

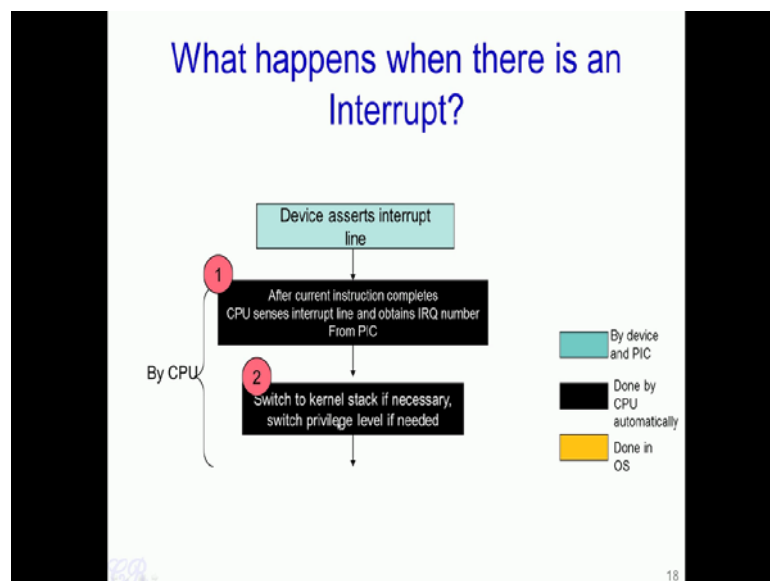
Introduction To Operating Systems
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

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 interrupt service routine being executed.

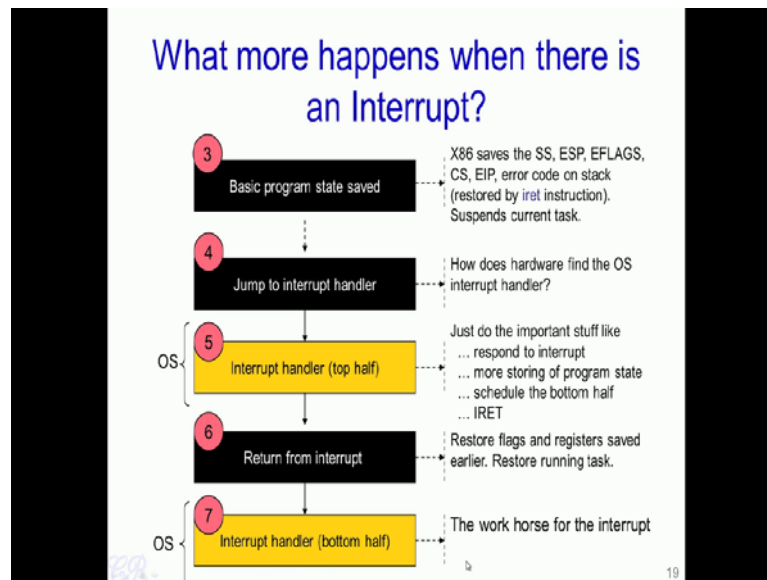
So, in this video we will look at more into detail how interrupts are handled. Interrupt Handling is a pretty involved process which involves both the CPU as well as the operating systems. So, we will go step by step and see what are the various stages in interrupt handling.

(Refer Slide Time: 00:50)



Let us start with a first. Let us say that the device asserts and 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? In the processor called CPU over here 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, the processor would switch to the kernel stack and also switch the privilege level if required.

(Refer Slide Time: 01:31)

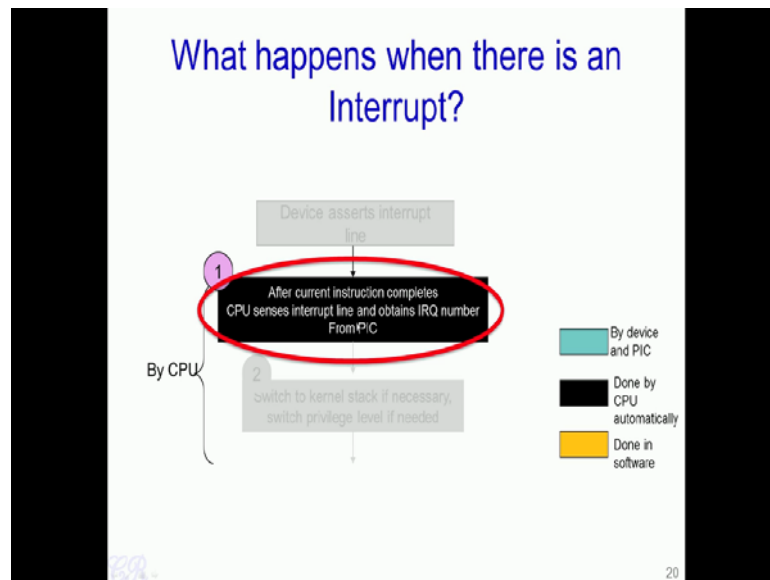


The third step is the current execution of the program is stopped and the program state is saved. So, in x86 the program state comparison of several registers such as the SS register that is a Stack Segment, the stack pointer, flags register, the code segment and the instruction pointer.

All these are saved on to the stack and then the processor would jump to the interrupt handler. In the interrupt handler, we have the top half of the interrupt handler 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 and after sometime there is the bottom half of the interrupt handler that runs. So, this bottom half of the interrupt handler is essentially known as the work house of the interrupt and does most of the difficult or time consuming jobs.

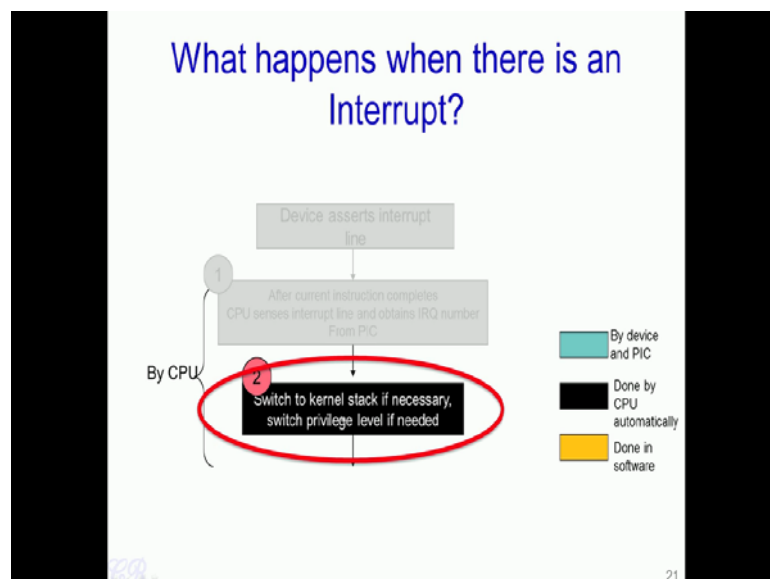
So, one thing to notice is that each of these stages could be done either by the CPU automatically that is the processor hardware itself does this automatically, so these are the blacked boxes, while the yellow boxes are done software, essentially by the operating systems. So, you see that some of these 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)



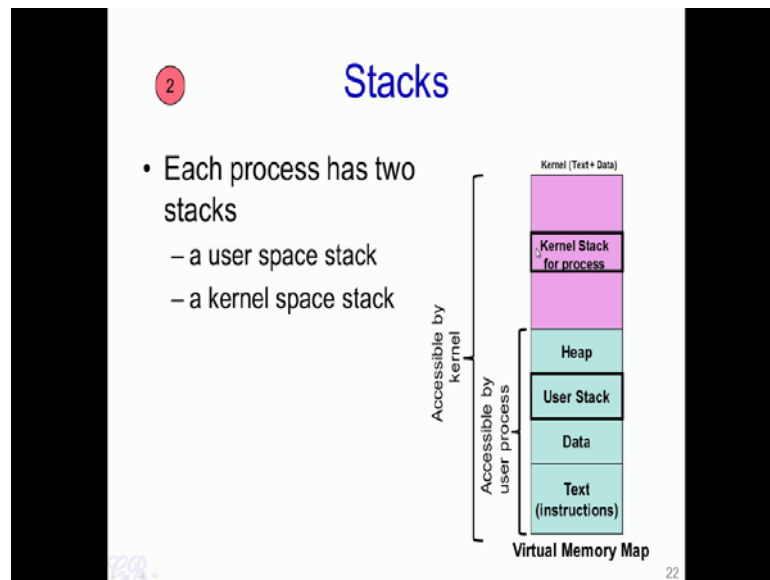
So, after the current instruction completes executing, the CPU senses the INT line and then when it determines that a particular interrupt has been requested it obtains the IRQ number of that interrupt from the PIC.

(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.

(Refer Slide Time: 03:46)



So, we will look at this, we will look at the stacks first. So, as we have seen before each process has two stacks. So, one stack known as the User Stack 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. 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 codes that executes in the kernel.

(Refer Slide Time: 04:33)

2

Switching Stack

- Why switch stack?
 - OS cannot trust stack of user process
 - User processes cannot access the kernel stack
- How to switch stack?
 - CPU should know locations of the new SS and ESP.
 - Done by task segment descriptor
- When interrupt occurs, the OS executes
 - The privilege changes from low to high

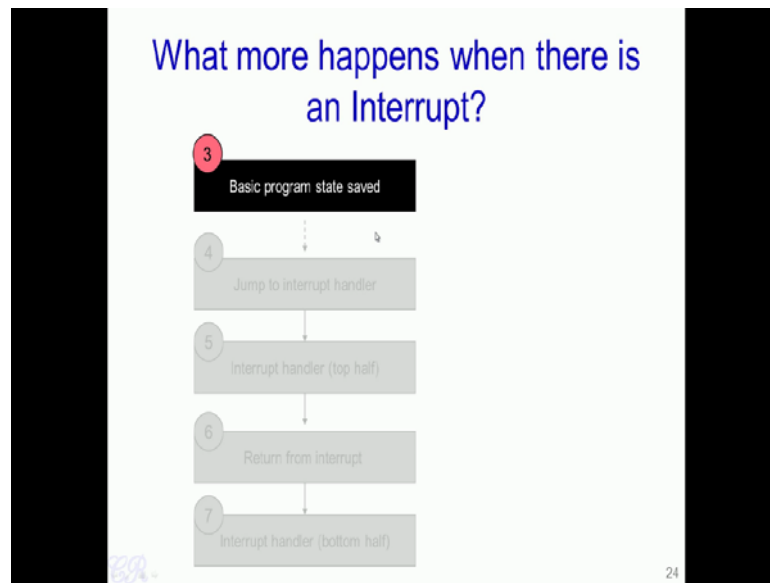
Done automatically by CPU

23

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 malicious virus for instance, it has no access to the kernel stack and therefore, cannot modify or change anything in the kernel.

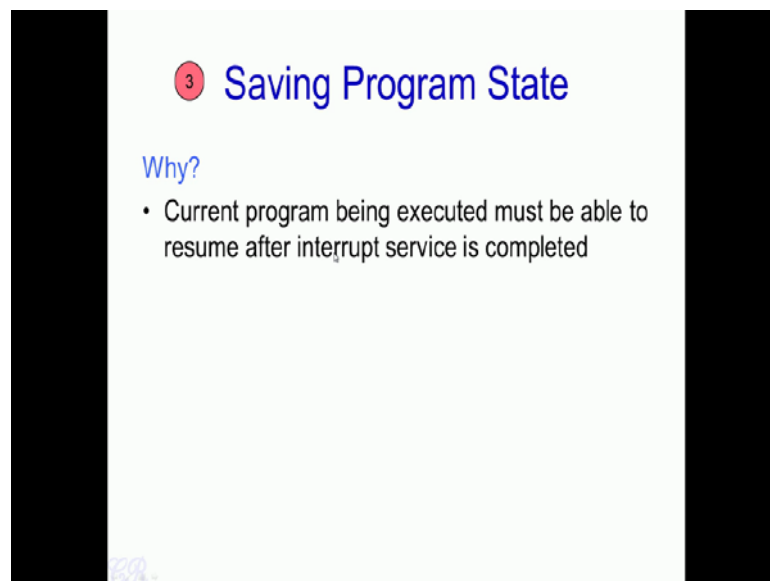
Second thing is how is the stack actually switched from the user stack to kernel stack. So, this is done by something known as the task segment descriptor and essentially what it does is that it changes the stack segment and stack pointer of the processor. The information of the new stack segment and 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)



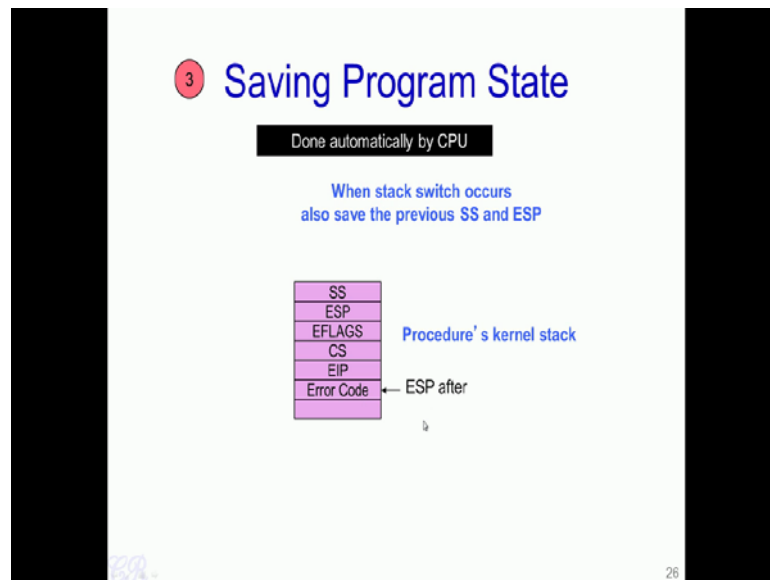
So, after changing stack and raising privilege level, the Basic program state is saved.

(Refer Slide Time: 06:04)



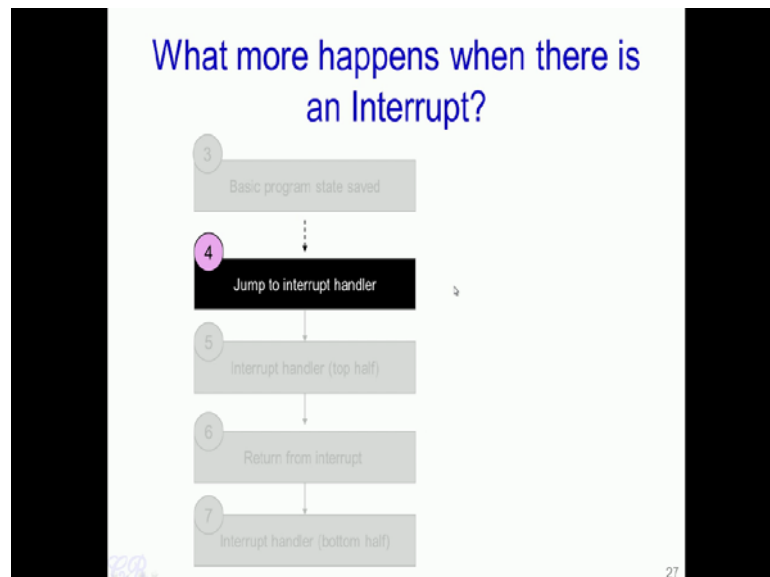
So, what this means is that suppose we have a program which is being executed in user space and an interrupt occurs, then the state of that process is saved. Why do we require to save the state of that process? It is required so that the process could resume after the interrupt servicing is completed.

(Refer Slide Time: 06:27)



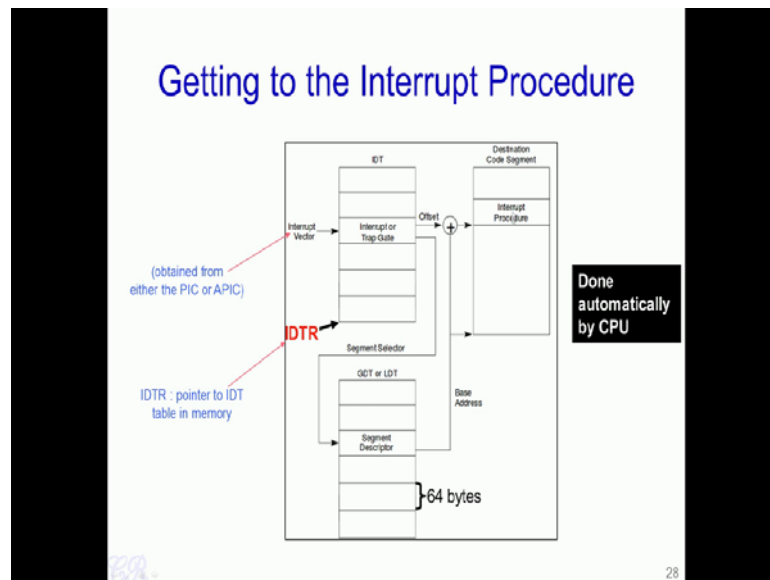
How is the program state saved? In order to save the program state, we will use the kernel stack which we have just pointed to when the interrupt occurred. In the kernel stack we would save the various register such as SS, ESP, EFLAGS, CS, EIP and an Error code if required.

(Refer Slide Time: 06:50)



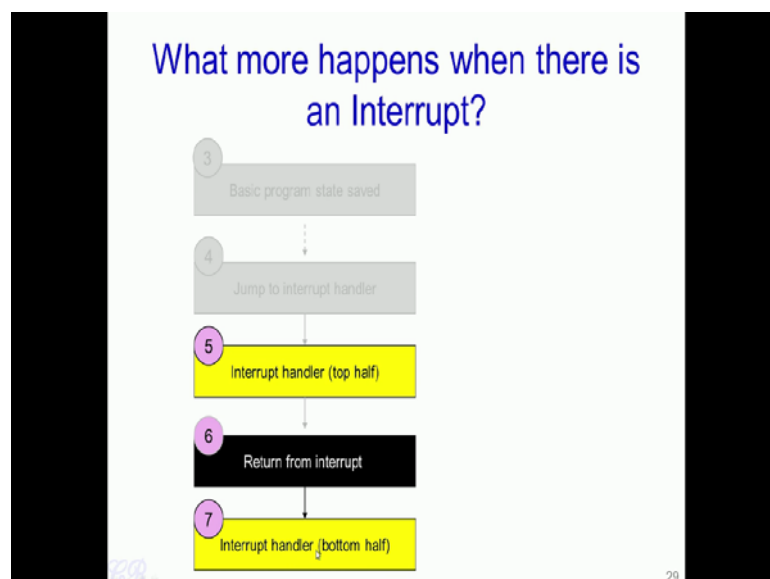
Next, we will see how the processor jumps to the interrupt handler.

(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. The segment selector is used to obtain the base address of the code segment, while the offset is added to the base address to obtain the location of the interrupt procedure. The processor then could begin to execute this interrupt procedure.

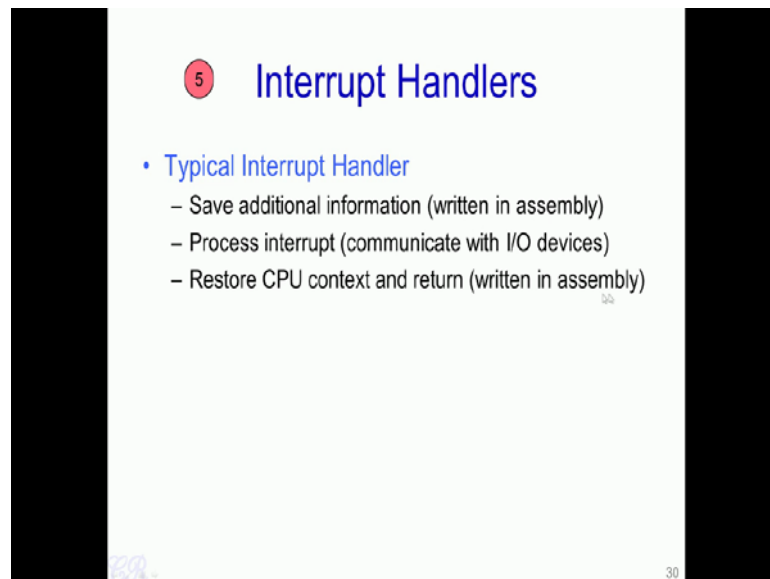
(Refer Slide Time: 07:25)



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

execute the bottom half of the interrupt handler.

(Refer Slide Time: 07:34)



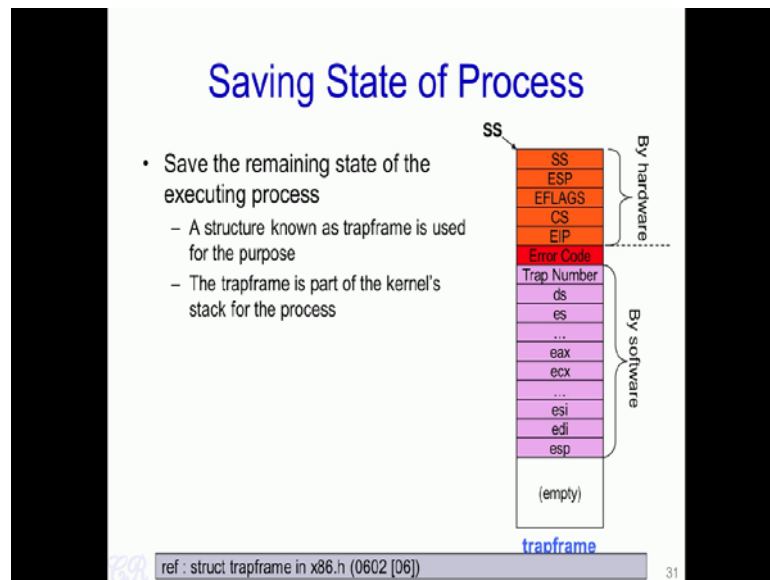
The slide is titled "5 Interrupt Handlers" in blue text. Below the title is a bulleted list under the heading "Typical Interrupt Handler". The list contains three items: "Save additional information (written in assembly)", "Process interrupt (communicate with I/O devices)", and "Restore CPU context and return (written in assembly)". The slide number "30" is visible in the bottom right corner.

- Typical Interrupt Handler
 - Save additional information (written in assembly)
 - Process interrupt (communicate with I/O devices)
 - Restore CPU context and return (written in assembly)

Let us see what a typical interrupt handler does. So, 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 the 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 interrupt is done then the CPU restores the original context and returns back to the user process. So, we have the first and third part written in assembly language that is typically written in assembly language that is this part as well as this part while the processing of the interrupt is typically written in higher level language like C.

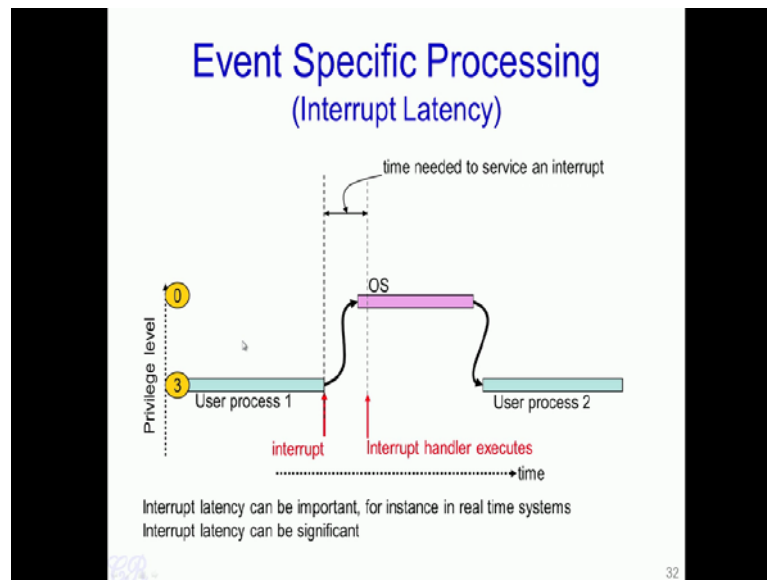
(Refer Slide Time: 08:27)



Let us see about the additional information that is stores. So, we have seen in an earlier slide 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 also stored on to the kernel stack. So, we have a part of this 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.

So, this additional part which done by the operating system will have no registered 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 registers like the ESI, EDI and ESP are stored. Or this pattern of registers which is stored on the kernel stack known as the Trap frame, and plays a crucial role during the return from the interrupt, in order that the user process which has been executing before could continue to execute from where it has stopped.

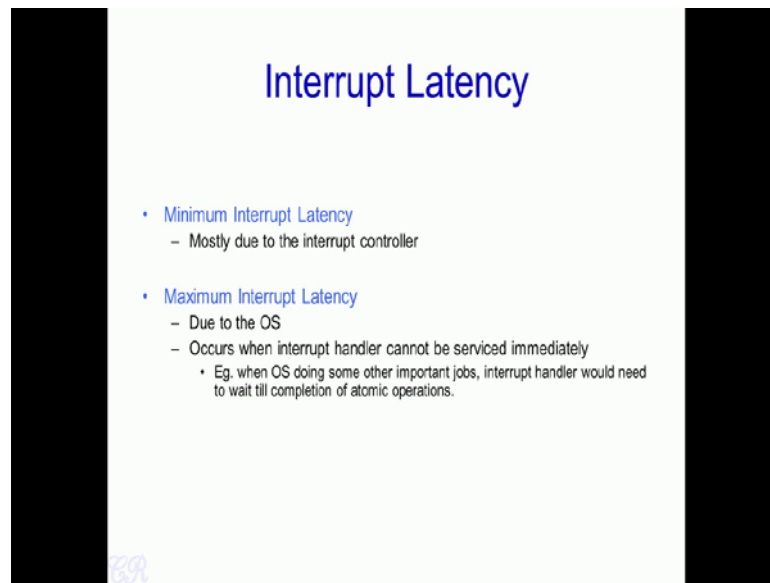
(Refer Slide Time: 09:48)



One important aspect when writing an interrupt handler is the Interrupt Latency. So, what is this Interrupt Latency? Let us say that we have a processor which is executing a user process that is user process 1 and after some time an interrupt occurs. So, when this interrupt occurs we know that there are series of tasks that happen between the CPU as well as the operating system. First, 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. 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 need a large latency for instance when an interrupt occurs in order to release the air bag which is present in 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 latency to be as small as possible. What affects this Interrupt Latency?

(Refer Slide Time: 11:25)



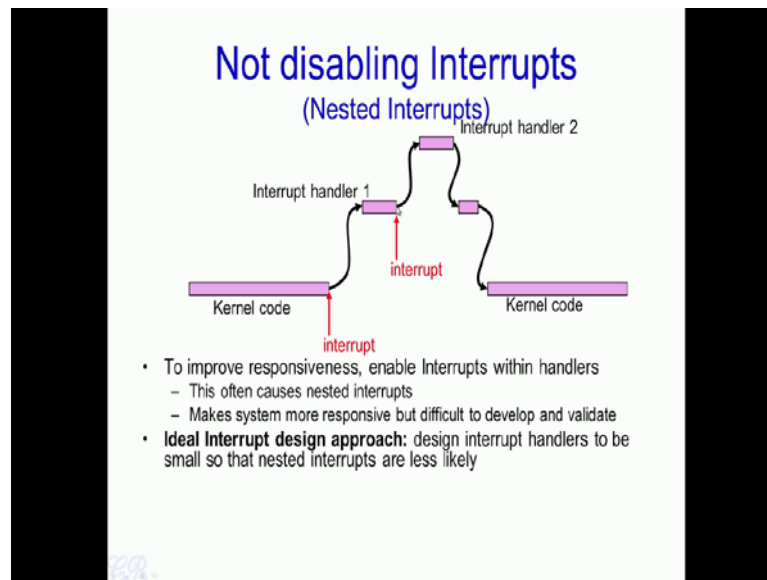
Interrupt Latency

- Minimum Interrupt Latency
 - Mostly due to the interrupt controller
- Maximum Interrupt Latency
 - Due to the OS
 - Occurs when interrupt handler cannot be serviced immediately
 - Eg. when OS doing some other important jobs, interrupt handler would need to wait till completion of atomic operations.

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 typically will 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 system for instance would disable interrupts when doing important jobs such as handling and other interrupts 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 operations completes. So, as a result you will get an increase in the interrupt latency.

(Refer Slide Time: 12:20)

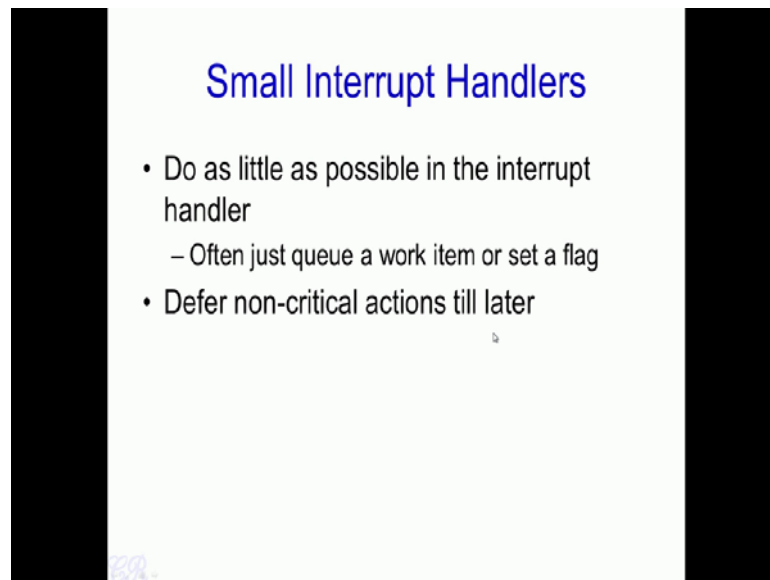


So, one way to reduce interrupt latency is by not disabling interrupts. However, this could result in what is known as nested interrupts and is depicted in this particular result. Let us say that the kernel code is executing in the CPU and after sometime an interrupt occurs due to which interrupt handler 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 occur that is it had stopped at this particular point and it will now continue executing from this particular point. So, after this interrupt handler finishes executing the original kernel code would 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 this nested interrupts makes designing the operating systems far more difficult, also validating this particular operating system will be more tedious. 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 in such a way that it would have completed its execution at this point itself, then there would be no need to actually nest the second interrupt.

(Refer Slide Time: 14:10)

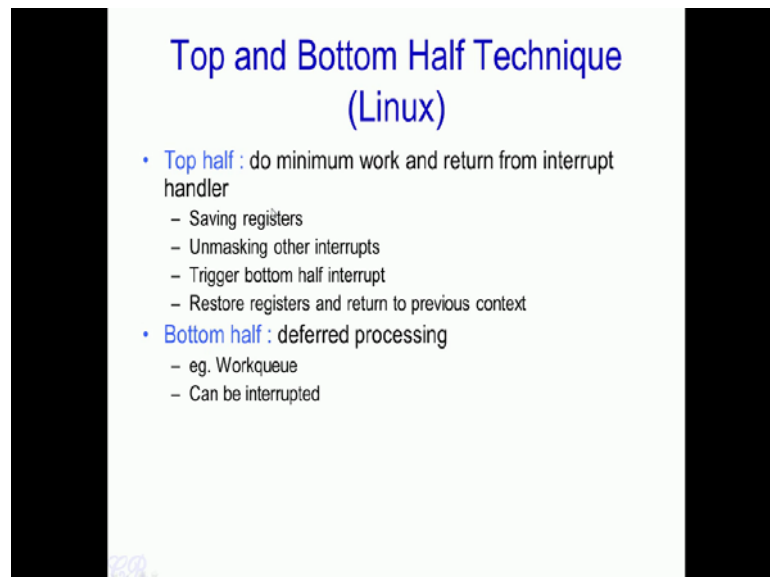


Small Interrupt Handlers

- Do as little as possible in the interrupt handler
 - Often just queue a work item or set a flag
- Defer non-critical actions till later

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 deferred too later.

(Refer Slide Time: 14:29)



Top and Bottom Half Technique (Linux)

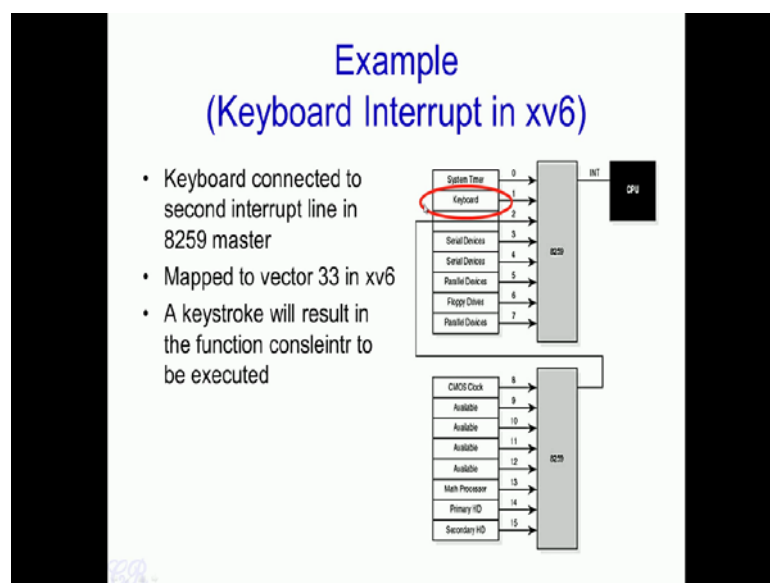
- **Top half** : do minimum work and return from interrupt handler
 - Saving registers
 - Unmasking other interrupts
 - Trigger bottom half interrupt
 - Restore registers and return to previous context
- **Bottom half** : deferred processing
 - eg. Workqueue
 - Can be interrupted

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. The

critical work involved is the saving of registers, unmasking of other interrupts, triggering the bottom half of interrupt handler to execute and restoring the register and returning to previous context.

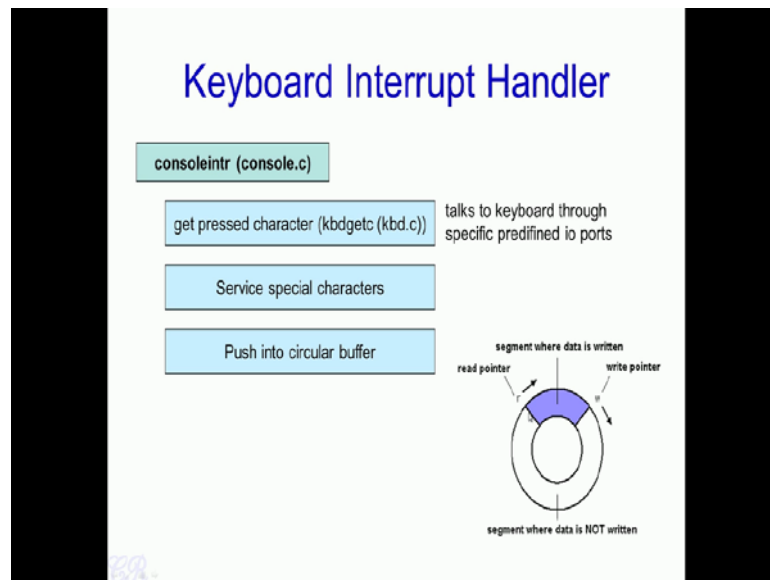
So, some time later the bottom half interrupt handler then executes. Bottom half interrupt handler would typically fetch some data which the top half interrupt handler has sent to it, through a say for instance a queue and it will process that particular data. So, unlike the top half interrupt handler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

(Refer Slide Time: 15:21)



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 and we have seen that the keyboard is connected to the IRQ 1 of the master 8259. So, when a key is pressed then it results in this particular line one being asserted and the master 8259 we then transfers 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 console INTR which is present in the console dot 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 dot c, to determine which key has been pressed. Now, if it was a special key then there was a special servicing done for special characters pressed in the keyboard and then the data is pushed into the circular buffer. The circular buffer is as shown over here - it has got a read pointer and a write pointer. The console INTR 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, we are able to determine what are the keystrokes we have been pressed.

So, this was the brief introduction to interrupts and how interrupt 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.