interrupt Handler Routine.

(Refer Slide Time: 05:58)




So, in this particular case since it is the keyboard which has resulted in the interrupt, the keyboard interrupt handler routine would be invoked. The processor would then begin to execute this keyboard handler routine until an instruction such as the IRET is obtained. When the IRET instruction gets executed, the context is switched back to the program which was originally being run. So, in this way we see that interrupts could occur any time during the programs execution, it would result in a new context being executed and at the end of that execution the processor goes back to the original context.

(Refer Slide Time: 06:41)

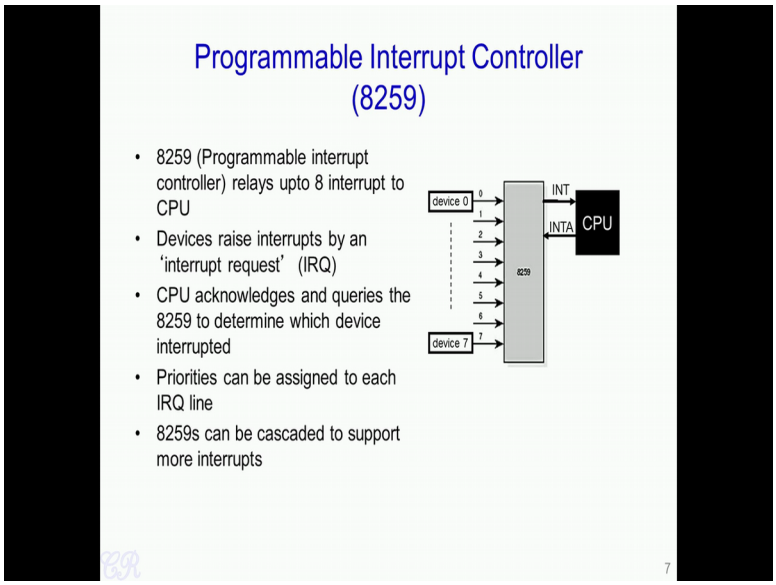



So typically systems do not have just one device connected to the processor, there could be several devices. For instance systems could have timers, USB drives, keyboards,

mouse, network cards and so on. However, as we have seen previously the processor just has a single interrupt pin. So how is it possible then that several devices share the single interrupt pin? In order to achieve this, a special hardware is used in systems. So, this is known as the Interrupt Controller (mentioned in above image). The job of the Interrupt Controller is to ensure that the single pin of the interrupt is shared between multiple devices. So, the interrupt controller would receive interrupts from each of these devices and then channelize that interrupt to the INT pin of the processor.

The processor would then communicate with the interrupt controller to determine which of these devices had actually generated the interrupt. As a result, the processor would then execute the corresponding interrupt handler routine. For instance, if the timer had resulted in the interrupt then the timer interrupt handler routine would be invoked. On the other hand, if a USB device had resulted in the interrupt, the USB interrupt handler routine would be invoked and so on (refer above image). Thus, the interrupt handler routine invoke is going to be very specific to the device that cause the interrupt to occur.

(Refer Slide Time: 08:15)

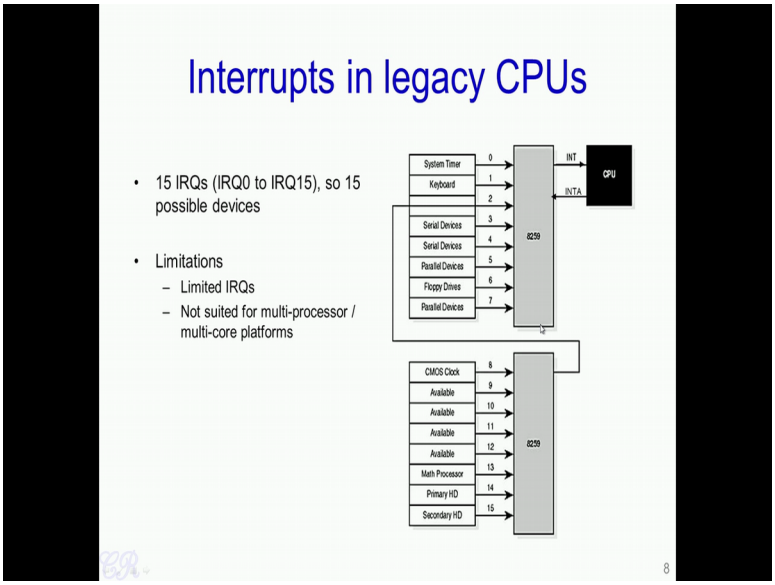



So, one commonly used interrupt controller is known as a Programmable Interrupt Controller. So, it is numbered as 8259 and pictorially this is how it gets connected. So, the 8259 has two sides, this is the input side and the output side. The output is connected to the INT pin of the CPU, there is also an INTA pin which is an interrupt acknowledge pin (mentioned in above image).

On the other side we have 8 IRQ lines. IRQ stands here for Interrupt Requests. So, on the input side there are up to 8 devices that can be connected to the 8259. These devices are labeled device 0 to device 7, all these devices could independently request an interrupt from the CPU. The 8259 would then channelize that interrupt through the INT pin of the CPU. The CPU would acknowledge the interrupt through the INTA pin and also determine which of these 8 devices had requested the interrupt.

Now, what would happen if two devices request the interrupt at exactly the same time? In such a case, 8259 would use some priority encoding algorithm to determine which of these devices should be given the privilege to request for the interrupt first. Another feature of the 8259 is that it can be cascaded to support more than 8 devices, thus more than 8 devices could cause interrupts to the CPU.

(Refer Slide Time: 09:52)

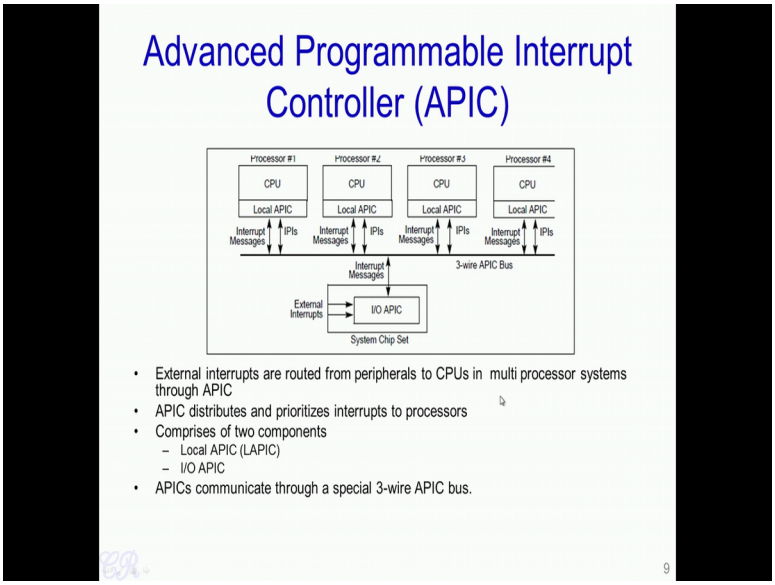



So in Legacy computers typically there are two 8259 controllers present - one is configured as the master, while the other is configured as the slave (first 8259 in above image is master and below that is slave 8259). The slave 8259 controller is connected to one of the input channels of the master 8259. So, if any of these devices connected to the slave 8259, request an interrupt, this interrupt is channelized to the master 8259 and the master 8259 would then channelize this interrupt to the CPU.

So, one limitation of this legacy configuration of the 8259s is the limited IRQs. So, each device could as we sink and support only 8 devices and therefore, if you have large

number of devices in your system then you would require several 8259 controllers to be present. Another major limitation of this configuration is that, the support from multi-processor and multi-core platforms is difficult. Essentially as we seen over here (refer above image), there is only one CPU that is connected to the master 8259 programmable interrupt controller. This particular INT pin cannot be sent to an other CPU, which may result in some problems.

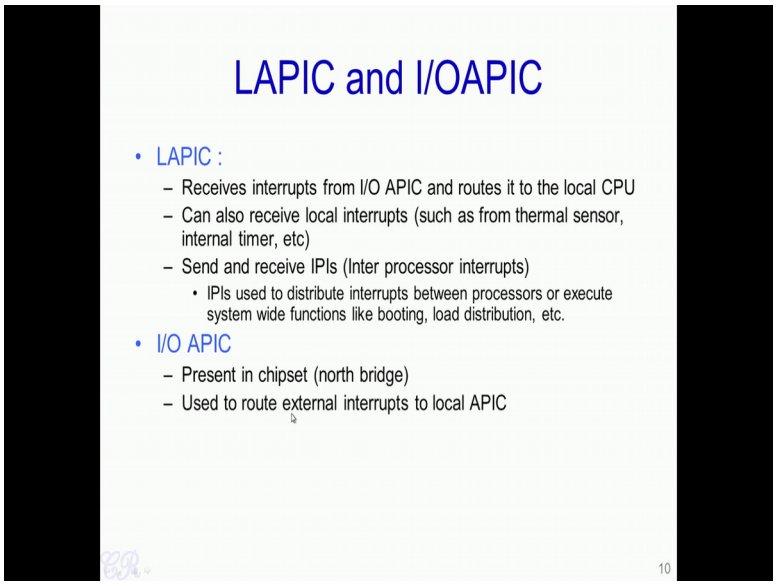
(Refer Slide Time: 11:14)




In current systems, the 8259 programmable interrupt controller is replaced by something known as APIC or Advanced Programmable Interrupt Controller. The configuration for the APICs in modern systems is as shown over here (mentioned in above image). So, each CPU in a multi-core or multiprocessors system would have a local APIC as shown here. So, Processor 1 has a local APIC, Processor 2 has its own APIC, Processor 3 has its own local APIC and so on.

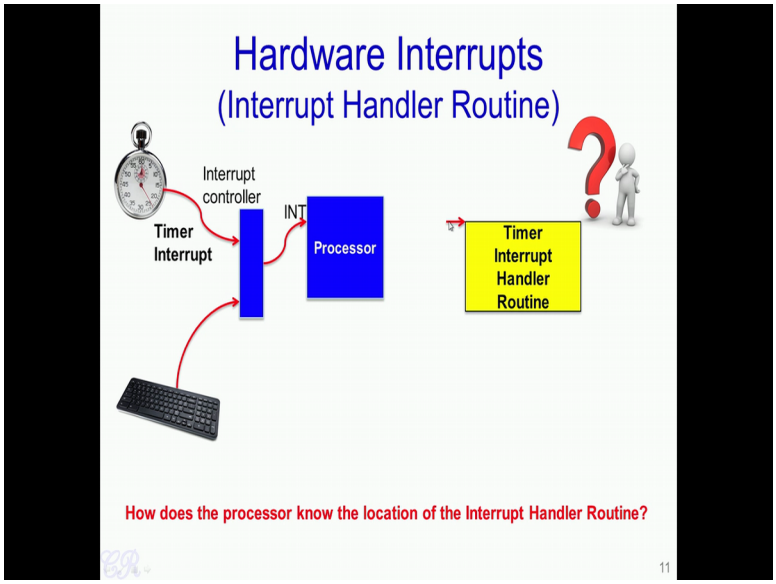
In addition to this, there is something known as an I/O APIC which is present in the System Chip Set (mentioned in above image). Now, external devices such as keyboards, mouse, network cards, and so on, would request interrupts through the I/O APIC which is then channelized to one of the local APICs in each CPU. Thus interrupts can be distributed and prioritized between CPUs. Also the local APICs as well as the I/O APICs communicate through interrupt messages and IPIs that is Inter Processor Interrupts.

(Refer Slide Time: 12:24)




So the local APIC receives interrupts from the I/O APIC and routes it to the corresponding local CPU. The local APIC can also receive some local interrupts such as interrupts from the thermal sensor which is present on the CPU, an internal timer and so on. So, the local APIC could also send and receive IPIs which is Inter Processor Interrupts, which allows interrupts between processors that it allows processors to do system wide functions like booting, load distribution, and so on. The I/O APIC is present in the System Chip Set which is sometimes known as North Bridge. This particular APIC is used to route external interrupts into the local APIC.

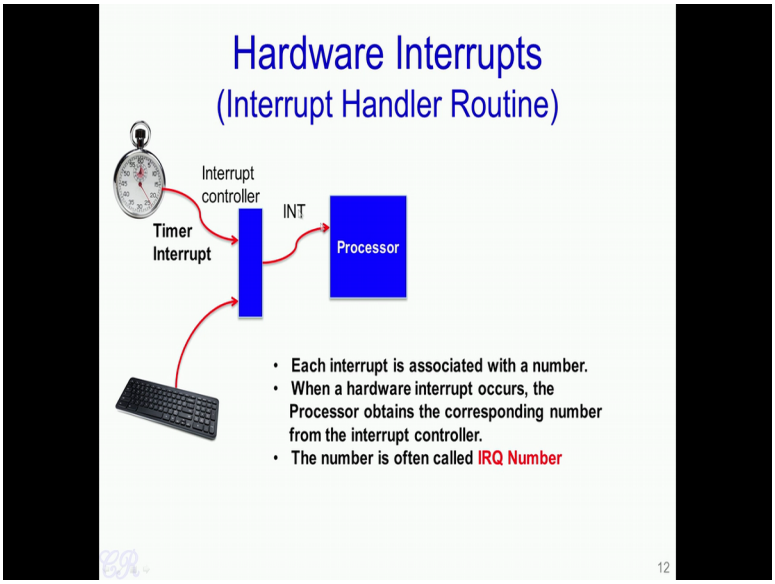
(Refer Slide Time: 13:12)



So, we have seen so far that we have multiple devices which are connected to an interrupt controller which could be either an APIC or in legacy systems based on the 8259 controller. When any of these devices request an interrupt, the interrupt is channelized to the processor and it would result in the corresponding interrupt handler routine to be invoked.

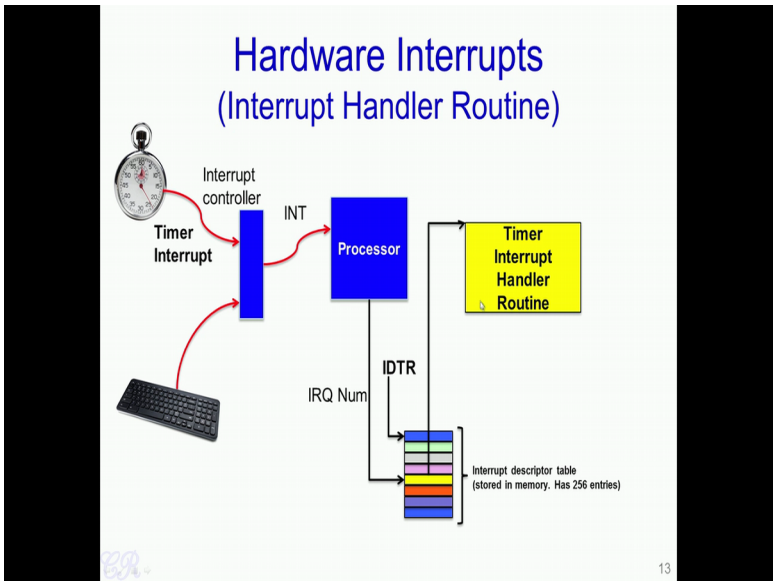
How does the processor know the location of the Interrupt Handler Routine? So, the interrupt handler routine is just like any other software code is also executed from the RAM. So, the question that we are posing here is how does the processor know the starting location or the address of the interrupt handler routine.

(Refer Slide Time: 14:13)



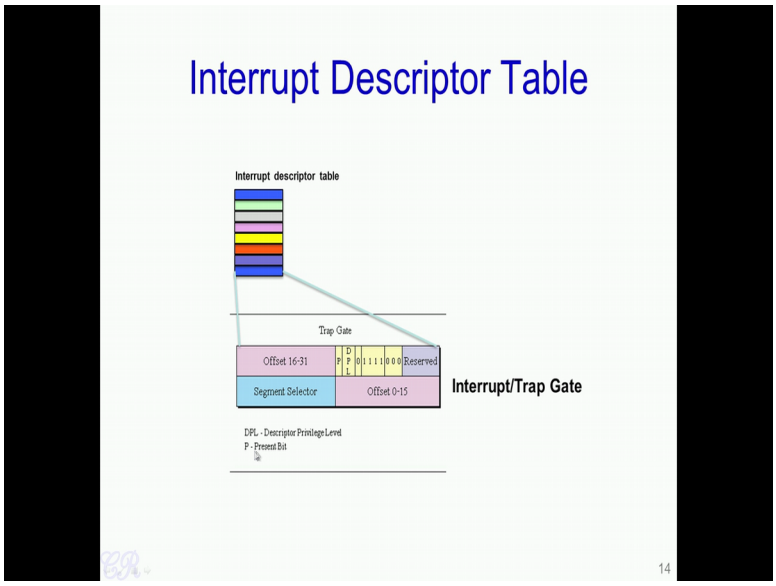

So in order to achieve this, what happens is the following (refer above image); so each of the devices that are connected to the interrupt controller is also has a dedicated number known as the IRQ number. So, when a device, request an interrupt and the INT pin in the processor gets asserted, the processor would then obtain the IRQ number from the interrupt controller. In this way the processor now determines which of these devices has requested for the interrupt.

(Refer Slide Time: 14:43)




We will now see how the processor uses the IRQ number (mentioned in above image). In the memory of the system, a table known as the interrupt descriptor table or IDT is stored. So this IDT table is pointed to by a register stored in the processor known as the IDTR. This is interrupt descriptor table register. So, each entry in this descriptor table contains information of where in memory the corresponding interrupt handler routine is present. So, the processor would use the IRQ number to look into the interrupt descriptor table, from this descriptor the corresponding location of the interrupt handler routine is obtained and there after the processor can then execute instructions from this handler routine.

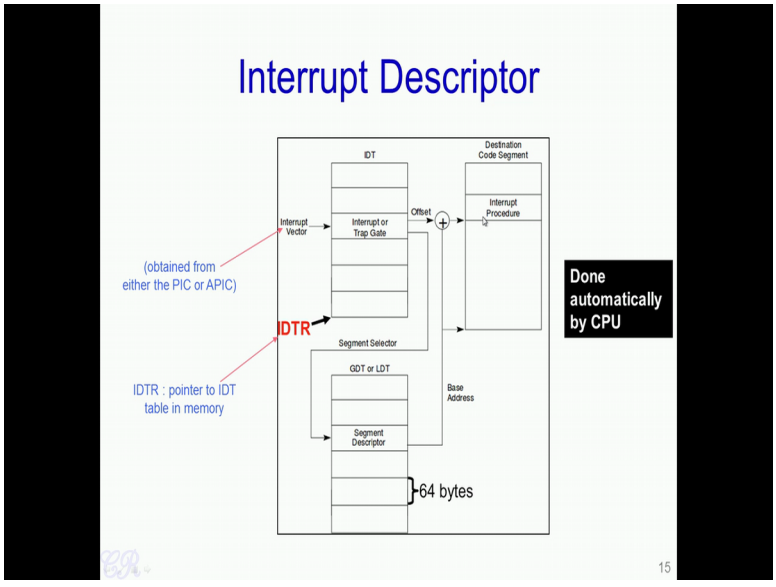
(Refer Slide Time: 15:36)




So, what is the contents of the interrupt descriptor table? In the x86 systems, since it also has segmentation. So therefore, the each interrupt descriptor would contain the segment selector as well as the offset. The segment selector as we have seen is of 16 bits and while the offset is of 32 bits. So, bits 0 to 15 are here and the bits 16 to 31 are present over here (mentioned in above image).

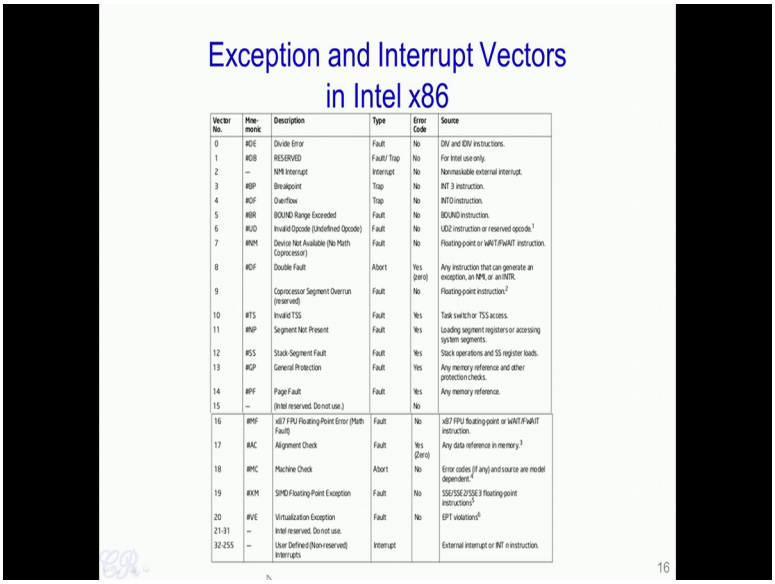
There are other aspects in the interrupt descriptor such as the Present bit and the Descriptor Privilege Level.

(Refer Slide Time: 16:11)



So let us see how this Interrupts Descriptors are used in reality. So, the processor would obtain the IRQ number or the interrupt vector from the programmable interrupt controller or the APIC, and as we have seen this interrupt vector is used to index into the IDT table, and the corresponding interrupt or trap gate contains the segment selector as well as the offset. The segment selector is then used to index into the GD or the LDT table from which, the base address of the code segment is obtained. The offset part is then taken and added, to the base address to obtain the final address where the interrupt procedure is present.

(Refer Slide Time: 17:01)




| Vector No. | Mnemonic | Description | Type | Error Code | Source |
|---|---|---|---|---|---|
| 0 | #DE | Divide Error | Fault | No | DIV and IDIV instructions. |
| 1 | #DB | RESERVED | Fault/ Trap | No | For Intel use only. |
| 2 | – | NMI Interrupt | Interrupt | No | Nonmaskable external interrupt. |
| 3 | #BP | Breakpoint | Trap | No | INT 3 instruction. |
| 4 | #OF | Overflow | Trap | No | INTO instruction. |
| 5 | #BR | BOUND Range Exceeded | Fault | No | BOUND instruction. |
| 6 | #UD | Invalid Opcode (Undefined Opcode) | Fault | No | UD2 instruction or reserved opcode.[1] |
| 7 | #NM | Device Not Available (No Math Coprocessor) | Fault | No | Floating-point or WAIT/FWAIT instruction. |
| 8 | #DF | Double Fault | Abort | Yes (zero) | Any instruction that can generate an exception, an NMI, or an INTR. |
| 9 | | Coprocessor Segment Overrun (reserved) | Fault | No | Floating-point instruction.[2] |
| 10 | #TS | Invalid TSS | Fault | Yes | Task switch or TSS access. |
| 11 | #NP | Segment Not Present | Fault | Yes | Loading segment registers or accessing system segments. |
| 12 | #SS | Stack-Segment Fault | Fault | Yes | Stack operations and SS register loads. |
| 13 | #GP | General Protection | Fault | Yes | Any memory reference and other protection checks. |
| 14 | #PF | Page Fault | Fault | Yes | Any memory reference. |
| 15 | – | (Intel reserved. Do not use.) | | No | |
| 16 | #MF | x87 FPU Floating-Point Error (Math Fault) | Fault | No | x87 FPU floating-point or WAIT/FWAIT instruction. |
| 17 | #AC | Alignment Check | Fault | Yes (Zero) | Any data reference in memory.[3] |
| 18 | #MC | Machine Check | Abort | No | Error codes (if any) and source are model dependent.[4] |
| 19 | #XM | SIMD Floating-Point Exception | Fault | No | SSE/SSE2/SSE3 floating-point instructions[5] |
| 20 | #VE | Virtualization Exception | Fault | No | EPT violations[6] |
| 21-31 | – | Intel reserved. Do not use. | | | |
| 32-255 | – | User Defined (Non-reserved) Interrupts | Interrupt | | External interrupt or INT n instruction. |

So the Intel x86 systems supports 56 different types of events. Some of these events are used internally for faults and aborts, while others can be configured for hardware interrupts as well as for software interrupts.

So, this particular table (refer above image) gives some of these interrupts and exceptions that are supported by Intel x86 systems. Especially we would like to see that interrupt with vector number 0 to 31 are internal to the processor, of these the vector number 0 to 20 are used for various faults or aborts. The interrupt vector numbers from 32 to 255 can be user defined. So, some of these user defined interrupts are configured as hardware interrupts, while others are used as software interrupts.

So in the next video, we will look at how interrupts are handled in the CPU as well as in the operating system.

Thank you.

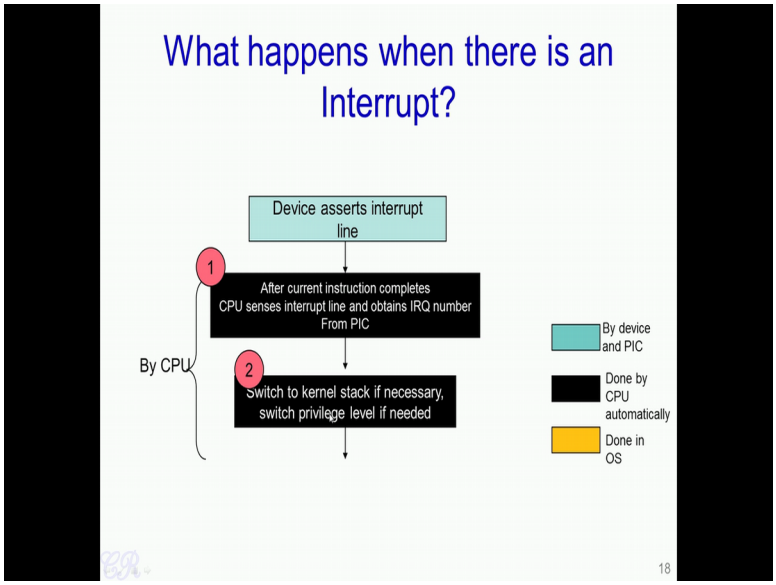
**Introduction To Operating Systems**
**Prof. Chester Rebeiro**
**Department of Computer Science and Engineering**
**Indian Institute of Technology, Madras**

**Week – 4**
**Lecture – 15**
**Interrupt Handling**

Hello. In the previous video we had seen how interrupts can be requested from any external device and it would result in a interrupt service routine being executed.

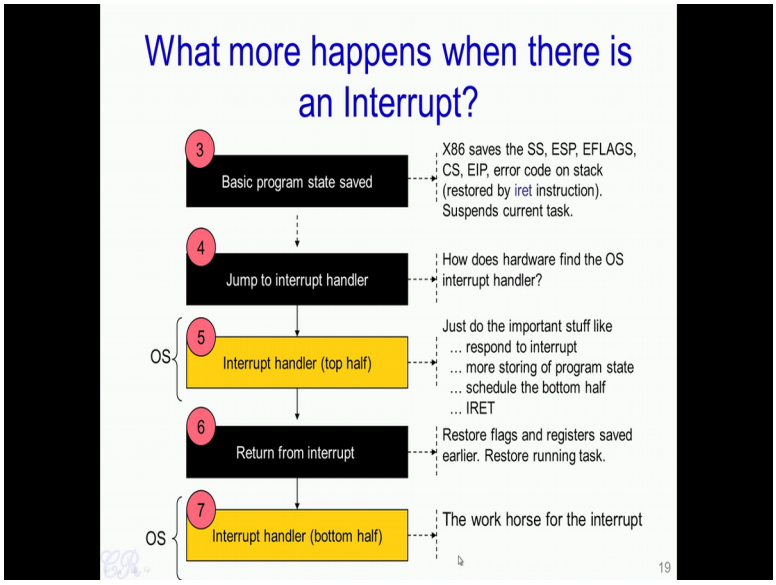
So, in this video we will look at more into detail how interrupts are handled. So Interrupt Handling is a pretty involved process which involves both the CPU as well as the operating system. So, we will go step by step and see what are the various stages in interrupt handling.

(Refer Slide Time: 00:50)




So let us start with the first. Let us say that the device asserts an interrupt line, and as we know this would result in the interrupt controller channelizing that request into the INT pin of the processor. So, what happens next? So In the processor called CPU over here (mentioned in above image), (1st point) the CPU would sense that the interrupt line or the INT line is asserted, and it would obtain the IRQ number from the PIC that is the Programmable Interrupt Controller. Then, (2nd point) the processor would switch to the kernel stack and also switch the privilege level if required.

(Refer Slide Time: 01:31)

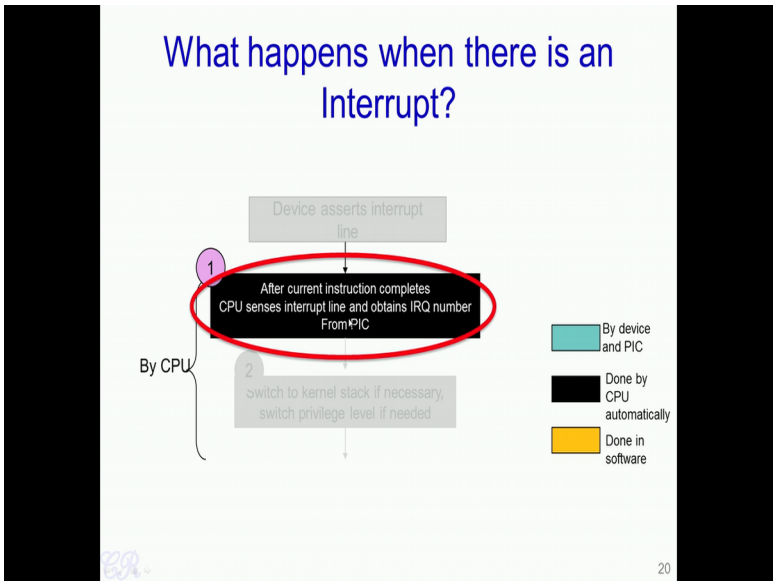



The 3rd step is the current execution of the program is stopped and the program state is saved. So, in x86 the program state comprises of several registers such as the SS register that is the Stack Segment, the stack pointer, flags register, the code segment and the instruction pointer.

So all these are saved on to the stack and then the processor would jump to the interrupt handler (4th point). So in the interrupt handler, we have a top half of the interrupt handler (5th point) important things like respond to the interrupt, more storage of program state, scheduling of something known as the bottom half of the interrupt handler is done, and then it is followed by the IRET which is the Return from the interrupt. The CPU then executes the return from interrupt (6th point) and after sometime there is the bottom half of the interrupt handler that runs (7th point). So, this bottom half of the interrupt handler is essentially known as the work horse of the interrupt and does most of the difficult or time consuming jobs.

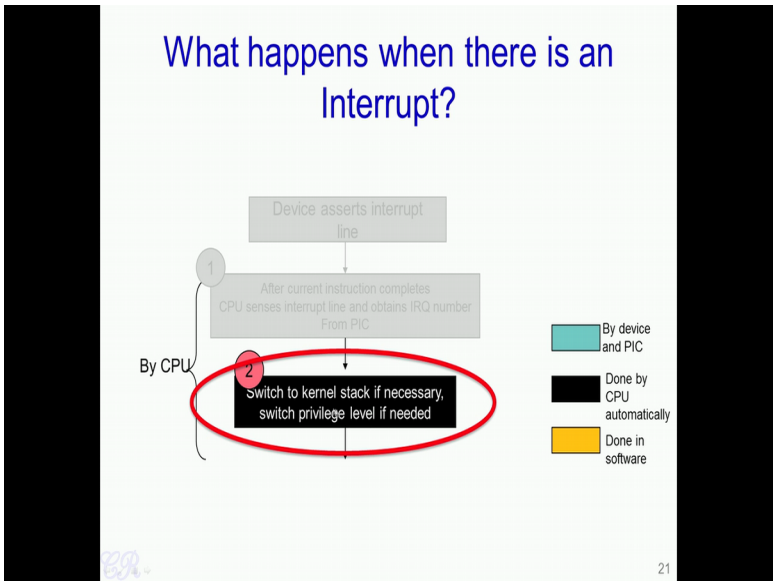
So, one thing to notice is that each of these stages could be done either by the CPU automatically that is the processor hardware itself does this automatically, so these are the blacked boxes (Refer Slide Time: 00:50), while the yellow boxes is done in software, essentially by the operating system. So, you see that some of these steps are done automatically by the CPU, while others such as the interrupt handler is done in software by the operating system. So, what we will see next is each of these stages in more detail.

(Refer Slide Time: 03:18)



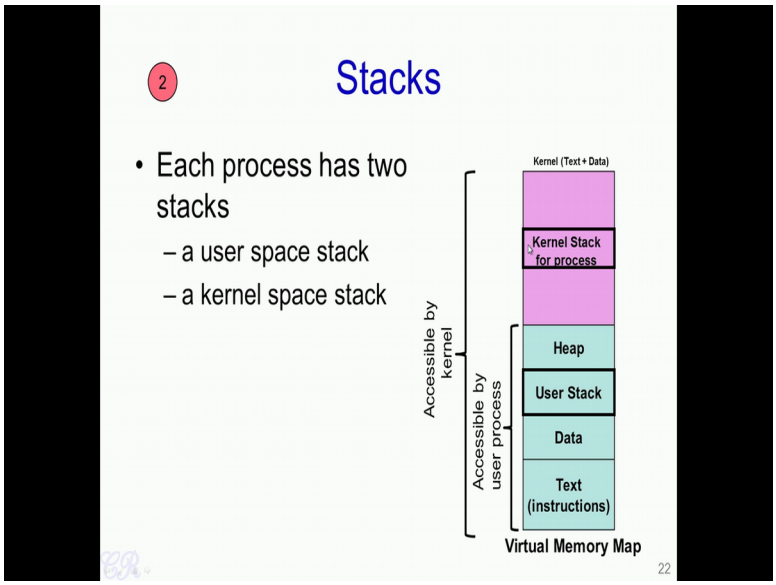

So, after the current instruction completes executing, the CPU senses the INT line and then when it determines that a particular interrupt has been requested, it obtains the IRQ number of that interrupt from the PIC (point 1st in above image).

(Refer Slide Time: 03:37)




Then it would switch to the kernel stack if necessary, and also change the privilege level to ring 0 that is it would allow the kernel code to start executing (point 2nd in above image).

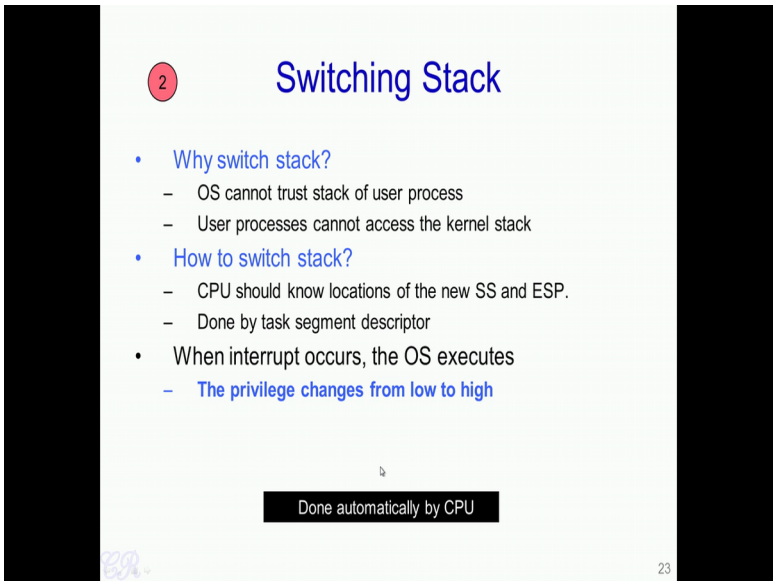
(Refer Slide Time: 03:46)




So, we will look at this, so we look at the stacks first. So, as we have seen before each process has two stacks (refer above image). So, one stack known as the User Stack (highlighted above in blue color) is visible in the user space and it is typically what is used to store various auto variables and for function calls for the instructions in the user program.

On the other hand, there is a hidden kernel stack corresponding to each process (highlighted above in purple color). So, when the processor detects the interrupt, the context changes from user stack to the kernel stack. Now, hence forth the kernel stack is going to be used to store auto variables as well as for function calls of the code that executes in the kernel.

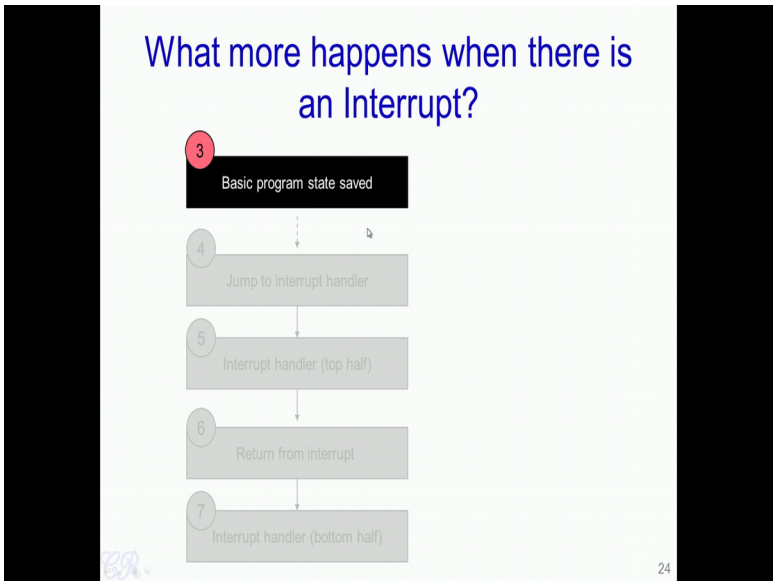
(Refer Slide Time: 04:33)




So, why do we switch stacks? Essentially stacks are switched because the OS cannot trust the user process stacks. The user process stacks may be corrupted and as a result we do not want that the kernel also gets corrupted due to this reason. Another reason is that user processes cannot access the kernel stack, so for instance if the user process is a malicious virus for instance, it has no access to the kernel stack and therefore, cannot modify or change anything in the kernel.

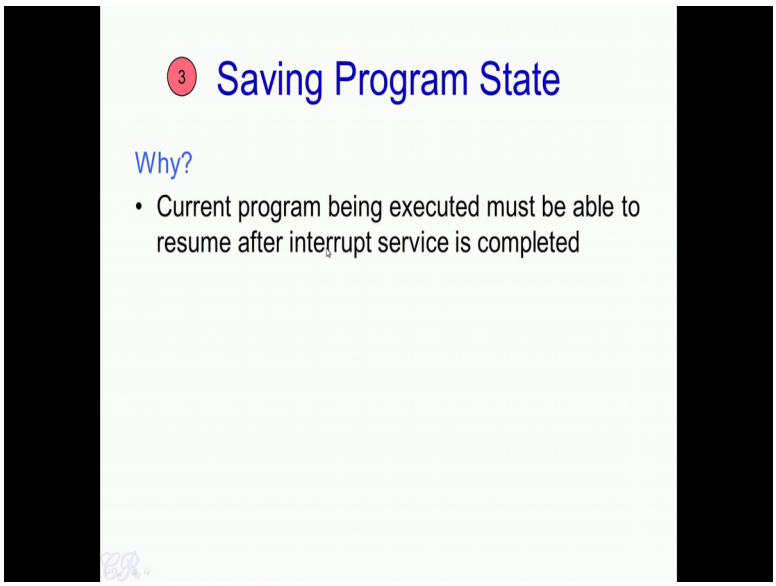
Second thing is, how is the stack actually switched from the user stack to kernel stack? So, this is done by something known as the task segment descriptor and essentially what it does is that it changes the stack segment and the stack pointer of the processor. So the information of the new stack segment and the stack pointer is present in the stack segment register. So, another thing that occurs during the process of switching stack is that the privilege level changes from low to high that is from ring 3 which where the user processes run to ring 0 where the kernel runs. So, all these things are done automatically by the CPU.

(Refer Slide Time: 05:54)



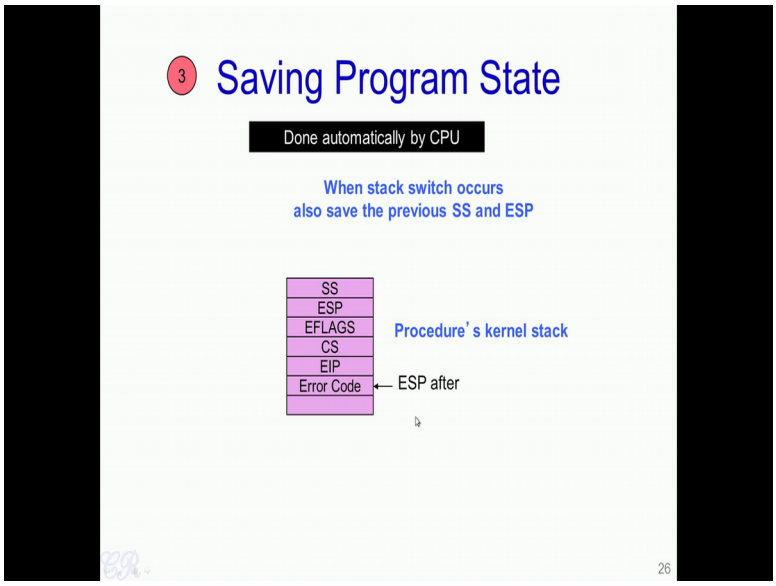

So, after changing stack and raising privilege level, the basic program state is saved (point 3rd in above image).

(Refer Slide Time: 06:04)



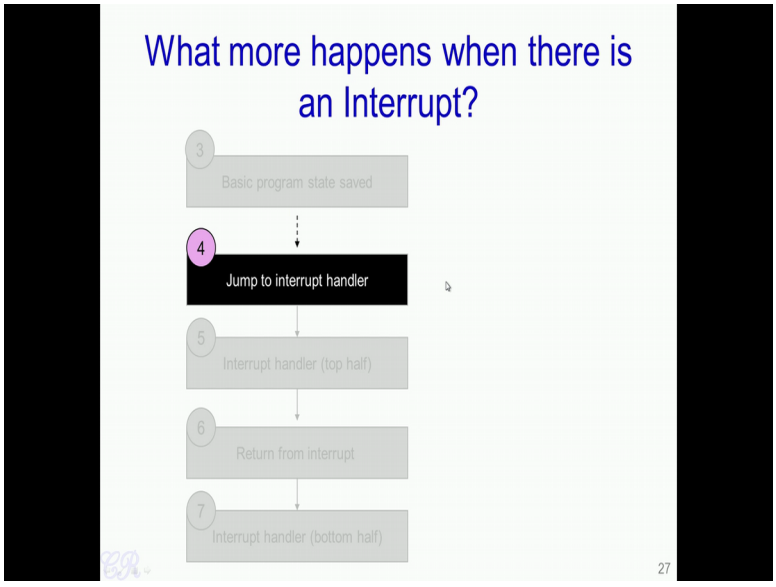

So, what this means is that, suppose we have a program which is being executed in user space and an interrupt occurs, then the state of that process is saved. Why do we require to save the state of that process? It is required so that the process could resume after the interrupt servicing is completed.

(Refer Slide Time: 06:27)



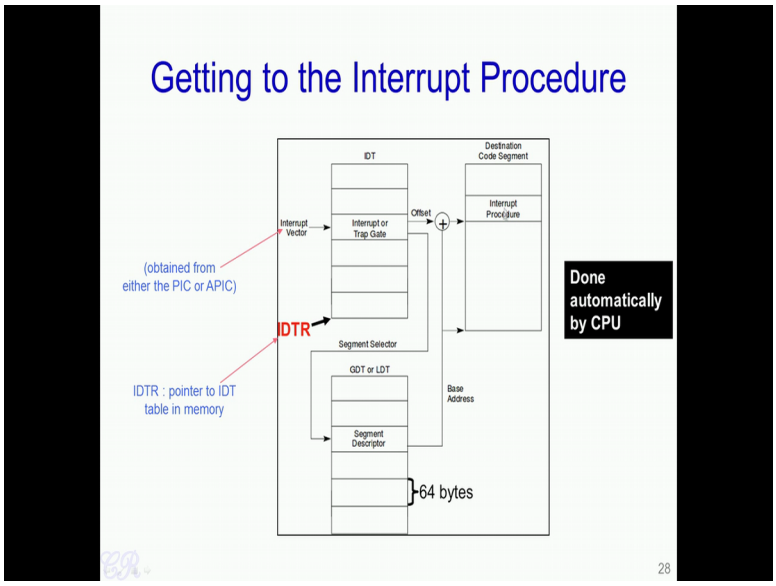

How is the program state saved? In order to save the program state, we will use the kernel stack which we have just pointed to when the interrupt occurred. So in the kernel stack we would save the various register such as SS, ESP, EFLAGS, CS, EIP and an Error code if required.

(Refer Slide Time: 06:50)



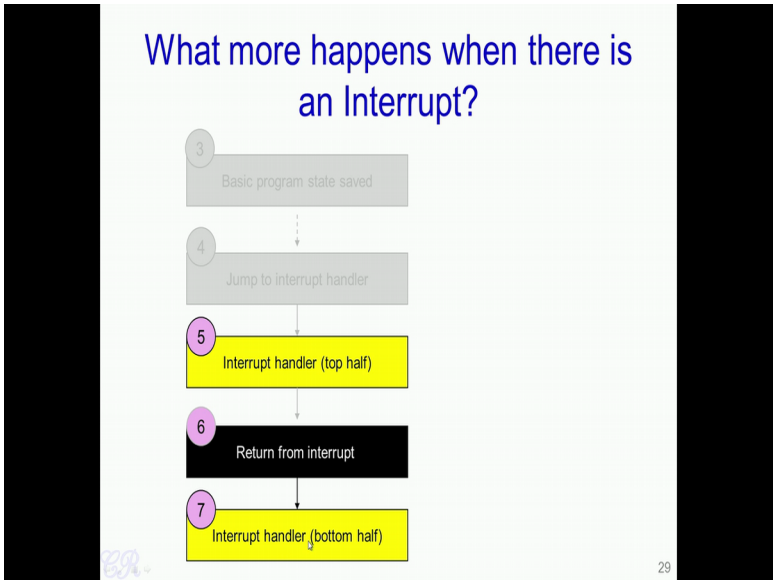

Next, we will see how the processor jumps to the interrupt handler (point 4$^{th}$ in above image).

(Refer Slide Time: 06:54)




So, we have seen this before that the processor would use the IRQ number, or the interrupt vector to index into the IDT table from where we would obtain the segment selector as well as an offset (refer above image). The segment selector is used to obtain the base address of the code segment, while the offset is added to this base address to obtain the location of the interrupt procedure. So the processor could then begin to execute this interrupt procedure.

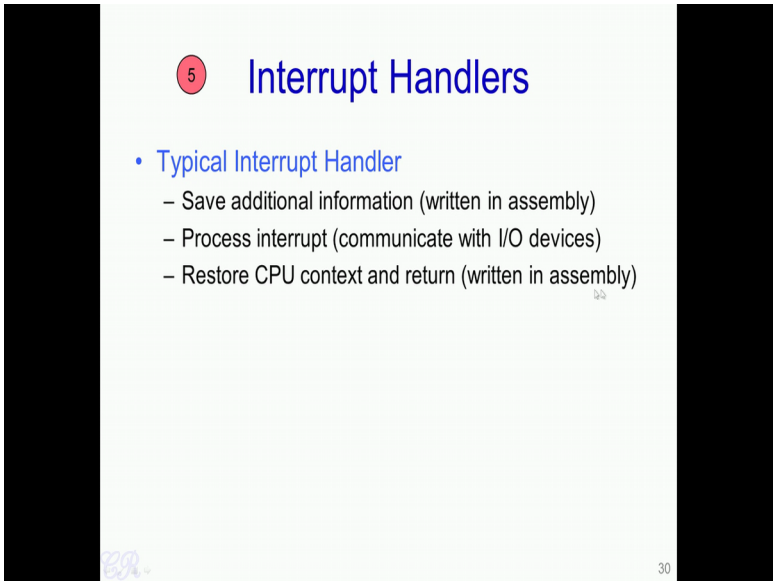
(Refer Slide Time: 07:25)




The next step is to actually execute the interrupt handler. Return from the interrupt and

also execute the bottom half of the interrupt handler (point 5th, 6th and 7th in above image).

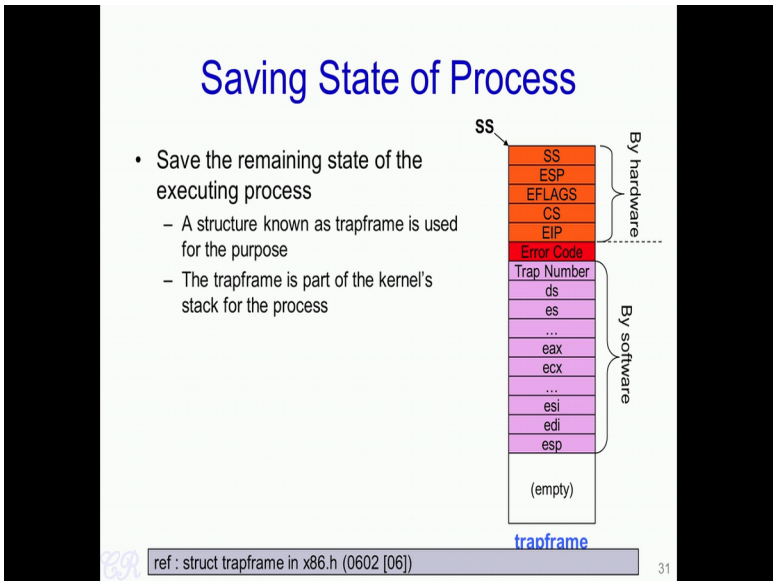
(Refer Slide Time: 07:34)




So let us see what a typical interrupt handler does (refer above image). So, a typical interrupt handler would have 3 parts - one it would save some additional information about the process being invoked. Then, it would process the interrupt which is going to be very specific to the type of interrupt. For instance, if it is a keyboard interrupt then the code executed over here would be very specific to the keyboard. On the other hand, if it is a timer interrupt then the interrupt would be very specific to timers.

After the processing of the interrupt is done, then the CPU restores the original context and returns back to the user process. So, we have the first and the third part written in assembly language that is typically written in assembly language (point 1st and 3rd in above image) while the processing of the interrupt is typically written in higher level language like C (point 2nd in above image).

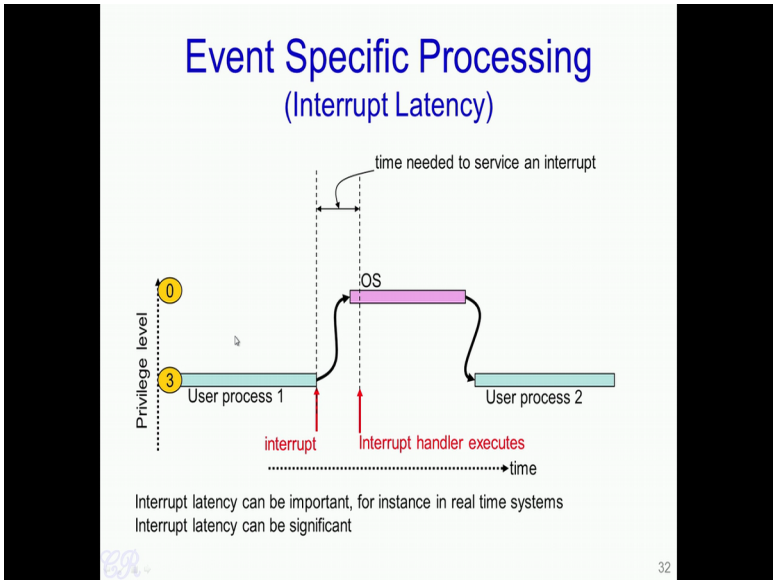
(Refer Slide Time: 08:27)




So let us see about the additional information that is stored. So, we have seen in an earlier slide (Refer Slide Time: 06:27) that there is a saving of the program state on the kernel stack and note that this is done automatically by the CPU, and second that a few registers such as the SS, ESP, EFLAGS, CS, EIP and an Error code may be saved. So, when the interrupt handler begins to execute, additional information is also stored on to the kernel stack. So, we have a part of this (mentioned in above image in orange color) which is stored by the hardware or that is by the CPU automatically and remaining part which is done in software that is in by the operating system (mentioned in purple color).

So this additional part which is done by the operating system will have more registers which are saved. For example, the segment register such as the DS, ES, and so on, and also the general purpose registers such as the EAX, ECX, and so on. Similarly, the other registers like the ESI, EDI and ESP are stored. Or this pattern of registers which is stored on to the kernel stack is known as the Trapframe, and plays a crucial role during the return from the interrupt, in order that the user process which had been executing before could continue to execute from where it has stopped.

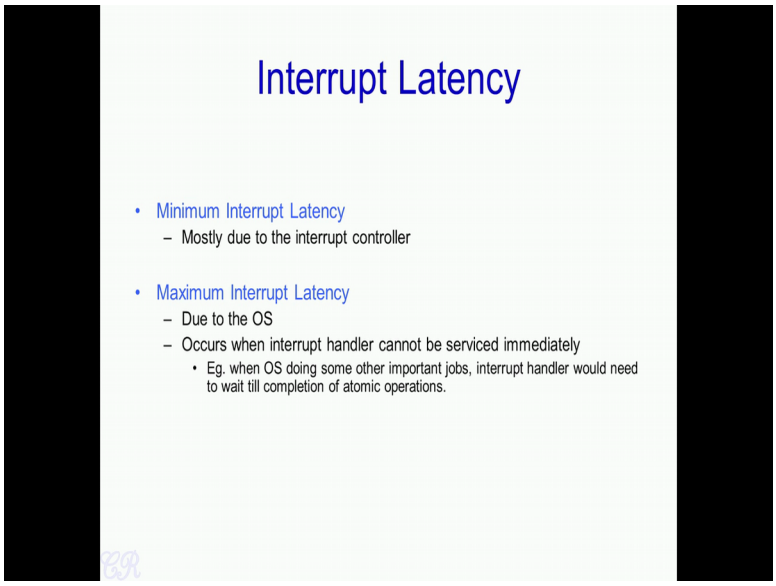
(Refer Slide Time: 09:48)




One important aspect when writing an interrupt handler is the Interrupt Latency. So, what is this Interrupt Latency? So let us say that we have a processor which is executing a user process that is User process 1 (mentioned in above image) and after some time an interrupt occurs. So, when this interrupt occurs there are a series of tasks that happen between the CPU as well as the operating system. First, the interrupt has to be detected by the PIC and then forwarded to the CPU, then the CPU detects the interrupt and then it has to do various things like change context from user space to kernel space, and then save various registers of the user space program and only then, it would be able to run the interrupt handler.

So, this time difference between, when the interrupt actually has been triggered to when the interrupt handler executes is known as the Interrupt Latency (difference between two red arrows in above image). So, this Interrupt Latency can be significant as well as important especially in systems such as real time systems. For instance, if you look at systems such as an operating system used in car, you do not want a large latency, for instance when an interrupt occurs in order to release the air bag which is present in some of the modern cars. So, for instance when an accident occurs you would require that air bags are immediately and extremely quickly opened up. Therefore, in such a case you would require the interrupt latency to be as small as possible. What affects this Interrupt Latency?

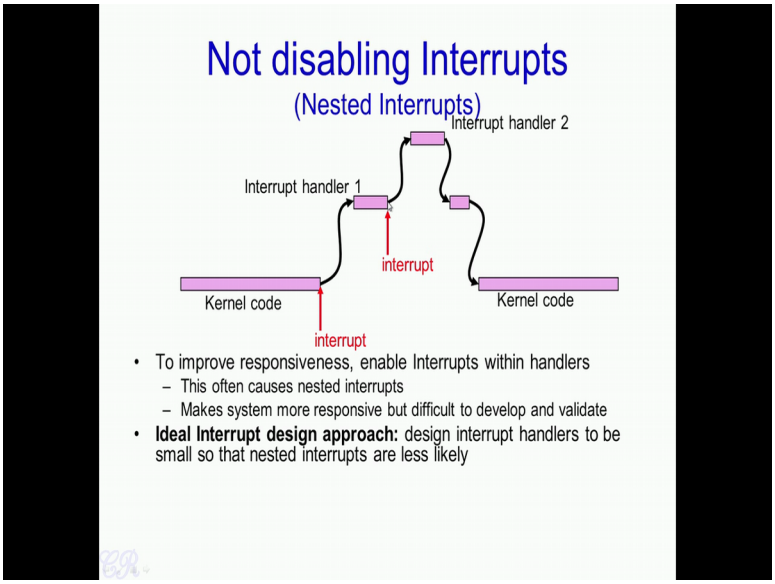
(Refer Slide Time: 11:25)




So the Interrupt Latency could vary between a Minimum and a Maximum in a system. The Minimum Interrupt Latency is due to the delays within the interrupt controller. So the system would typically not be able to get interrupt latency which is less than this minimum latency specified by the controller.

On the other hand, the Maximum Interrupt Latency is due to the way the operating system is designed. Some operating systems for instance would disable interrupts when doing important jobs such as handling an other interrupt or doing some atomic operations. So, during this process if a new interrupt occurs, that interrupt would have to wait until the previous interrupt completes or the atomic operation completes. So, as a result you will get an increase in the interrupt latency.

(Refer Slide Time: 12:20)



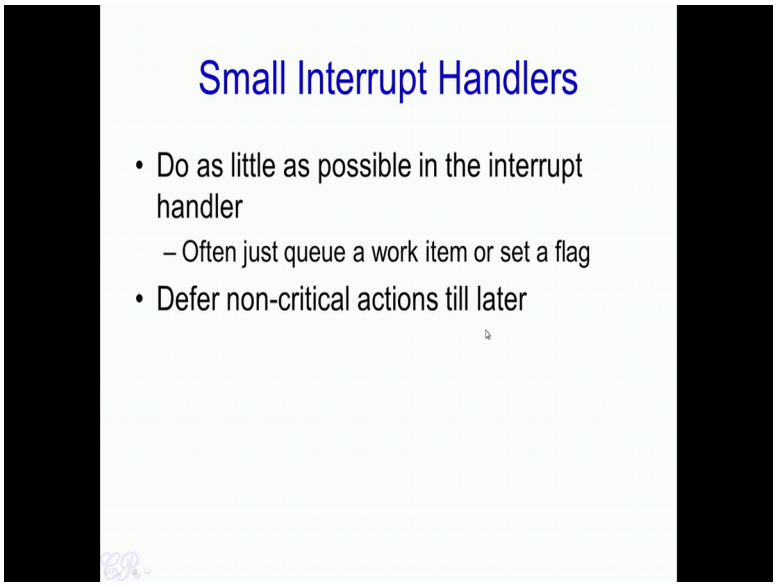

So, one way to reduce interrupt latency is by not disabling interrupts. However, this could result in what is known as Nested Interrupts and is depicted in this particular figure (refer above image). So let us say the kernel code is executing in the CPU and after sometime an interrupt occurs due to which the interrupt handler begins to execute. So, while the interrupt handler executes, a second interrupt of a higher priority occurs and this would lead to the second interrupt handler being executed.

After the second interrupt handler completes, the first interrupt handler continues from where it had stopped just before the interrupt occurred, that is it had stopped at this particular point (second interrupt arrow in above image) and it will now continue executing from this particular point (refer above image). So, after this interrupt handler finishes executing, the original kernel code will continue to execute.

So, as we see the system becomes more responsive, in the sense that when a new interrupt of a higher priority comes the latency incurred is much lesser. However, the limitation is that nested interrupts makes designing the operating system far more difficult, also validating this particular operating system will be more tedious. So therefore, as far as possible we would like to design interrupt handlers to be extremely small so that such nested interrupts are highly unlikely. For instance, if we design this interrupt handler 1 (mentioned in above image) in such a way that it would have completed its execution at this point itself (where red arrow points), then there would be

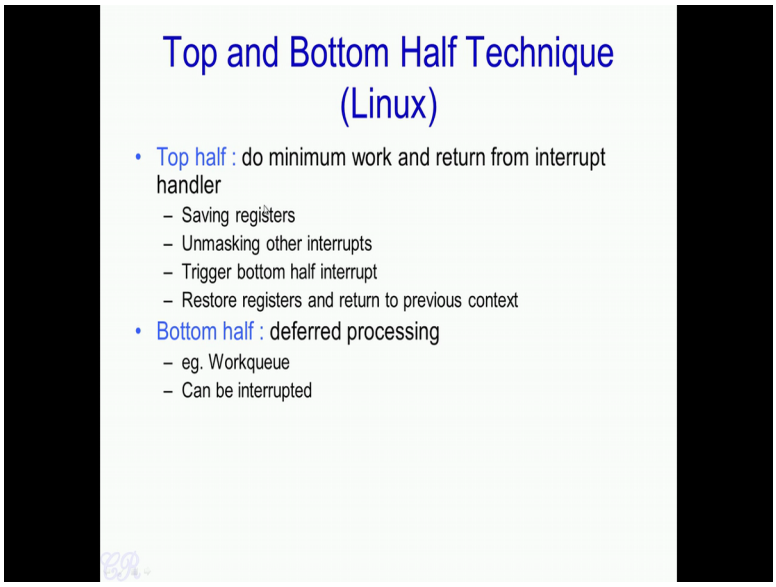
no need to actually nest the second interrupt.

(Refer Slide Time: 14:10)




One way to actually achieve small interrupt handlers is to design it in such a way that only the required crucial and critical operations are performed in the interrupt handler. All other operations are deferred to later, that is all other non-critical actions are deferred to later.

(Refer Slide Time: 14:29)

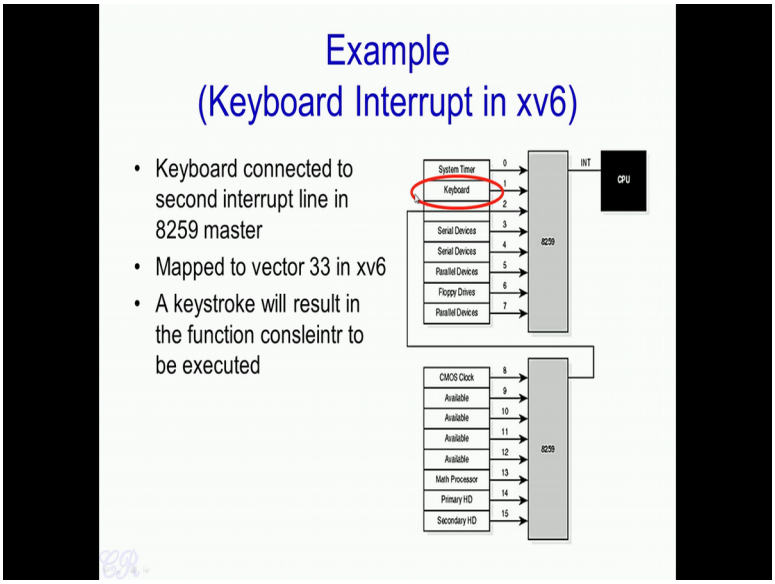



In Linux, this is achieved by having a top half interrupt handler and a bottom half interrupt handler. The top half interrupt handler gets executed first, and does the

minimum amount of work which is critical and then returns from the interrupt. So the critical work involved is the saving of registers, unmasking of other interrupts, triggering the bottom half of the interrupt handler to execute and restoring registers and returning to previous context.

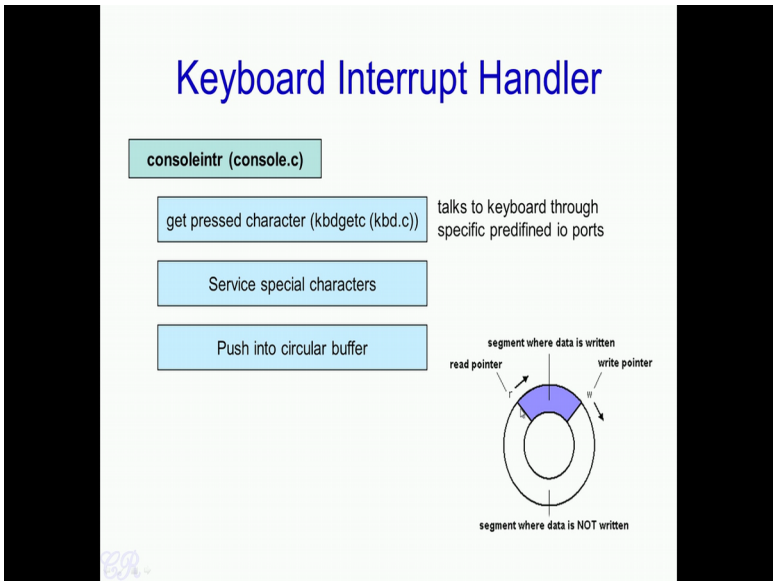
So, some time later the bottom half interrupt handler then executes. The Bottom half interrupt handler would typically fetch some data which the top half interrupt handler has sent to it, through a say for instance a queue and it will process that particular data. So, unlike the top half interrupt handler, the bottom half interrupt handler can be interrupted.

(Refer Slide Time: 15:21)




So let us take an example of interrupt handling in xv6. In particular we will see about interrupt handling with respect to the keyboard. So, we have seen this particular figure before (refer above image) and we have seen that the keyboard is connected to IRQ 1 of the master 8259. So, when a key is pressed then it results in this particular line 1 (red in circle) being asserted and the master 8259 will then transfer the interrupt to the CPU through the INT pin. The CPU would then detect that a keystroke has been pressed, and determine the corresponding interrupt vector which would result in this function consoleintr() to be executed.

(Refer Slide Time: 16:08)




Now, in the consoleintr which is present in console.c in the xv6 code, what is first done is that the function would communicate with the keyboard using the keyboard driver present in kbd.c to determine which key has been pressed. Now, if it was a special key then there is a special servicing done for special characters pressed in the keyboard and then the data is pushed into a circular buffer. So the circular buffer is as shown over here (mentioned in above image) - it has got a read pointer and a write pointer. So the consoleintr will push the data at the memory pointed to by the write pointer and then increment W. So, at a later point other functions in the operating systems would then read data using the read pointer and therefore, be able to determine what are the keystrokes which been pressed.

So, this was the brief introduction to interrupts and how interrupts are worked and how they are handled by the CPU and the operating systems.

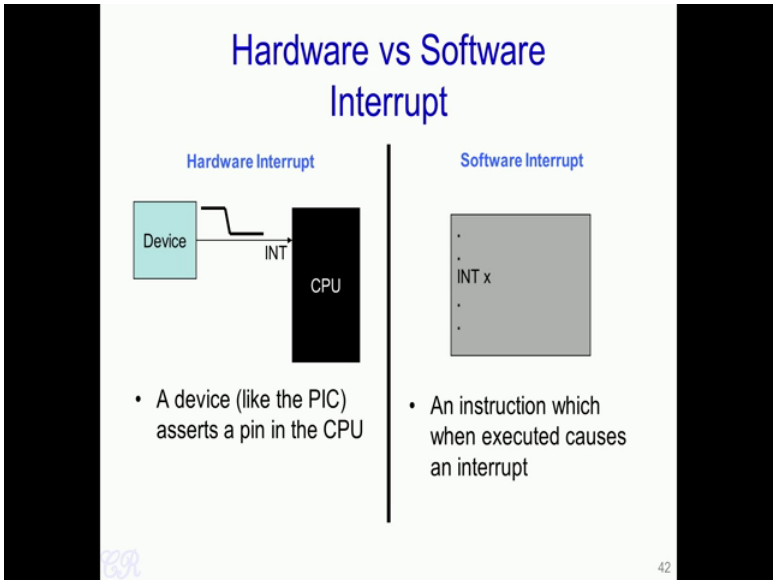
In the next video we will look at software interrupts and how they are used to implement system calls in the operating systems.

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

**Week - 04**
**Lecture – 16**
**Software Interrupts and System calls**

In this video, we look at an important type of interrupts known as Software Interrupts; and their applications in system calls.

(Refer Slide Time: 00:28)

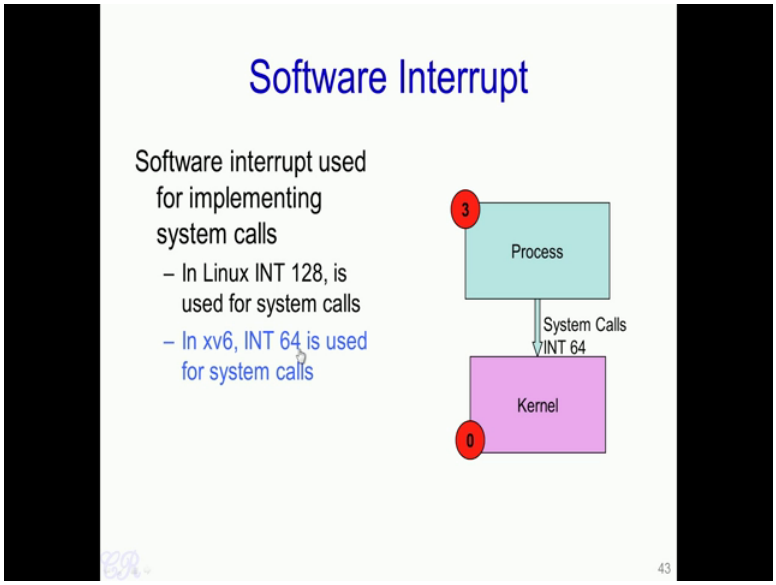



In the previous video, we had looked at hardware interrupts. We had seen how a device such as a keyboard or a network card could assert a particular signal in the CPU and this would cause the CPU to asynchronously execute an interrupt handler corresponding to the device. So, as we have seen in the previous videos, this device (refer above image) would typically send a signal to the CPU through an intermediate device such as a PIC or a programmable interrupt controller.

In much the similar way, we have what is known as a Software Interrupt. However, unlike having an external device which causes the interrupt, here an instruction in the program would trigger the interrupt. In this particular case, for example, an instruction such as INT (refer above image) would cause the interrupt to occur and the operating system to execute. So, here the instruction is INT x, so x here is the interrupt number. It

typically has a value less than 256, and it is used to specify or distinguish between software interrupts.

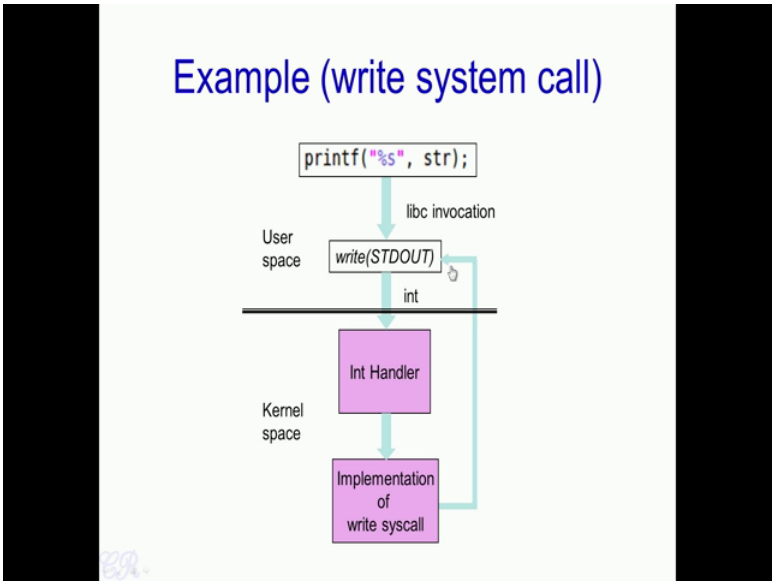
(Refer Slide Time: 01:54)




So, where is the software interrupt used? So, software interrupts are used to implement system calls. So, as we know a user process (mentioned in above image) could invoke a system call to perform some Kernel operation. For example, it could be to read a file or to write a file, to print something to a monitor, or to send a packet through the network and so on. More specifically, all operating systems implement system calls through one particular software interrupt.

For example, in the Linux operating systems, the software interrupt 128 is used to specify system calls. Therefore, in a Linux OS, if I have a INT 128 which is executed in the user process, it would lead to an interrupt that occurs and cause the kernel or the operating system to execute, and thereafter the OS would execute code depending on the interrupt. In xv6, the software interrupt used to implement system calls is 64 or instruction like INT 64 in the user process (refer above image) would be meant to implement a system call.

(Refer Slide Time: 03:19)



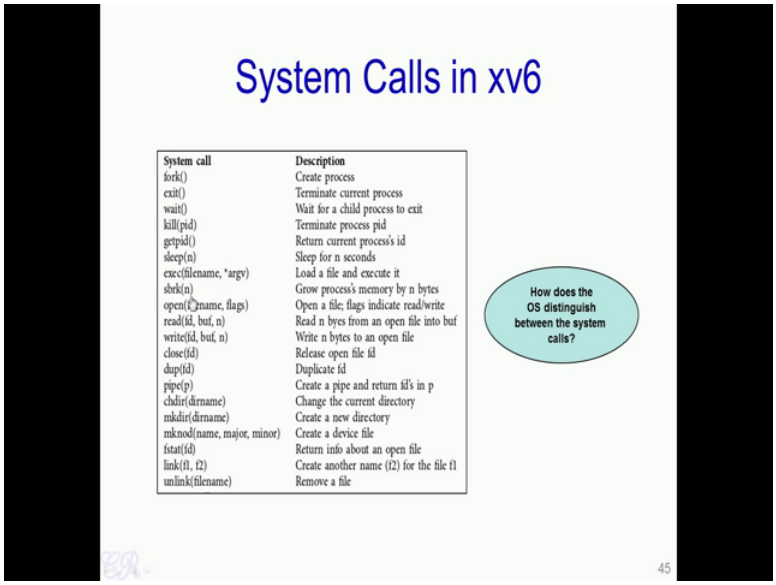

So to take an example, let us consider that our application has a printf statement present in it. So, printf would print this string ‘str’ (mentioned in above image) to the standard output, which typically is the monitor. Now printf is a function present in the library libc and it would cause the libc function to be invoked. Now in the libc function there is a call to the write system call with the specifier STDOUT i.e write(STDOUT). So, the STDOUT here is the file descriptor; and it is a special file descriptor which is meant for the standard output or the monitor.

So in the write function, it would invoke INT 64 in xv6 or INT 128 in Linux and cause a software interrupt to occur. So, the software interrupt as we know would cause the transformation from the User space to the Kernel space and it would result in the operating system executing. The OS would then determine that the interrupt was in fact due to a system call and then it would determine what system call it was from; in this case, it was from a write system call and it was a write to the STDOUT - the standard output.

So the operating system would then invoke the handler for the write system call, and this handler (second pink box in above image) would take care of communicating with the various devices such as the video card to display the string onto the monitor. So, after this handler completes execution, the IRET instruction is executed which would result in

a transformation back from Kernel space to the User space and the program will continue to execute.

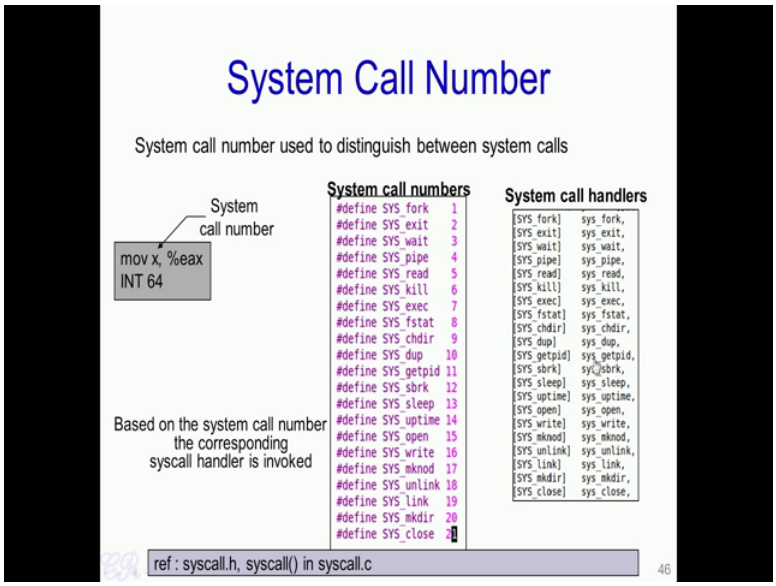
(Refer Slide Time: 05:31)




Now, typically operating systems support several different types of system calls. So, this particular table over here (refer above image) shows the various system calls supported by xv6. So, in a previous video we had seen some of them. For example, we had seen the fork(), exit(), wait() so on. And you are also familiar with several types of system calls such as open(), read(), write(), close(), change directory chdir(), make directory mkdir() and so on. So, each of these system calls would be executed by having a software interrupt such as INT 64, because it is the xv6, so it is 64. So, each time any of these system calls are invoked by a user process, it would trigger the operating system to execute.

Now, the next obvious question that one would ask is from the OS perspective, how does the OS distinguish between the various system calls? So, we mention that all the system calls will use either INT 64 for xv6 and INT 128 for Linux. So how does the OS determine whether the system call was with respect to fork(), wait(), sleep(), exit(), and so on. Essentially this distinguisher comes from the user process itself.

(Refer Slide Time: 07:09)



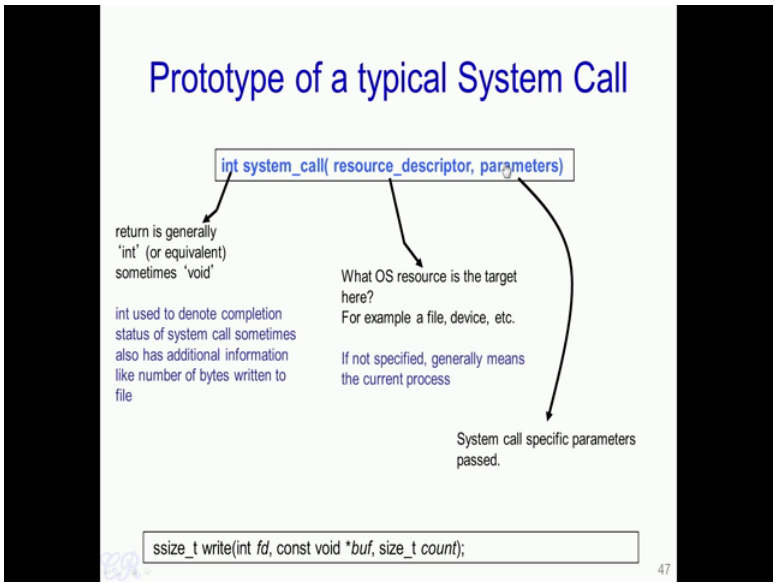

What happens is that, before the INT 64 instruction, the user process will move a system call number to the eax register. So, for example, instruction mov x, %eax will move the system call number to the eax register. Now each system call in the operating system will have a unique number, system call number. Now the operating system when triggered by the INT instruction would look up the eax register, and then determine what system call was invoked. For example, in xv6, if we look up these particular header files (mentioned in above image), we would see the various system call numbers defined. For example, we have each system call given a specific number, for example SYS_fork given 1, SYS_exit giving 2 and so on.

Now when the OS gets trigger due to the INT 64 instruction getting executed, the OS will determine the system call using this system call numbers and then invoke the corresponding system call handler. So, each of the system calls also have a corresponding system call handler. This is shown over here (refer above image), corresponding to each of the system call numbers SYS_fork that is 1, SYS_exit it is 2 and so on.

So, these are system call functions (mentioned above in System call handler list) present in the operating system which gets triggered based on the type of the system call. For example, if eax had a value of 11, the operating system will look in to the eax register and determine that this corresponded to the getpid system call being invoked. And then it

would look into this particular table (mentioned above in System call handler list) and see that the getpid system call is handled by this function sys_getpid, and therefore it will then invoke this sys_getpid function.

(Refer Slide Time: 09:36)




Now, let us look at the typical prototype of a system call. So, a typical system call is as shown over here (refer above image). So, of course, it has a system call name that is a function name and then it is passed some resource descriptor and parameters and typically would return an integer. So, the resource descriptor specifies what operating system resource is the target here. For example, it could be a file or a device; and as we have seen in the previous slides it could also specify a particular monitor. For example, if the resource descriptor is STDOUT then the resource in use here is the monitor.

So, some system calls also do not specify this resource descriptor; in such a case, the system call is meant for that resource itself. For example, if we use the sleep system call, we only specify the time and no specific descriptor such as the file, a device and so on. This means that the current process wants to sleep for that given interval time.

The next part is the parameters. So, these parameters are specific for the system call. For example, if we invoke read, write, open or close or any other system call, the parameters specified here is going to be very specific for each of these system calls. For example, the write system call (mentioned in above image) has the parameter *buf that is a void pointer and the count. So, the open or the close or any other system call would have

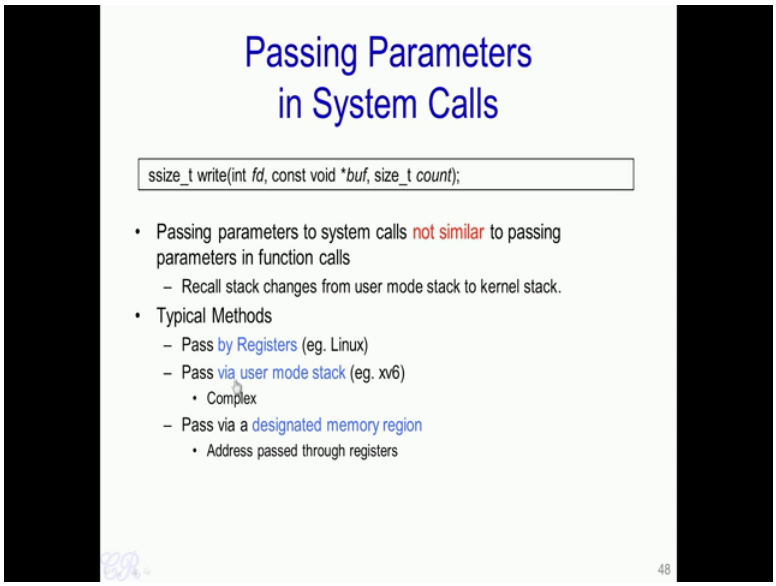
different set of parameters. So, essentially these parameters are very specific to the type of the system call.

The return type is typically int or integer, and sometimes it is a void. So, ‘int’ is typically used because in this way the operating system will be able to send the completion status of the system call, whether it had executed successfully or it had failed and so on. So, sometimes the return is also used to specify certain specific information about the system call. For example, in write (mentioned in above image) the return is ssize_t which in fact, is typdef to integer and it specifies number of bytes that have been written to the file, specified int fd. So, the return type could also vary depending on the type of system call.

The next thing what we will look at is how these parameters that is the resource descriptor and the parameters pass to the system call are sent to the kernel. So, note that system calls are invoked very differently from a standard function call. So, in a function call, as we know the instruction call would be used and the call would specify a address which is where the function would reside and the various parameters for the call are passed through the local stack.

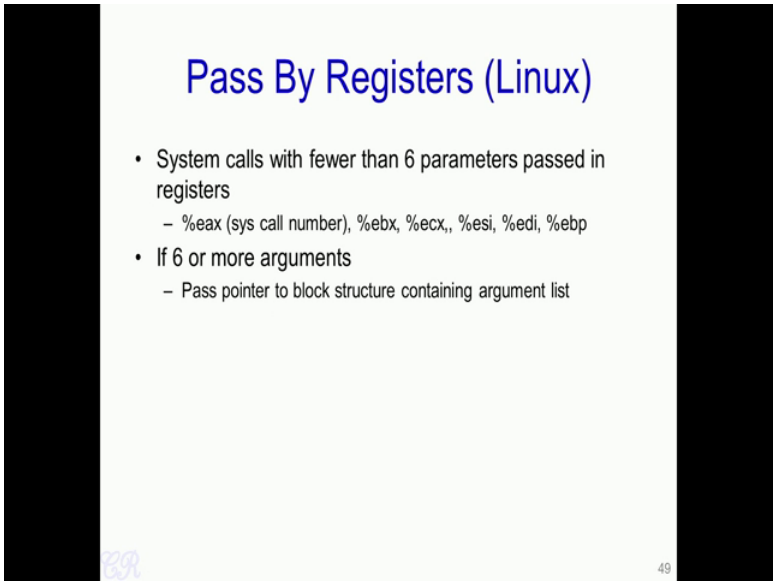
Similarly, the int return which is returned from the function call would be returned through the eax register. So, system calls on the other hand work very differently from function calls. So, as we have seen system calls invokes the kernel by the INT 64 instruction as in the case of xv6. So, how are the parameters such as the resource descriptor and the other parameters passed to the kernel?

(Refer Slide Time: 13:49)



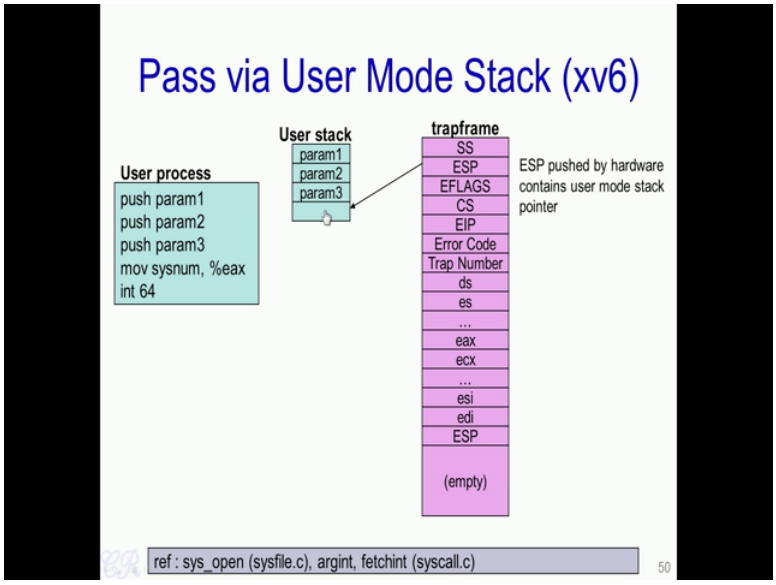

So essentially, there are three ways of doing so. The first is by pass by registers which is typically done in Linux; the second way is by passing through the user mode stack which is done in xv6; and the third way is by passing through a designated memory region. So, in this particular case (third case), what is done is that in the user process itself, a designated region most likely in the heap would be used to save the various parameters that is needed to be passed to the system call; and the address to this region in the heap is passed through the registers. So, we will look at the other two cases that is pass by registers and pass via user mode stack in more detail.

(Refer Slide Time: 14:44)



Now Pass by Registers which is used by Linux system calls would use the registers present in the processor to pass parameters to the kernel. So, we know we have already seen an example of this, of how the %eax register is used to pass the system call number from the user process to the operating system. In a similar way, other registers such as the %ebx, %ecx, %esi, %edi, and %ebp are used to pass the various parameters of the system call from the user process to the kernel. If the system call has more than 6 arguments then a pointer to a block structure containing the argument list is passed to the kernel.

(Refer Slide Time: 15:41)



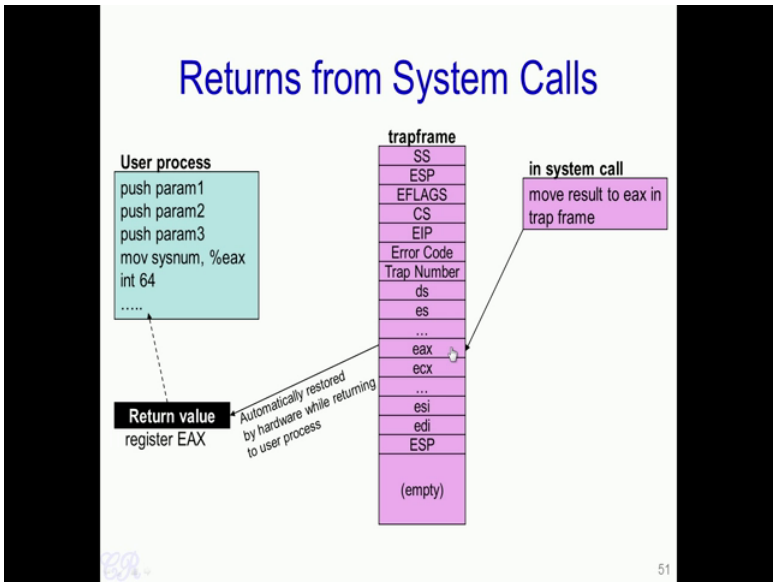

Now, let us look at the second case that is Pass via the User Mode Stack and this is what is done in xv6. So, in this particular way, before the int 64 instruction (mentioned above in User process box) is present various parameters in the system call are pushed onto the stack. For example, if the system call had 3 parameters; param 1, param 2, and param 3, so these three parameters are pushed into the user space stack and then the system call number (number before int 64) as we have seen is moved into the %eax register. So, this here is the user space stack of the user process containing the 3 parameters.

Now, when the INT instruction is executed, as we know, it triggers an interrupt causing the switch from the user space into the kernel space. Also as a result of this interrupt execution, as we have seen, there is the switch in the stack from the user space stack to the kernel space stack. And what we have seen in the previous video that this kernel

stack is used to create what is known as the trapframe. So, this trapframe is shown over here (mentioned in above image). So, what we have seen in the previous video that some of these entries in the trapframe are pushed into the stack automatically by the CPU. So, in particular, these registers specified in capitals letters are all pushed onto the kernel stack that is on to the trapframe by the CPU.

Now, the SS and ESP here (in trapframe box) is important for us. So, these are the stack segment and the stack pointer, and these registers correspond to the user space stack. So, as we know before the INT 64 has been executed, the last known stack pointer was pointing to this particular location. And therefore, the contents of ESP will also point to this location in the user space stack. So, in this way, the kernel will then use the SS and the ESP from the trapframe to determine the various parameters for the system call.

(Refer Slide Time: 18:22)




The next thing we will look at is how the return value is passed from the system call back to the user process. Again we will recollect that the entire reason for creating this trapframe in the kernel stack for the process is due to the reason that when the interrupt or the system call completes its execution, the entire state in the trapframe is restored back into the corresponding CPU registers, and it would result in the user process continuing to execute from where it had stopped, and also the context of the user process is restored with the help of the trapframe.

Now, in order to return a value from the system call which is executing in the kernel space back to the user space, so what is done is that the eax register in the trapframe is modified; essentially, we had seen that the eax register because of this particular instruction (mov sysnum, %eax) would contain the system call number.

Now this system call number is overridden by the return value of the system call. So, this could be a negative number like -1 or a positive number as we have seen in the earlier slide. So, now when the system call executes and completes its execution and the context is transferred back to the user process, the entire trapframe including the new value of eax in the trapframe is restored back in to the registers of the CPU. The process continues to execute from this particular instruction (after int 64) with the new value of eax, which contains the return from the system call.

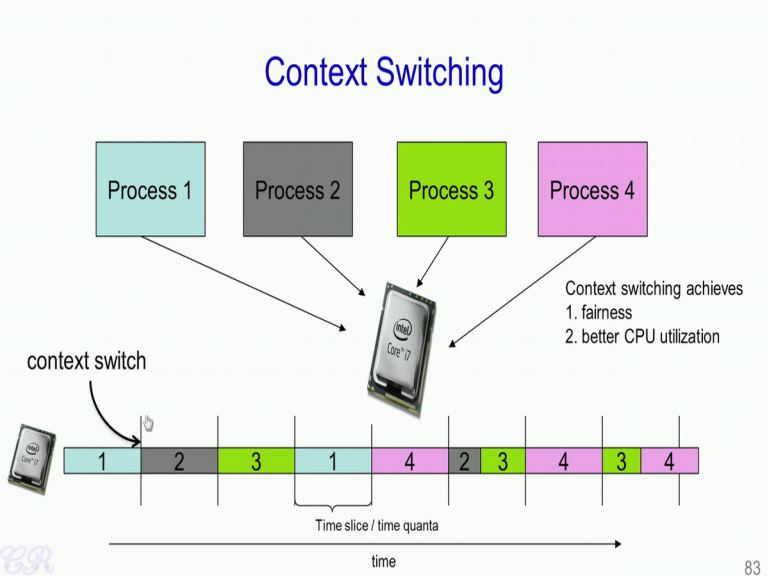
Thank you.

**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)

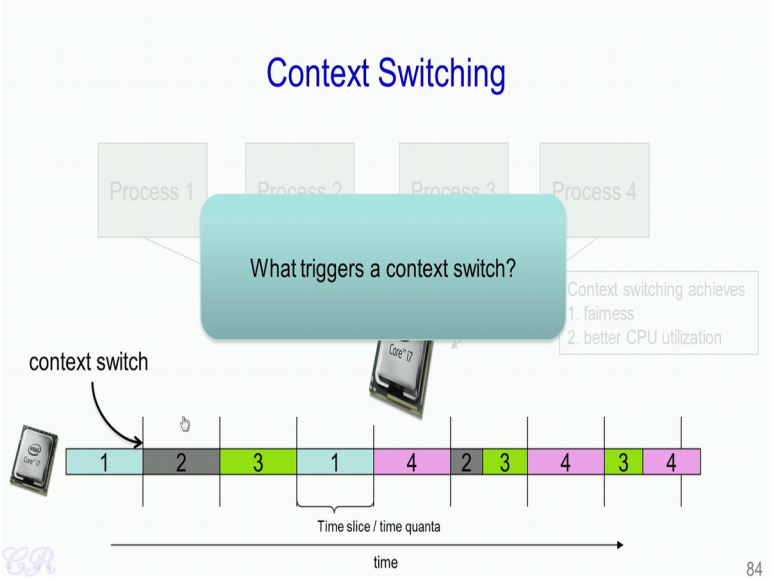



So let us consider this particular figure (refer above image), where we have a single CPU in the system which is shared among multiple processes. So, the operating system by a feature known as Multi Tasking enables that this CPU is fairly shared among the various processes. So, in a multi tasking environment 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 (as per above example) process 2 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

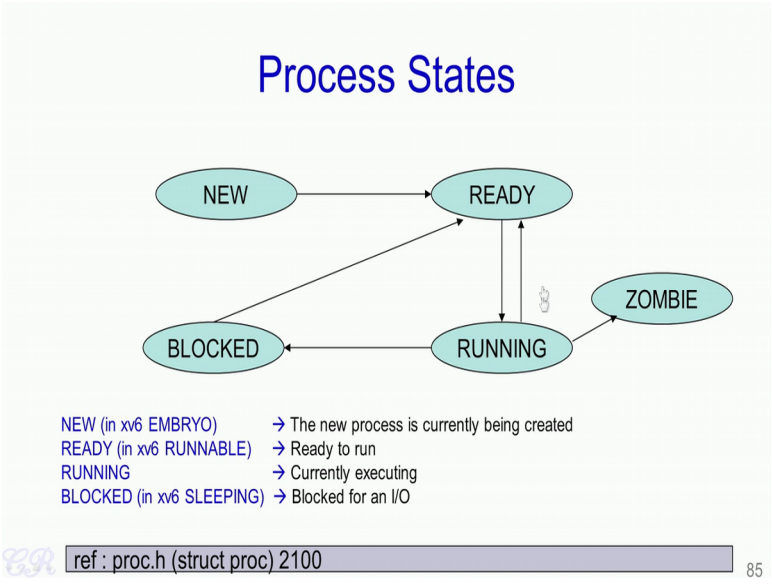
environment such as this (mentioned in above image), processes execute for brief intervals of time known as Time slices. So, for instance process 4 executes in this time slice (between process 1 and 2) then there is a period where it does not execute and then it continues executing in this time slice (between process 3 and 3) 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)



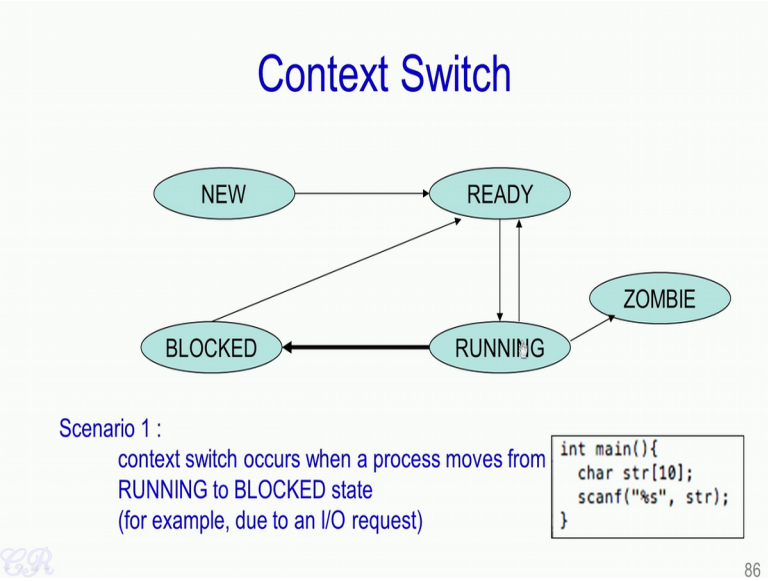

To begin with, we will see what triggers a context switch in an operating system?

(Refer Slide Time: 02:07)



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 different states, and this is represented by this process state diagram (refer above image). What we will see now is how these various states will trigger a context switch to occur.

(Refer Slide Time: 02:31)

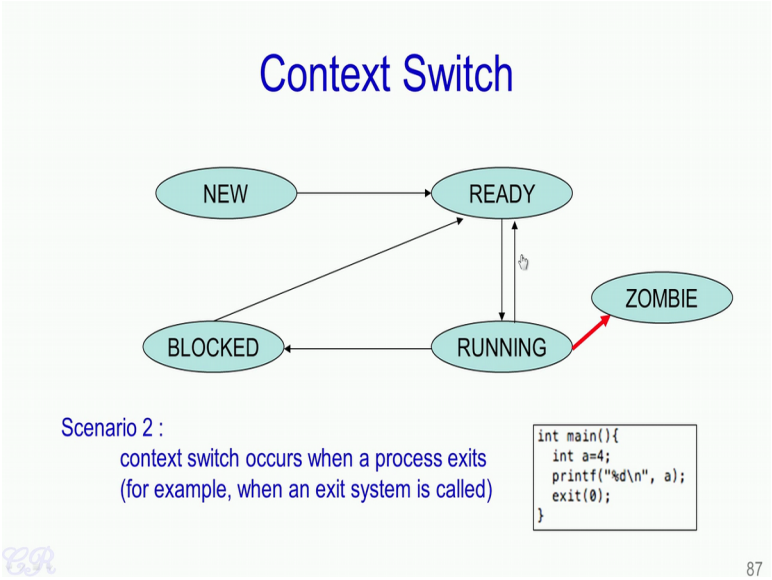



Let us see this 1st scenario about how a context switch gets triggered. Let us consider this particular program (refer above image) which essentially invokes the scanf. Therefore, it requires a user to input something through the keyboard.

Let us say this function (scanf() ) 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 blocked state. Now the process will remain in the blocked state until the user has input data into the console.

So 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.

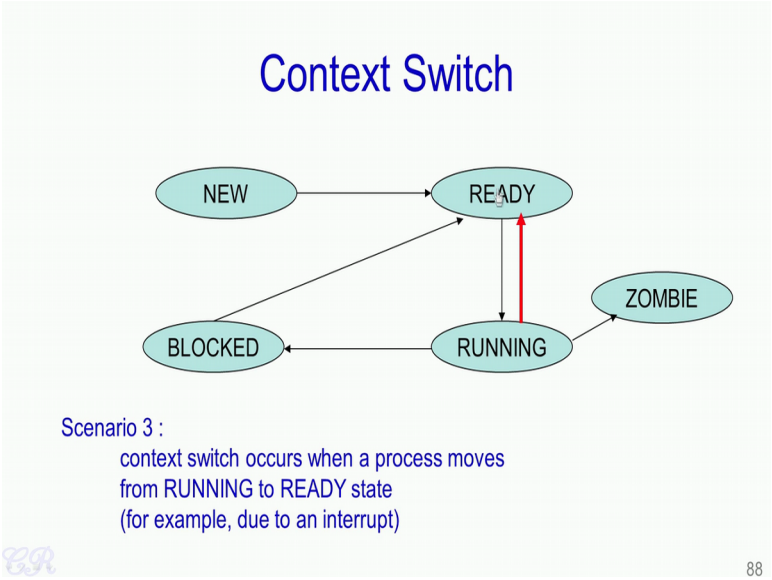
(Refer Slide Time: 04:25)




So let us take the 2nd scenario where context switch can occur. So let us say that we are considering this particular program (refer above image), where there is a printf and then there is an exit and 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.

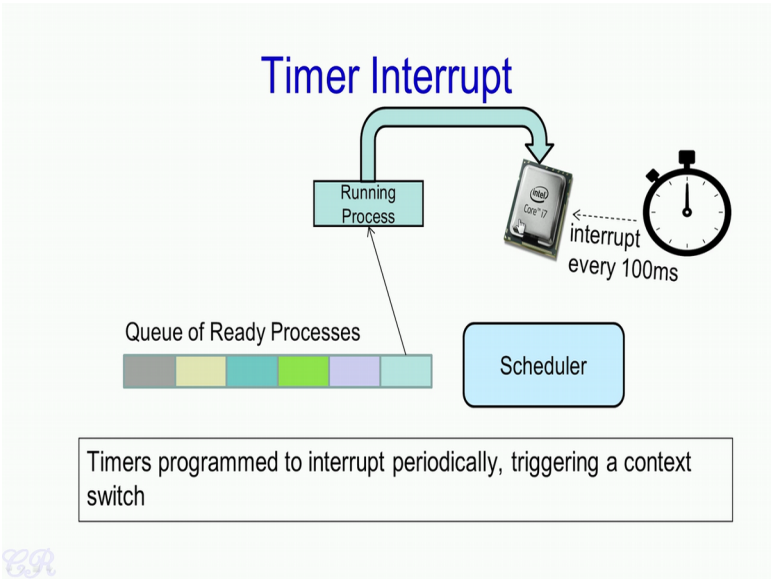
So at the end of the exit system call, the operating system will trigger a context switch. So, this will ensure that the current process which is just in a exiting mode will not have the CPU resources anymore, but rather a new process will be assigned the CPU. Therefore, a new process will come from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

(Refer Slide Time: 04:31)




Now, let us look at the 3rd scenario that can occur. So, if a process is in a running state, and an event such as an 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 it 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 into the running state.

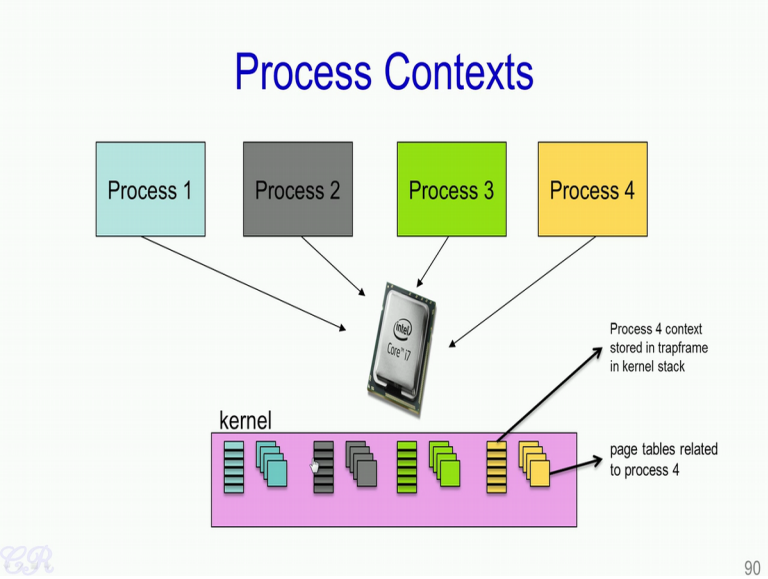
(Refer Slide Time: 05:14)



A very popular interrupt in this context is the Timer interrupt. Now all systems have a timer within them. So, this timer (refer clock timer in above image) 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 (mentioned in above image) which are in the ready state, and then based on some algorithm will choose 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)

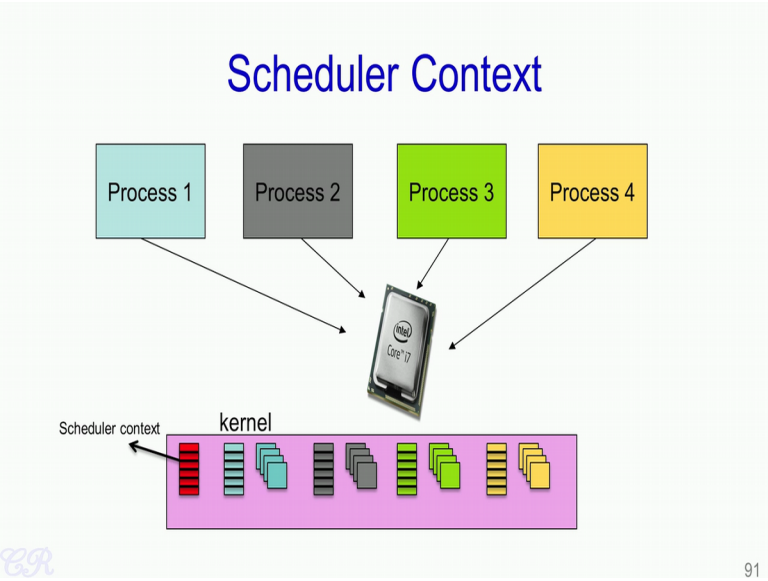



So now let us look more into detail about how a context switch occurs. So, in a previous video we had seen that corresponding to each process in the system, the kernel stores some metadata. Essentially there are three metadata for each process, they are; the process control block, the kernel stack and 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 (mentioned in above figure in kernel block) would allow the process to restart

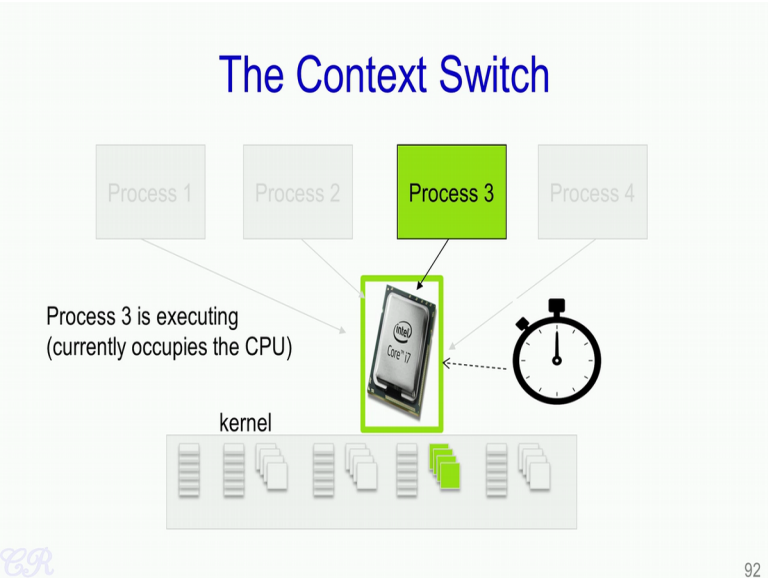
execution from where it had stopped. So, in this particular figure for instance (refer above image), each process has its own trapframe or own kernel stack which was the trapframe when an interrupt occurs, and its associated page tables. Similarly, process 3 has its own trapframe and page tables, process 1 and process 2 as well.

(Refer Slide Time: 07:31)




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.

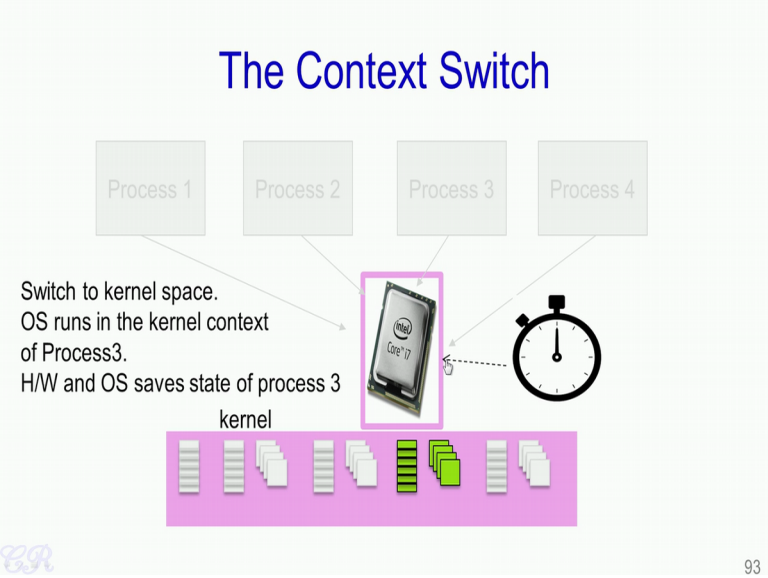
(Refer Slide Time: 07:45)



So 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 (refer above image) 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 3's page table which does the translation from the logical address into the physical address.

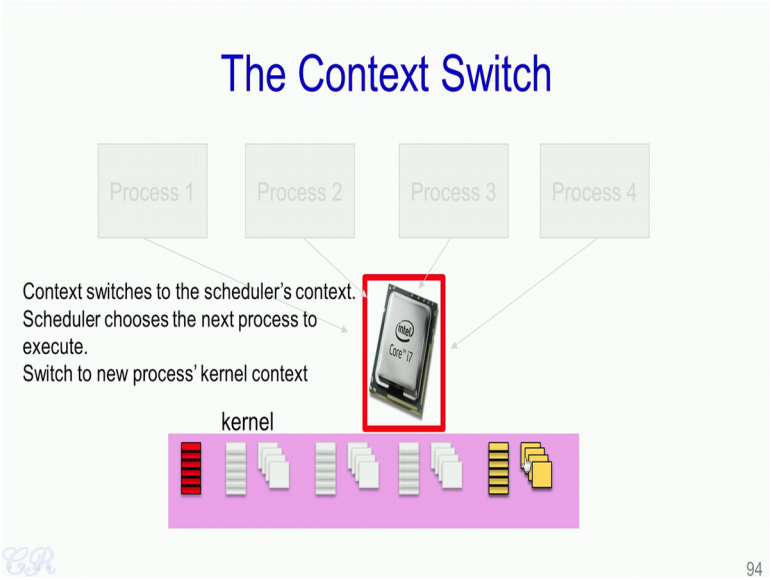
Now, when an interrupt occurs we have seen that there is a switch from the user mode into the kernel mode.

(Refer Slide Time: 08:26)




Therefore the kernel begins to execute again, and it is the kernel which will now hold the CPU. All instructions that are being executed in the CPU, thus 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 (green stack in above image) is stored in a structure known as the trapframe and this trapframe as we have 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.

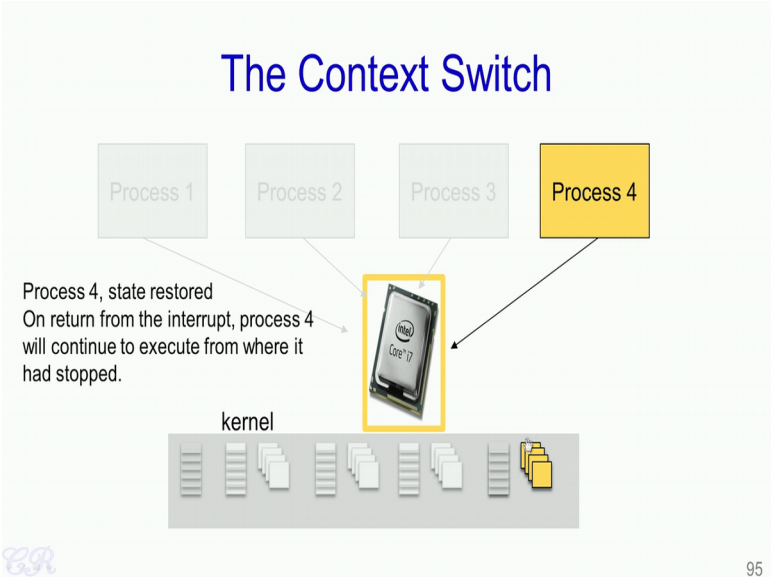
(Refer Slide Time: 09:19)




The scheduler would then switch from the kernel stack of the user process to its own stack or the scheduler stack (red stack in above image); therefore, it would obtain the scheduler context. The scheduler then chooses from the ready list, the next process to be executed in the CPU and then there is a switch to the new process's kernel context. Thus we are moving back from the scheduler's context back to the new process. In this case process 4 that has been selected by the scheduler and we switch back to process 4's page tables and process 4's kernel stack (mentioned in above image in yellow color).

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 were process 4 had stop executing.

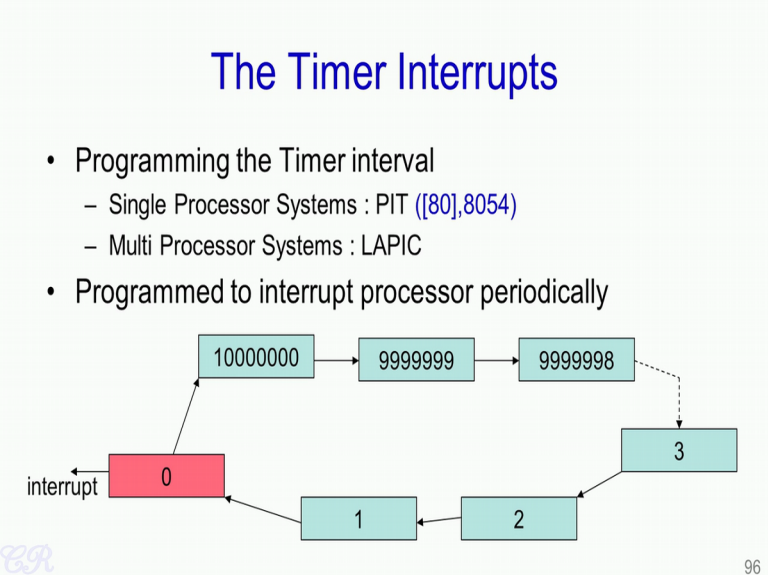
(Refer Slide Time: 10:23)




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

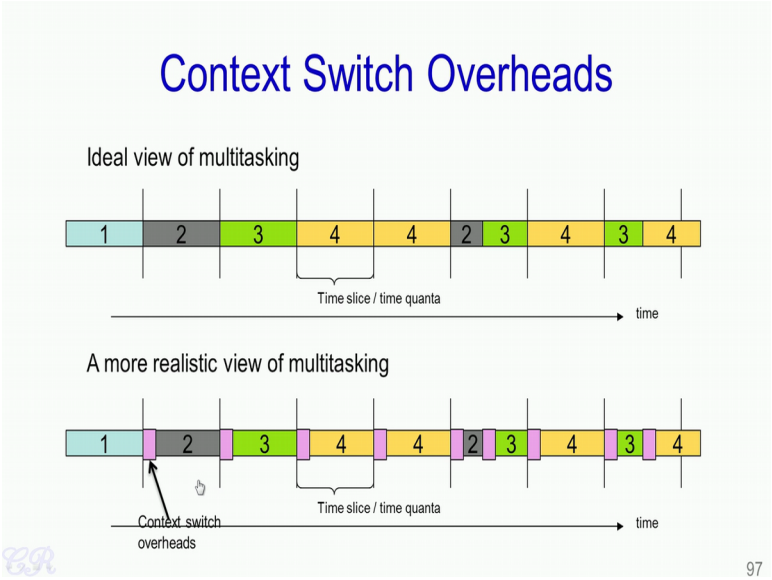
(Refer Slide Time: 11:21)




Now, let us look how timers actually interrupt processors. As we have mentioned before all systems have a timer present in them. In legacy system 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 (10000000 mentioned in above image), 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 loaded back with the original value (10000000) and this process continues. So, essentially the counter keeps going from this large value to 0 and then keep cycling across these values. Now when 0 is obtained by the counter an interrupt is asserted 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.

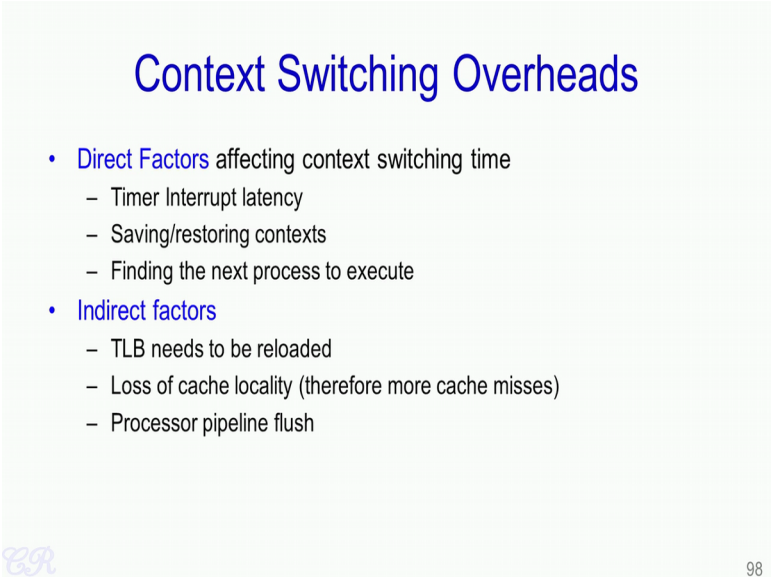
(Refer Slide Time: 12:42)




This figure over here (refer above image) 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.

So, 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 (process 2 in above image) execute in the CPU. So, this time difference between when the first process is preempted from the CPU to when the second process starts to execute is the context switch overheads; now, this context switch overheads could be significant.

(Refer Slide Time: 14:04)




The factors affecting the context switching overheads can be classified as either the direct factors or the indirect factors. Among the direct factors these are the three examples of direct factors (mentioned in above image), 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 overheads in the contexts switching. Essentially there would be time taken for saving and restoring context for the various processes. In addition to this overheads are also caused by the scheduler, which needs to choose the next process to execute in the CPU.

The indirect factors are more subtle 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 aside 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 change 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 overheads. 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 as 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 overheads.

Another aspect is that every time an interrupt occurs, the processor pipeline would need to be flushed. So, this also adds up to the overhead. The context switching could incurs significant overheads 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 have also seen overheads that are incurred due to context switching.

Thank you.