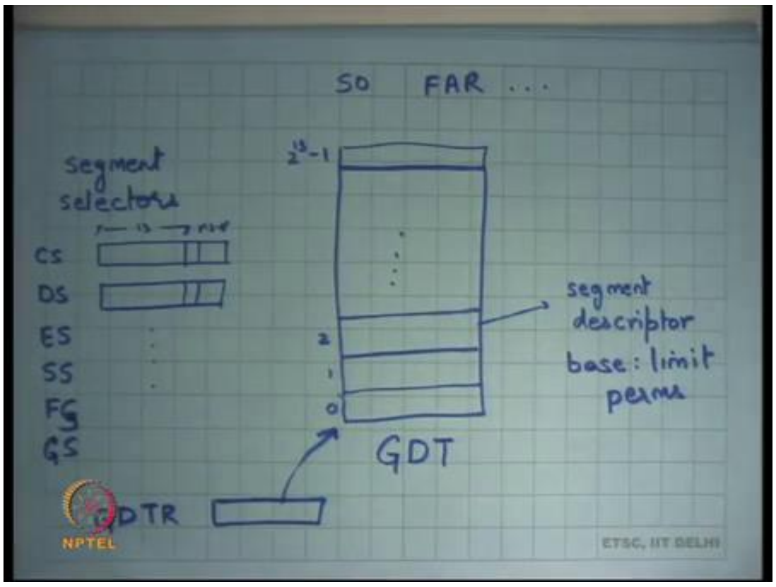
Operating Systems
Prof. Sorav Bansal
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture - 08
Traps, Trap Handlers

So, welcome to Operating Systems, lecture 8, right. So far, we have been looking at how an operating system implements the abstractions that it does. And the first abstraction we looked at was the address space, and we said look segmentation is one way to implement address spaces.

(Refer Slide Time: 00:46)



And there is something, there is a structure called a global descriptor table which lives in the physical memory. It is roughly of the size 2 to the power 13 which is perhaps the reason why it should live in the physical memory, right because the chip will not have that much capacity to be able to store large structures like this. So, structures like this which are relatively large need to be stored somewhere else and the typical plays are stored in is memory, right.

And then, but on the processor you have this register which is called the global descriptor table register which points to this GDT and that is how the hardware knows where to look for when it is actually try executing the MMU operations, right.

Now, these segments, these registers code segment, data segment, etcetera these are called the segment registers and within them they store the segment selectors, right. Depending on the instruction a virtual address will choose one of these registers. For example, if its if you are dereferencing the instruction pointer you will go through the CS register.

If you are dereferencing you know any regular data the default segment will be the data segment, so you will dereference a DS register. If you are dereferencing the stack through esp or ebp point registers, then the default segment will be SS registers. So, there are certain default segments.

And then you can also override the default segments by explicitly specifying that this is the segment register that I want to use for this particular address. In any case, the segment selector is used to index into the GDT. So, the algorithm inside the hardware is that will first dereference GDTR, it will first read GDTR to find the address of GDT.

You know, add the selector to that value to understand where exactly the descriptor lives, read the descriptor, get the corresponding base and limit values, perform the appropriate operation of pa is equal va plus base and checking a va is less than limit. And if these checks succeed it actually uses the computed physical address to index the physical memory, all right. Question.

Student: Sir, is the value of GDTR checked that every memory access instruction?

Is the value of GDTR read at every memory access instruction? That is a great question. In other words, for every memory access do I make another memory access, so does do as every memory access that a program makes does the hardware need to make two memory access, one to the GDT and then one to the real physical address.

Well, logically speaking yes, but actually no because you know these entries get cached inside the chip, right. So, there you know there are semantics on you know when the cache gets, when the caching takes place and when it gets invalidated and let us ignore that discussion for now.

But you know you can imagine that you know the descriptors for all the 6 segments that are present on the CPU they just get cached inside the CPU. So, you do not have to go over the bus every time to access the GDT descriptor, right. So, that is in it; that is an optimization, but let us look at the semantics for now.

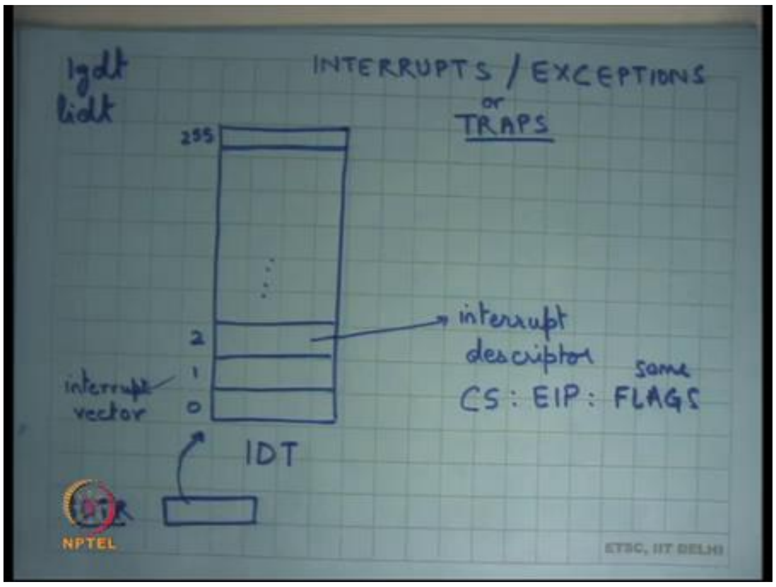
So, each segment descriptor has a base and limit and it also has permissions which basically says at which privilege level am I allowed to go through this segment, all right. So, the privilege level is determined by the lowest two bits of the CS register.

So, if the lowest two bits of the CS register are 0 which means I am executing in privileged mode, I can you know I can access any of the configured segment descriptors here or I can dereference to any other segment descriptors. On the other hand, if its 3 then I can only dereference segment descriptors that have permission set to 3, right or unprivileged, all right. So, all right so, far so good.

Basically, this what this allows you to do is every process will have a private address space, nobody else can touch it. The OS will have its own protected address space no process can touch it, and moreover each process has a uniform address space you know starting at 0 for example, right.

So, you do it the process the compiler or the linker does not need to be worried about where the process will actually get loaded. So, the loader and the linker can become completely independent in some sense, all right.

(Refer Slide Time: 05:00)



The next thing we said was look that is fine a process; this is how operating system implements address spaces for processes. But a process needs to do more in particular there needs to be a way for a process to make a system call, there needs to be a way for the OS to actually take control away from the process, on some external event like interrupt from an external device or an interrupt from a timer device for example, especially because I need to implement protection.

So, one process should not be able to run away with the CPU. So, I should be able to a run after every predefined time interval. Moreover, I should be able to get control if the process performs any illegal action like divide by 0, segmentation violation. So, the word segmentation fault has actually you know as historical roots in the segmentation procedure.

So, if a segmentation fault means that you violated the segmentation rules, right. So, you actually try to exceed the limit and so, you say violated the segmentation rules and that is how you know that is why it is probably called segmentation for, all right. So, for to facilitate this and also system calls there is a mechanism called the interrupt descriptor table or IDT as I am going to call it.

And the idea is that in case of an event which involved which is either an external event that never device needs extension it asserts a interrupt pin or it is an internal in event that the application actually executed something illegal or the exceptional condition then you know the processor is going to stop execution there for the process and look into the IDT to figure out where is the handler of this particular condition, right.

The condition could be an external device asserting something, but the condition could be internal, in either case there will be a number associated with that particular condition and that is that number is called the interrupt vector. So, the interrupt vector will determine which entry in this IDT should be dereferenced and that entry is going to be used to find the program counter of the hand law, right.

So, each descriptor in the interrupt descriptor table contains a pair compute code segment and EIP which is basically a pointer to the handler of that particular condition, right. So, for example, if you know if the network device and asserted the interrupt pin and you have assigned it number 2, then you know you should have the network device handler at this particular address or if you if there is segmentation violation then you should have the segmentation fault handler at this location and all.

And these handlers will typically do what? They will either you know execute the device driver logic in case it is an external interrupt or if it is an exceptional condition, they will execute the appropriate logic to deal with that exceptional condition. So, for example, if I was in segmentation fault the operating system could say they just kill the process, all right or it could convert that exception into a signal and pass it on as a signal to the Unix process, right.

And recall what a signal is. A signal is nothing but interrupting the process execution and making a call to the signal handler of the process, right. So, in some sense the abstraction of Unix signals is very much inspired from what happens at the hardware level in terms of the exceptions and interrupts, right.

So, an interrupt also causes a handler to get executed and the signal got, but in this case it is a hard it is an interrupt handler, in case of a process it was a signal handler which was the process have registered. The process was able to resist a signal handler for itself similarly the operating system should be able to register interrupt handlers or exception handlers for itself.

The mechanism to be able to register signal handlers is provided by the OS. The mechanism to register interrupt handlers or exception handlers as provided by the hardware, right ok, all right. So, we understand handlers so far.

The other important thing is that the code segment could actually be a privileged code segment. So, even if I am executing in unprivileged mode, if this code segment has last two bits set to privileged which is 0 then you know when the interrupt is going to occur it is going to start executing in privileged mode and that is required because you want that you know whatever code you are going to have for device drivers or for exceptional conditional handling should run in privilege mode, right.

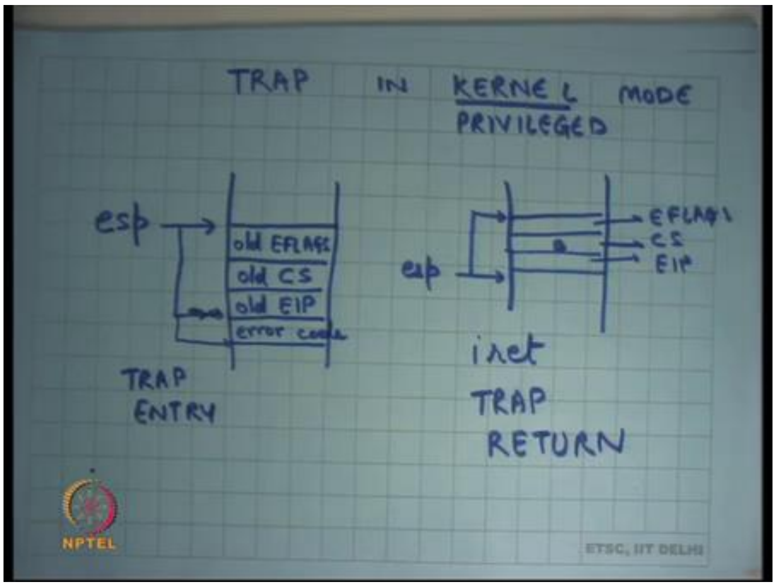
So, the hardware designers provide that facility because by allowing you to specify any CS cell, right, ok, all right. So, with that let us look at what happens if you know if. So, before I start an interrupt or an exception, so I mean I use the word interrupt for anything that is from an external source like external device like a timer or a disk or whatever and I use the term exceptions for something internal.

Like a segmentation fault or divide by 0 etcetera or an illegal instruction was executed etcetera, right. So, an illegal instruction is executed. One example would be a process try to execute a privilege instruction like lgdt, right. So, we saw that loading the GDTR is a privilege instruction, if you are running in unprivileged mode and the process tries to execute the privilege instruction lgdt, right.

You recall the lgdt instruction which loads the GDTR, you going to do the process is not going to be allowed to do that and what will happen is an exception will get raised and the operating system will come into action and it will decide what to do with the process, right, all right.

And another thing the IDT itself is also stored in memory. Once again IDT is a relatively large structure cannot be stored on chip completely. So, rather the IDT is stored in memory and there is a pointer called IDTR inside the chip which basically says where to look for IDT just like GDT, all right.

(Refer Slide Time: 11:14)



So, now let us say I get a trap. So, I am going to, I am going to use the word trap as a general term for interrupts or exceptions. So, I could get an interrupt I could get an exception let us just call them a trap. This is the question. Can we change the entry in the

IDT? Let us hold that thought for a moment that is a great question but let us just first understand how this works and then we will talk about security.

You know what can a process be able to do and what it should not be able to do such that you know it is not able to gain control of the system. So, let us say I get a trap in kernel mode, kernel or you know you can also say privilege mode, right. So, I am going to use privilege, unprivileged or kernel user same thing, right. So, kernel is privileged mode, kernel mode is privilege mode and user mode is unprivileged mode.

So, what happens is let us say I was executing, and a trap occurred. So, what will happen is let us say this was my stack pointer this was the value of my stack pointer. As soon as a trap occurs what I am going to do is I am going to push the old values of CS and old value of EIP and I am going to set my CS and EIP registers to the values which I get from the interrupt descriptor table and I am going to now start executing, right.

So, I got the trap in the kernel mode, I look I used the esp value at that time to push the old values on the stack and now I use I start excluding the interrupt handler, right. This is similar to how signal handling was happening, right. A signal comes you just execute that as a function call as an asynchronous function call.

Similarly, this is doing the similar thing. You know an interrupt game, or a trap happened you just executed the handler as an asynchronous function call using the same stack that the kernel was using, right ok. Another matter of detail apart from CS and EIP it also stores old EFLAGS, right. So, the interrupt descriptor table entry could also potentially contain flags resistor or certain flags and when the interrupt gets actually starts to run the semantics are that you are going to replace not just CS and EIP, you are also going to replace certain flags, some flags, fine.

So, before you start execution of the interrupt handler at this point, we are also going to replace some flags with the values that are present in a descriptor. And because you are going to change things you also need to save them, and once again you save them on the stack, all right. So, this is clear.

Student: Sir, this is stored in the kernel stack?

This is so, we are talking about a trap received in kernel mode. So, let us say does this is an interrupt while I was executing in privileged mode, this is what will happen. So, it will just push things on whatever the value of esp is currently, ok. Don’t we require storing general purpose registers?

Yes, we do require saving more things, but this is the minimum that the hardware saves for you and now the control has been transferred to the handler and most likely if the first thing the handler is going to do is save the rest of the state. It may you.

Student: Sir, is a part of OS only?

The handler is yes part of the OS.

Student: But this transfer take place by a hardware.

By a hardware, yes. See basically the hardware has to change the values of some registers to be able to execute the transfer and so, whichever registers it changes, whichever is the minimum set of registers it needs to change those are the set of registers it saves. Other things it does not need to change. It is really up to the software item whether he needs to save them or not, right.

These kind of things the software writer would not have been able to save, right because you have to say where to execute the handler and so, before that I mean it is a chicken and egg problem. So, the hardware has to come in into the middle and says, these are the things I will save and the rest of things you can say, right.

So, there are certain flags I am going to look see later which need to be saved by hardware. So, cannot E, why cannot EFLAGS we also saved in software, right. So, there is a reason for that, and we are going to discuss that later. It is just, let us just leave it for now, all right.

So, this is what happens on a trap entry, right, right, and then you know the handler is going to execute. Typically, the handler is going to execute on the same stacks. So, it is not going to change the stack value, right. So, just like an asynchronous function call, you just execute the handler and then you want to return from the handler, right.

In case of a signal, when we saw for Unix I just executed the return instruction and I will get back to where I was, right because the semantics of the signal entry was that the OS used to push the return address on the stack. And so, when the signal handler used to call return, I used to pop the return address on the stack and get back to where I was, right.

In this case, also it is going to be similar except that it is not a regular function call, so I cannot just use return to return, because return is only the semantics of with the return instruction is that I just pop the stack and look at the EIP and that is it, right. I just pop the EIP and said that. It has nothing to do with CS and EFLAGS, recall it the return instruction.

So, we need a special instruction call interrupt return iret, in trap return, you want to call it trap return, all right. And here there is if the esp is pointing at this place and interrupt return is going to pop the stack and put this into put the first value into EIP register, the second value into CS register and the third value into EFLAGS register, right.

So, this is the semantics of the iret instruction. What is going to do is its going to look at the current stack pointer and pop these, pop the first 3 words on the stack and fill these registers with those values, all right. So, this is interrupt from, return from interrupt.

Student: Sir.

Yes.

Student: Yes, (Refer Time: 17:51).

Yes. So, right now I am talking about a trap in the kernel mode, right. So, because I was executing in the kernel mode already.

Student: Esp is of kernel.

So, esp is of the kernel, all right. So, I am already done executing in the kernel mode. So, esp is trusted in this case. So, I can do these things. Next, I am going to talk about traps in the user one that was the discussion that we have we were having yesterday, right.

As a matter of additional detail there are some vectors, some interrupt vectors which actually push 4 words, all right. This is just a matter of; just a matter of the fact really there is no there is no fundamental behind it, but there are certain vectors for their which there are certain extra, this one extra value that is pushed that is the error code, right. So, for example, the page fault. For example, you can say you know a certain exceptional condition could additionally push this error code for the handler to know exactly why that exception occurred, right.

So, certain exceptions can occur because of various reasons and so an extra value is pushed on to the stack to indicate that, right. So, the only the hardware knows exactly why this exception occurred. So, one exception could actually be representing multiple conditions and so error code basically tells you which of these conditions is actually the reason for that exception.

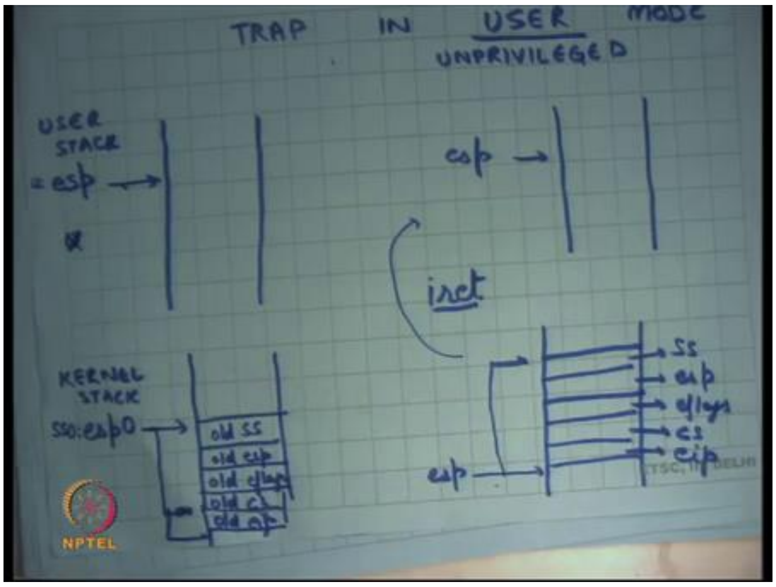
What this means is because some vectors push the error code and others do not. The handlers need to be appropriately set up such that they understand this, right. So, the handler for the vectors which push the error code will be set up to know that an error code is already there, and the handlers for which the vectors which do not push the error code to know that it is not there.

Just to simplify things typically on x86 and OS handler will all the handlers for which the error code is not pushed the handler will the first thing it will do is just push a 0 value there, so that this stack becomes uniform, right. So, it can assume a uniform stack frame really, right. So, if there was no error code pushed by that vector the OS in software will just push a 0 for that particular, in that particular handler.

Student: So?

On return the iret instruction does not assume the presence of error code, all right irrespective of what happened here. So, it is a responsibility of software to pop the error code before it causes the iret, ok, all right, ok.

(Refer Slide Time: 20:23)



Now, let us look at a trap in user mode, all right or let us say unprivileged mode, ok. Once again, I was executing and here is my stack pointer that is where it is pointing, and a trap occurred. Now, what should the kernel do? Can it do the same thing? Can it just push; so, clearly it needs to override certain registers for example, it needs to override CS and EIP definitely, it also needs to be override EFLAGS, ok.

So, now question is where it pushes the old values of these registers because it needs they might interrupt return. So, where does it push them? It wants to push them on stack. But can I just push them on this stack?

Student: No, sir.

Why? Because this particular value of this register is modifiable by the user and the user could set it to anything which cannot be trusted, right. I do not trust. So, the model is that I do not trust the process. I do not trust the process in the sense that I would not let the process bring down the whole system no matter what the process does, it should never be allowed to bring down the whole system.

In this case, if the process had just set esp to 0 for example, right or some you know some invalid address and then, right. So, let us say esp was set to 0 and then it executed some exceptional condition which caused the trap. Then what will happen is the kernel will try to push on a full stack, right, 0 cannot be decremented any further and so, it will get in to an infinite exceptional condition and this is the CPU will actually halt, right. So, that is not; that is not acceptable.

So, what I am going to, so what is really needed is that on a if you receive a trap in user mode you should also switch the stack pointer before you start pushing things, right. So, so what happens is on a trap it actually switches to another stack and that stack is let us say esp 0 and then on that stack it is going to start pushing things, right. Notice that because I am modifying more registers now, unlike the previous case where I was just modifying a CS and EIP I am now also modifying esp.

In fact, I am also modifying SS, right. So, the semantics are that I am going to modify the entire virtual address which is represented by SS colon esp. So, initially if it was SS colon esp now it becomes SS 0 colon esp 0 and so, I need to save the old values of SS and esp and I am going to do that on this stack, right. So, what I am going to do is I am going to push 5 bytes here and I am going to say let us say old SS, old esp, old EFLAGS, old CS and old eip, right

So, these are the things that the hardware saves for you and now the interrupt handler is set up to be able to run in a secure environment. So, it has a secure stack that I did not trust, and it has it is on the, right instruction pointer and now it can start running and it may want to save more things. And typically, what it will also save those things on the same stack on which it was started because this is trusted stack anyways, ok.

Student: Sir, second one is a kernel stack.

The second one is the kernel stack. This was the user stack, right. Now, what happens if when the kernel executes the iret instruction? So, in the now the interrupt handler will execute, or the trap handler will execute in the on the kernel stack and eventually it will want to return. And it should actually return to exactly the same point in the user space we had actually left off, right.

So, once again it is going to call the iret instruction and the semantics of the iret instruction is going to once again, but you know let us say this was the esp at this point. It is going to pop off the first 3 words and set up set those up as CS EIP and EFLAGS just like before.

But now it is going to see that oh the CS is actually an unprivileged CS, right. So, the CS that it is actually popping into the; into the register is actually an unprivileged CS. It can see that from it is value and so, it is going to realize that because it because I transition from a privileged to an unprivileged, I am transitioning back from and privileged to unprivileged mode there must have been two more words that have must have been pushed and so, it also pops off those two words and actually pops off 5 words to set to basically reload these registers back again, all right.

So, in the previous exam case the iret was just popping 3 values in this case iret is popping 5 values. How does iret know how many values to pop? By looking at the value that was popped in the first 3 words, right or by looking at the value of the second word, right that contains the CS, all right.

And so, when you execute the iret instruction you are actually going to get back to the user stack, right because you have changed the value of SS and esp, so you are actually now going to start executing in the user stack with the exactly at exactly the same EIP at which you left off, ok. So, now let us talk about security. So, what prevents a user from being able to take control of the system?

So, we said a user can in basically sandboxed within his own address space because firstly, he cannot modify GDTR; secondly, we said the GDT itself should live in a portion of memory that is not accessible by the user, right and thirdly these the values of these selectors can only be set to one of the values that have been put in the GDT, right and the OS should be careful that it only puts the ah, right values or permissible values in the GDT, right. So, that is how I was ensuring that a user is not able to jump out of its address space question.

Student: Previous one, how do we interpret in the kernel mode that esp never becomes 0?

Good question. So, how do we ensured that the curve in the kernel mode esp never becomes 0 or esp never under flows, right or I had never done out of stack? So, this is the, this is something that an OS designer has to be careful about. Nothing in this world is, nothing is infinite, right, infinite. So, even in the user mode a stack is never infinite. It is just an illusion of infinity, right.

So, if you ever try to cross your boundaries you are going to if a process ever crosses its boundaries, it is going to get a segmentation fault, ok. And OS is should ideally never cross its boundaries, so the OS designer should write its OS in such a way that there is a bound to the maximum stack, stack length that you can have, all right. I am going to see how that is done.

So, an OS designer or OS writer or a kernel developer has an extra bound that you know you cannot grow the stack too much, there is a maximum bound to how much is stack can be. Typically, a stack would be for example, the Linux kernel has a stack of one pay of around 4000 bytes or maybe even 8 thousand bytes, right. So, between 4000 to 8000 bytes are enough in general for the stack, right.

Even your xv 6 kernel which we are going to look at an academic kernel called xv 6 (Refer Time: 28:33), it also has you know 4000, 4096 bytes of stack and that is enough for of course, ok, all right, ok. So, we saw how; this is basically saying that a program can never jump out of its address space. It is also saying that a program can never execute an instruction which is privileged because I am going to run it in the privileged mode etcetera, ok.

And the program can never lower its privilege level. So, once you have set up the privilege of the CS register, I cannot just lower their privilege level. But we also saw that the IDT is a way for the process to actually lower its privilege level, right. So, for example, I can just execute some exceptional condition and I will now be executing in privileged mode.

So, the first thing is that the OS should ensure that all these entries all the handlers of the IDT are appointing, all the entries in the IDT are pointing to valid values. If one of these IDT values is pointing to some garbage then an OS can actually cause that particular vector to get fired and you are going to actually try to execute some in invalid instruction and the system can get down, right.

So, the first thing is the IDT itself should have completely same values. The second thing is the instruction to load the IDT which is the lidt instruction, which just load the IDTR should be a privileged instruction, right a user should not be able to just say lidt and because of the user can say lidt then you can just take control of what gets executed in privilege mode. So, the instruction lidt should be a privilege instruction, all right.

The third thing is that IDT itself should live in the OS address space, it should not be able to, it should not be visible to the for any process, ok. So, only the OS can set up these values. The OS needs to be very careful about settings of these values, so the user cannot take advantage of any bugs in the OS plus you basically ensure that this structure is not modifiable by the process, ok, all right, ok.

Student: Sir.

Yes, question, ok. Is it possible that a process wants to handle an interrupt in a different way than another process? It is a matter of what abstractions the OS is providing to the process. In the abstraction that we have seen so far that is not possible. A process has no idea what an interrupt means, right. It only understands system calls and signals, ok. So, it does not make sense to say that a one process should be able to convert and to control the handler of a particular interrupt, right.

The process can control the handling of a particular signal, right. So, that is the difference. Of course, you know; so, what you are, what you are put is you put a layer in between the hardware and the process and you have said that this is these are the permitted things that you can do and I am going to implement those things, right.

And we have seen one type of abstractions that are the Unix abstractions, and then there are other types of abstractions which actually allow the thing kind of thing that you are saying, all right and they are performance advantages to be able to do that, right, but it also makes things more complex and etcetera and you are going to look at those trade offs later on, ok, all right.

So, in this figure when I said that when our trap is received in user mode I actually switch the stack to as a 0 and esp 0. There is a minor matter of detail where does the hardware or get these values, SS 0 and esp 0 from, right. So, there needs to be some way of telling the processor, look these are the values of SS 0 and esp 0, before you actually before the OS actually gets control to the user he should set up SS 0 and esp 0 appropriately, so that if a trap occurs while the process was executing the hardware knows that this is what you should load, right.

(Refer Slide Time: 33:01)

TASK STATE
SEGMENT

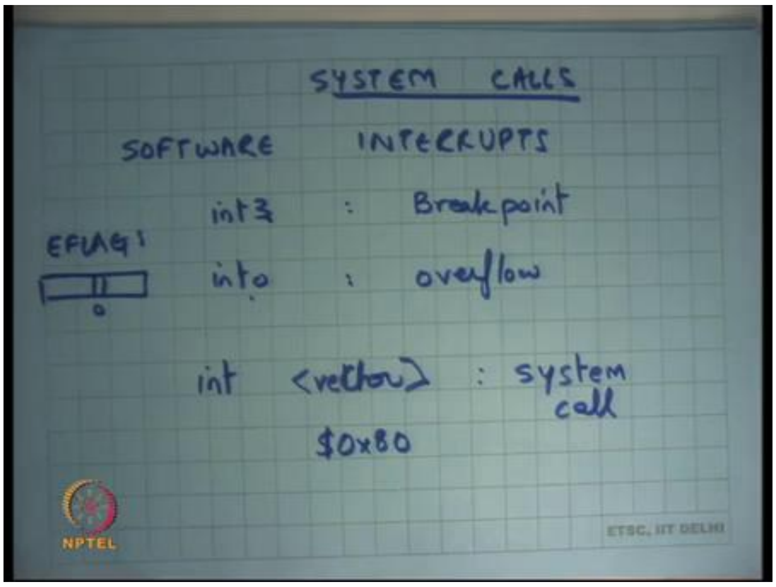
And on x86, there is a structure called task state segment which allows you to do this and they have more detail they are going to find out in your programming assignment. I am not going to go into detail, let us just for the purposes of the of our discussion let us just assume that there is some place where you can store these values and tell the hardware that here is where you should pick up these values from, right when you get an interrupt in user mode, ok.

Another thing I would point out is that notice that when I am talking about the semantics of instructions like iret, while the iret the semantics have been designed in such a way that they complement or the bracket the operation at interrupt entry or they you know, so whatever the interrupt entry is doing iret is undoing. But actually, iret exist as a separate instruction in itself, right which has these semantics that is going to pop off the first 3 words and load them into these registers and depending on its values and maybe pop of two more words etcetera.

So, actually between the execution of the entry between the entry and the return the interrupt handler could potentially modify these values, right for whatever reason. Typically, it will not modify these values, but sometimes it may, right. One example where it may modify these values is for example, when you want to set up the first process or let us say when you implement the fork system call, right. So, what happens in a fork system call?.

Before we discuss that let us also talk about how system calls are implemented. We discussed that lesson yesterday, but let us just review that.

(Refer Slide Time: 34:38)



So, apart from interrupts and exceptions there are also something called software interrupts. These are interrupts that the program can actually invoke. So, instead of some exceptional condition happening or some external device saying that I want an interrupt to getting handler to get executed.

The program itself could execute an instruction like int 3, it is a one byte instruction or int o which basically says invoke this particular interrupt or trap. The semantics is that this is going to simulate an interrupt or vector 3, right. This is going to simulate an interrupt of vector whatever o stands for, overflow in this case, right the semantics for this is that this is basically used for debugging. So, this is the breakpoint, right.

So, if you have wondered how GDB works for example, one you know it basically; so, if you put a breakpoint at some point you say I want to stop the execution at this particular point, right at this particular value of the instruction pointer, what GDB does one way to do that is basically that GDB writes this particular instruction int 3 on that particular byte.

So, let us say I wanted to get interrupted at EIP 1000, what GDB going to do is it is going to write int 3 instruction at 1000. It is going to replace the original contents of that

EIP of that memory location and put int 3 there. What will happen is when then program gets executed as soon as it reaches that point it is going to execute int 3 and an interrupt going to get simulated. The interrupt handler will be the OS interrupt handler and in this case the interrupt handler will know that this is a breakpoint interrupt handler.

The breakpoint interrupt handler let us say what it does is it converts the exception into a signal that it gives to the GDB process and so, the GDB is installed a signal handler which basically says stop execution and return back to the user with a prompt and ask for the for the next command from the user, right. So, this is one way of implementing breakpoints, right. So, a good example of why software interrupts are used, ok.

Similarly, in interrupt overflow is an overflow condition. So, this basically says that if now if in the EFLAGS register the overflow bit is set then cause an interrupt. So, the way the hardware designers imagined this to be used is basically you perform some computation and then you execute the int instruction. So, if the execution actually created an overflow it will cause an interrupt. If it did not cause an airflow overflow it will not cause an interrupt.

And so, the idea is in the common case when there was no overflow you will just you know very quickly go to the next instruction, you do not have to have if then else kind of logic in your code. So, this makes gives you a nice, very fast way of doing this kind of exceptional condition handling, all right. And of course, then there is the normal int instruction which can take any vector number, and this can basically say simulate this particular interrupt, right. And this is what we use for system calls.

So, example the particular vector number let us say you know the Linux kernel had been using the number 128 or hexadecimal 80, to do the system call. So, basically means if a process makes int dollar 0x80, it is going to simulate an interrupt at vector 0x80 and the handler at 0x80 is going to assume that a system call was made.

It is going to also assume that the arguments of the system call and the system call name itself is stored in certain places. For example, it is stored in the registers. And so, it is going to read the value of the register to figure out what the argument what system call I need to execute for example, exac fork etcetera and what are the arguments for to it, right. For example, this the address of the string for exec etcetera, ok.

So, system calls are also implemented by using this interrupt descriptor table structure and, ok. And so, and so, we were discussing how iret can be used. So, we said that iret can actually iret has us these semantics and handler may want to actually change these values before the interrupt runs and one example where you would do that is the fork system call, right. What happens on a fork system call? You make a system call; the handler is a system called handler it figures out that you are trying to call fork.

What it does is it creates a new address space and copies your address space into the new address space. One way to create a new address spaces get that much get the same amount of memory in physical memory, right, copy all the contents, set up a new base and limit in your internal data structures and set up the stack in this way such that EIP, CS, EFLAGS, ESP, SS is identical to how it was in the process which called the fork with the only difference being that the return value should be different, right. So, how does it do that? Where is the return value stored?

Student: ES.

Student: ES.

Let us say it is the return value of the system call is in the ES system, so it just changes the value of the ES register and executes the iret instruction. So, it is a set of new stacks, it copies these values, it maybe changes some register and causes iret, right.

Another way you know a fork system call could potentially return is basically you know store the value on stack, right. So, let us say the return value is coming on stack even that is possible. What the handler will need to do in the new process is dereference the stack pointer of the user and maybe change some values there before calling iret, all right. And so, the stack will actually see new values, slightly different values let us say, ok.

Similarly, you know when the kernel boots there is really no process, there is really no user mode, everything is run running in privileged mode. So, when you create to create the first process you know for example, the init process on Linux you know what the kernel will do is it will just set up the stack in a certain way, it will set up the address space in a certain way such that you know a certain executable is loaded in the address space and now we will just call the iret instruction, and so, now, the first instruction of init starts executing.

So, even though in it actually never existed before the iret, iret is basically simulating as though I am returning back from a system call that made, right. So, another I mean basically a kernel can set up a fresh stack and still call iret or it can modify these values and still call iret to a to implement desired functionality.

Student: Sir, when while coating cannot we just move the values into the CS and EIP to move instruction instead of first coating with stack (Refer Time: 41:54) iret?

Cannot it just move? So, you cannot just move into CS, right, there is no instruction that says move into CS, there is no instruction which says move into EIP. There has instructions like l jump.

Student: L l jump.

Right. So, that is an interesting question. Can you just say l jump to this particular value and let us see why is that not allowed? So, you also need to change the stack, so you will basically need to load the stack from the user mode to the kernel mode and then you are going to call l jump.

Student: Sir.

I believe it is not allowed in the processor. So, you cannot just say l jump from one privilege level to another privilege level, right, but let me confirm that. So, exactly why the x 86 architecture does not allow you to adjust to l jump. But more importantly even this is an atomic operation, right. So, you can just basically set up the stack and the EIP in one go, as opposed to being it we are doing it in multiple instructions.

So, first you will have to load the stack and after you have loaded the stack you are still executing in privileged mode and so, that has also has its own security implications you know executing on an unprivileged stack in privileged mode it has its own problems.

Student: So, we want to need to agreeing to return to both the process (Refer Time: 43:15) parent and the child.

Right. How do we do that? That is easy. The parent can just return. So, for the parent it was just a system call, a regular system call, right. So, just how just like it returns oh my regular system call is going to call iret and is going to return from it. For the child you need to create a fresh stack, right.

So, here is the process, it made a system call, write the system call basically internally figured out its a fork and so, it created new state and create a new stack, it added a new process to its list of processes and now it just calls iret on the original stack, right. So, the parent can just continue as it is, right. And now the child on the other hand will get will continue in on its new stack.

Student: So, we do not iret on the child.

We call iret on both, right. So, only one process enters the kernel and two processes exit the kernel.

Student: Need data.

And the stack.

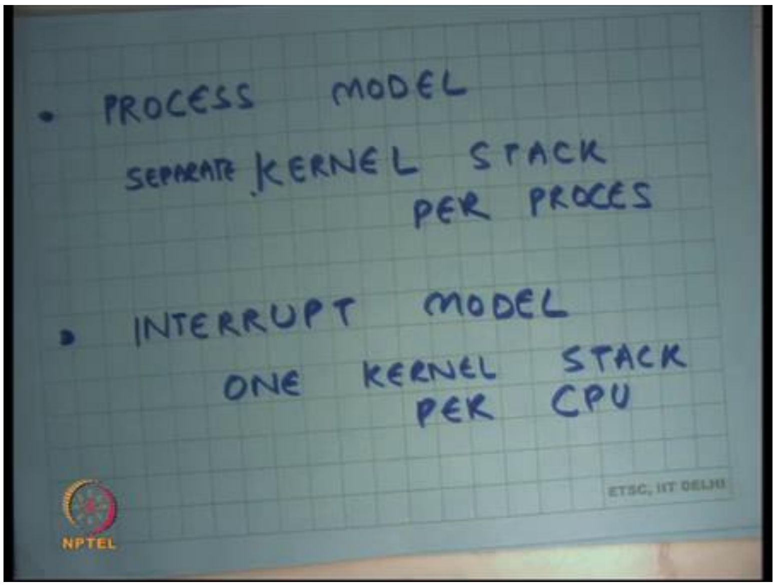
Student: In the stack.

So, we are copying the entire address space and we are setting up the stack pointer to be identical, so that basically means you are copying the stack also.

Student: But there is only one kernel stack, right.

I see. So, what basically, ok. So, now, you are now you are asking how many kernel stacks there are, all right. So, that is your question, good, ok.

(Refer Slide Time: 44:46)



- PROCESS MODEL

  SEPARATE KERNEL STACK PER PROCES

- INTERRUPT MODEL

  ONE KERNEL STACK PER CPU

NPTEL
ETSC, IIT DELHI

So, how so, there are two ways of kernel as implemented one is the process model, where there is a kernel stack, separate kernel stack per process, all right. So, in this case, what will you do on a fork? You will create a new kernel stack and you will copy the entire kernel stack, so apart from the user address space which has nothing to do with a kernel stack you wait and so, you going to have a separate process a kernel stack per process.

So, when you fork you going to actually have two stacks and you are going to call iret on each stack independently. So, that is how you implement one entry and two exits, right. There is another model in which you can implement things which is called the interrupt model. So, this is basically you know how you implement your kernel. In the interrupt model there is just one kernel stack per CPU. So, let us say there is just one CPU in your system then there is just one kernel stack.

And so now, what the kernel needs to do is that it needs to store; so what will happen is that there is just one kernel stack and that is the value that goes into SS 0 and esp 0 that the hardware knows and that is permanent and so, whenever you enter you enter on that stack. But you are going to when you switch, so let us say you know when I was let us say a function process make a fork system call and I was executing on the stack and now I created a new process called child and now I wanted to switch to the child.

So, what I will do is I will save all the contents of the kernel stack in some other data structure and load the content of the child's stack from his data structure into the stack, right. So, eventually the same thing you basically need separate states in the kernel which simulates a stack per process, right. In this case, you are having you are actually having a separate kernel stack per process, in this case you actually have only one stack which is visible to the hardware, but internally you are swapping state to basically fill that stack, right.

In this case, you will actually tell the hardware, so on each context switch you are going to tell the hardware that this is the this is stack, this is the value of SS 0 and esp 0 you should use, right on every context switch you change the hardware structure. In this case on every context switch you do not change the hardware structure you just do this internally. One of them, we only have one CPU let us assume, and then.

Student: Sir. So, we cannot call them parallely before the iret?

So, you cannot let us say there is just one CPU. So, what will happen is one CPU, one process made the fork call, it created a new process, it added it to the list of processes that are possible to run, right and now let us say the parent continues to run. So, parent is going to call iret and now parent can continue to run and then let us say the parent says I want to get out of the CPU or let us say a timer interrupt occurred and the OS forcibly brings him out of the CPU and now this process's turn comes and so, now, he gets to run and now he will call iret.

Student: But at that point already popped out all the as the (Refer Time: 48:03).

No, we popped out things from the parent stack. So, we have a separate stack per process, right. So, we popped out things from the parent stack the child stack still remains.

Student: Sir, in the interrupt model what it is exactly (Refer Time: 48:15) as once you have popped of the parent child?

In the interrupt model you know it is basically the same thing except I mean you basically. So, storing the value with which the stack needs to be initialized when it gets context switched.

Student: And we are storing in?

In some data structure, right. So, let us say in that process, in the list of processes you also saying you know this is a these are the values which you should initialize the stack before you start it running.

Student: (Refer Time: 48:38).

Ok.

Student: So, (Refer Time: 48:40).

So, I mean the there is no fundamental difference really, in one case you are exposing it to the hardware and the other case you are keeping things internally, all right, ok, all right. So, question.

Student: Sir, what will happen in case of (Refer Time: 48:57) parent and child?

Right. So, what will happen? So, this seems to be confusing people that what happens in case of one kernel stack. So, let us say, this is one kernel stack, all right, but each time you context switch you are going to reinitialize that kernel stack with certain values, right and so, for in case of parent and child you store that you know this is what you should be initialize it before you start running it, ok. So, that is the difference.

Student: (Refer Time: 49: 26) then where is that stored, parents where are these values that (Refer Time: 49:30) actually?

In the first, clearly in the OS address space, so where are these values stored? Clearly in the OS address space how they are stored, in some data structure which is maintaining the set of all the processes and associated information, right. In any case, when I have a separate kernel stack or process, I am also storing the stack with the process, right.

So, this is the this is your stack, this my stack, this is his stack etcetera and so, I am going to load the esp with that stack. In this case the esp remains the same I just initialize the memory locations that is the only difference, right.

Student: Sir, iret possible that we can copy all these 5 contents who some (Refer Time: 50:07) twice the copy of the same?

Is it possible to make two copies of the same thing on the same stack?

Student: Yes, sir.

I mean for intentionally or unintentionally?

Student: Intentionally.

And then why?

Student: Then, first time when we called iret for either child or parent then we get the first time (Refer Time: 50:31) second time when we call the second time.

No, it has a lot of problems. So, the question is can we use the same stack for both the parent and the child. And actually, I have two frames one for the child on the parent and use the same stack and not having two contexts switch. There are also other processes in the system, right.

So, you know I mean the cleaner way would be that you basically say that you know this is yours, this is how you should initialize things. So, other it is possible that between the parent gets to run and between the child gets to run the other process that get to run in the middle. So, you know why you would want to do that, ok, all right.

So, your programming assignment, you are now roughly ready to actually start on your programming assignment and you should start on it immediately, ok, all right, ok. So, let us stop.