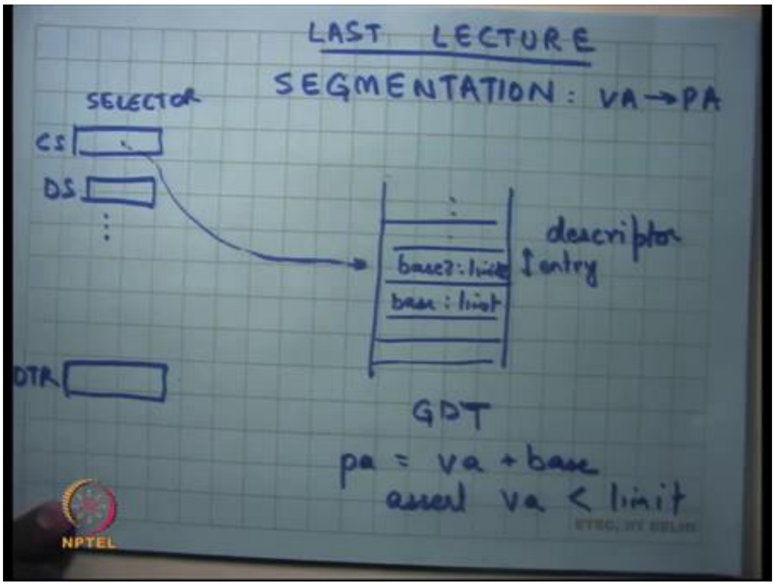
Operating Systems
Prof. Sorav Bansal
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture - 07
Segmentation, Trap Handling

(Refer Slide Time: 00:23)



In the last lecture, we were discussing Segmentation. Segmentation is a way to implement process private address spaces, and to divide physical memory into virtual address spaces for the processes right. So, we said there are you know in some way segmentation provides mapping from virtual address to a physical address right.

A virtual address is what the process is, and the physical address is what actually goes on to the via to the memory right. And so, you know we said that the there is some logic on the hardware of the processor, which we call the Memory Management Unit or MMU which does this translation. And the particular mechanism that we are looking at right now is segmentation all right.

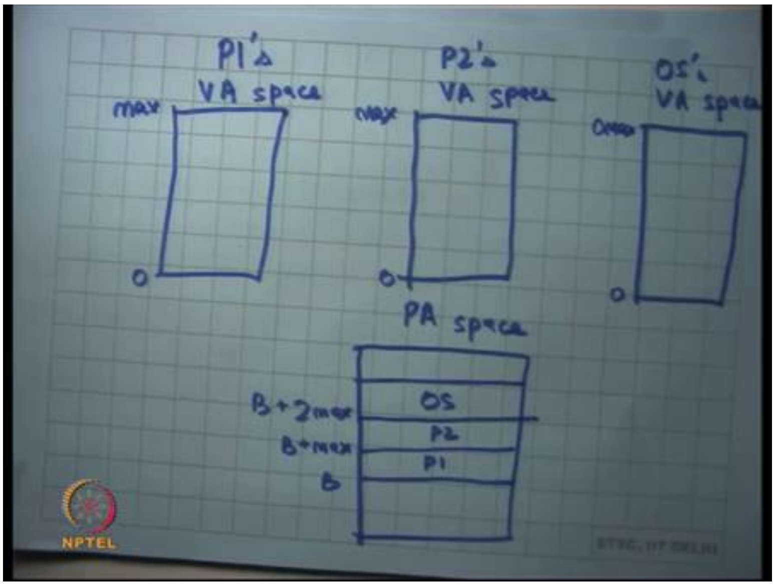
So, this is how segmentation works. There are multiple segment registers. Every memory address or every virtual address that an application specifies which could be a specified using direct, indirect, or displaced modes which we have already seen in the instruction encodings. Whatever address there is it has to be prefixed with a segment id, so that address that is computed by an instruction is actually the offset inside the segment that is specified by the segment id, and the segment id would be CS, DS, CS whatever right.

The segment gives you a pointer into a structure called the global descriptor table which also lives in memory, right. The address of the global descriptor table is stored in, yet another register called the global descriptor table register right. So, what the hardware does on each instruction is that it will dereference GDTR to get the address of GDT.

It will add the value of the segment selector, so these are called segment selectors, it will add the value of the segment selector to the base of the GDTR or to GDTR into whatever the size of the entry is. So, this is one entry of the GDT right to get another appropriate segment descriptor.

So, each entry inside the global descriptor table is called a descriptor, a segment descriptor. What the segment descriptor holds are values like base and limit right. And what the hardware is going to do is it is going to compute pa as va + base, and it is also going to check that va should have been less than limit ok. So, the hardware is doing this at runtime on each and every instruction all right.

(Refer Slide Time: 03:26)



So, if you just look at it, if you we just look at what it is actually implementing, if this is the address space of process P1, so let us call it P1's virtual address space P1's VA space and this is P2's VA space right, and let us say this is the OS's VA space all right.

And let us say this is the physical memory all right, so let me call it the physical address space PA space, and we looked we saw last time how the PA space is organized there as bios, there are devices and then there is actual physical memory inside the PA space right. And now what the OS does is basically sets up the segment descriptors in such a way that you know, each of these maps to different regions of physical memory.

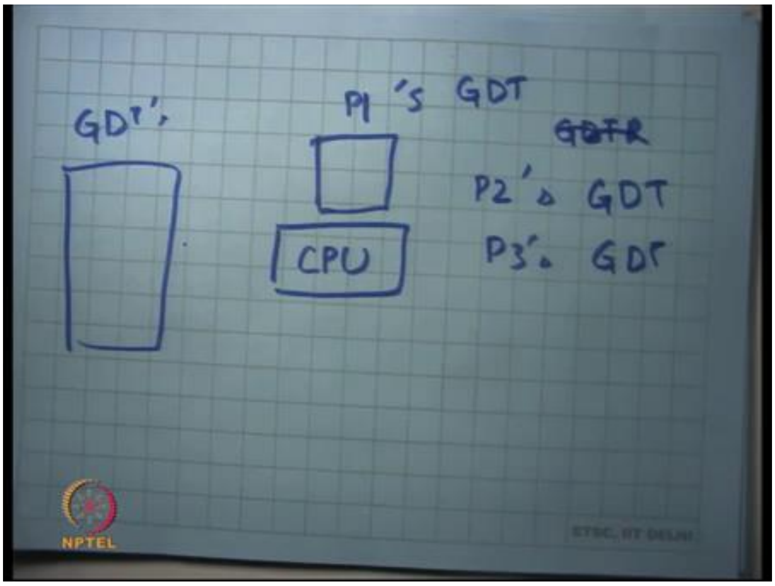
So, let us say you know P1's VA space is here, so you know let us say P1's VA space starts at 0 and goes up to some maximum value max. And let us say all the processes just for simplicity I have a maximum value of max that is maximum at address space a process can have, and let us say you know the OS also has 0 to some you know O max, OS is max right.

So, then you know I could not the OS can decide to place P1 anywhere he likes, so example he starts at B and so and, so what does he set his basin limit to he sets a base to B and limit to max right. Similarly, P2 can live here, when this case becomes you know B + 2 max let us say and he can put the OS somewhere, so let us say OS put somewhere all right.

So, this is how the physical address space looks like when I am using segmentation, and this is what the physical address the virtual address space of each process looks like. Notice that the virtual address space is completely independent, or the naming of the virtual address space is completely independent of where it lives in physical memory right.

So, it gives you independence, so it gives independence between the linker and the loader. The linker at link time does not need to worry about at what address I am going to get loaded right, the OS can make that decision at runtime all right ok. So, one way for an OS to implement virtual memory is that it can say so let us say here is a CPU all right.

(Refer Slide Time: 06:16)



So, an OS could say at this point process P1 is scheduled on the CPU. So, I am going to switch my GDTR, so I you know let us look at some ways in which an OS could actually allow multiple processes to live to coexist in the system. So, one way is you know let us say I have one GDTR per process possible right. A GDT is 2 to the power 13 entries, each entry is let us say 8 bytes, so roughly 2 to the power 15 bytes that is 32 kilobytes roughly ok.

So, you have 32 kilobytes of space per process just for doing implementing virtual memory all right. So, I could have you know another GD, so I could have P1 GDT, I could have P2 GDT, P3 GDT and so on right, so that is one way of doing it. And each time I decide which process is going to get to run on the CPU, I just switch the GDT all right, so I switch the GDTR to that processes GDT make sense, no question.

Student: Sir, why do we need different GDT for each process?

All right, so why do we need different GDT for each process, it seems wasteful. Another way to do it is let us have just one GDT, and each time I switch a process I just change the entries in the GDT now that is another way of doing it, so these are all ways. The second way is obviously superior to the first way all right, because I am not wasting space, I just have one GDT and I can just manipulate the entries on each context switch.

Student: (Refer Time: 08:30) very large.

Very large yes. I was just trying to you know give you a perspective on what all can be done right. So, basically if I want to schedule this process, I load his address space; if I want to schedule that process, I load that process address space and so on ok, all right. So.

Student: (Refer Time: 08:54).

Yeah.

Student: You say that how every process changes its address space and can only change the segment registers to specific values that is how it will be controlled. So, the OS knows for which process, which segmented, which values will be allowed, they look they are not a person it has to organize all those values in that way.

All right. So, let us just review protection how does protection work? The GDT itself lives in the address space of the OS. So, an application is not allowed to manipulate the entries of the OS or and so it is not allowed to manipulate the entries of the GDT all right. And secondly, I am not able to load another GDT a process is not allowed to load another GDT, because the instruction to load the GDT can only be executed at privileged level.

And so the process the processor has the concept of unprivileged and privilege modes, and so the process if its running an unprivileged mode cannot execute this instruction, so I cannot change the GDT; I cannot reload the GDT, I cannot change the GDT, because GDT is not mapped in my space and that is enough right.

Now, what the OS needs to do is it needs to do some kind of bookkeeping that which area, how much memory is allocated for which process and where I have allocated it and so that kind of bookkeeping the OS will have to do all right. So, for example, the OS will keep it keep information like what is the base and limit of all the processes that live in my system all right.

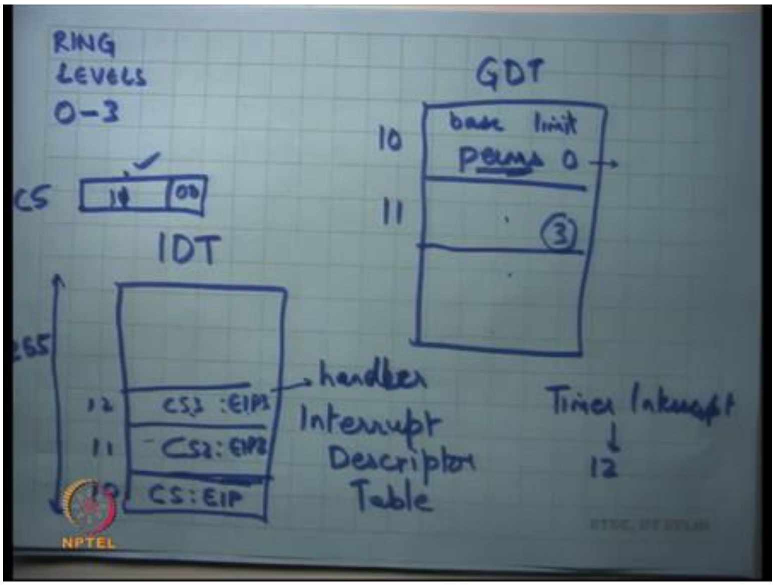
And so, when I switch to that process, I am going to pick up that base and limit and put it in my GDT all right, and I am going to start that process right. Where are where these data structures being stored, which contain the base and limit of each process.

Student: In the OS address.

In the OS address space right. A process should not be able to manipulate these things either right, so these things also need to be live living in a protected address space of the OS all right, The other thing is that sometimes so now, I am going to talk about how things like system calls are implemented. So far we have talked about a process having its own address space and a process being confined to his address space, so he cannot do anything else right, but we have also said that you know a process cannot live in isolation, it also needs to talk to the OS sometimes for example to make system calls.

And so, and also the OS needs to take control from the CPU at periodic intervals to be able to do scheduling and preemption. So, for that even though there are times when you would want that both the OS and the application can should be mapped in the address space. So, both if the so when the transition happens from application to OS, the OS is address space should also be mapped, so that in other OS can execute right.

(Refer Slide Time: 11:47)



So, GDT is actually each entry in the GDT, so let us say this is the GDT, this is a GDT entry. So, we have so far looked at base and limit, it also has certain permissions all right, let us call it perms and which basically say you know whether this segment can be de referenced in at what privilege level right. So, you also saw that a processor the (Refer Time: 12:10) processor has this concept of rings or ring levels and you know, they are 4 ring levels, but for most practical purposes let us just assume that 2 ring levels 0 and 3 all right.

And so, this permission is going to tell you at which ring at which privilege level am I allowed to dereference through this descriptor, right. So, for example if the permission says 0, then this descriptor cannot be de referenced if I am running in unprivileged mode. On the other hand, if this one says 3, then it can be accessed either in unprivileged mode or in privilege mode right.

So, apart from this base and limit descriptor also has permissions right. So, what that means is that an OS can map some of its address space always in the GDT. And so, by the way how does the processor know which privilege level I am running in?

Student: Last (Refer Time: 13:15).

Last two bits of the CS register right, so this last two bits of the CS register store what which privilege level I am running in. Also we said that a process cannot just lower its privilege level right, just I cannot just say you know if I am running at ring level 3, I cannot just say make it ring level 0, because otherwise then all protection is lost right ok.

So, if I am running at ring level 3, I can only set my segment values register values to one of the descriptors which have perms equals greater than equal to 3 right, just 3 ok. So, I cannot set CS to this descriptor. So, let us say this descriptor 10 and this is descriptor 11, you know setting it to 10 is illegal which means the two calls cause in or call the processor exception and the OS will take over right, but setting it to 11 is ok, so that allows the OS to keep itself mapped in the GDT right.

So, which means that if a processor needs to ever transition from unprivileged to privilege mode, it already has some of his address space map and so it can start using it immediately right. It does not have to set up the its own address space as soon as it enters the privileged mode.

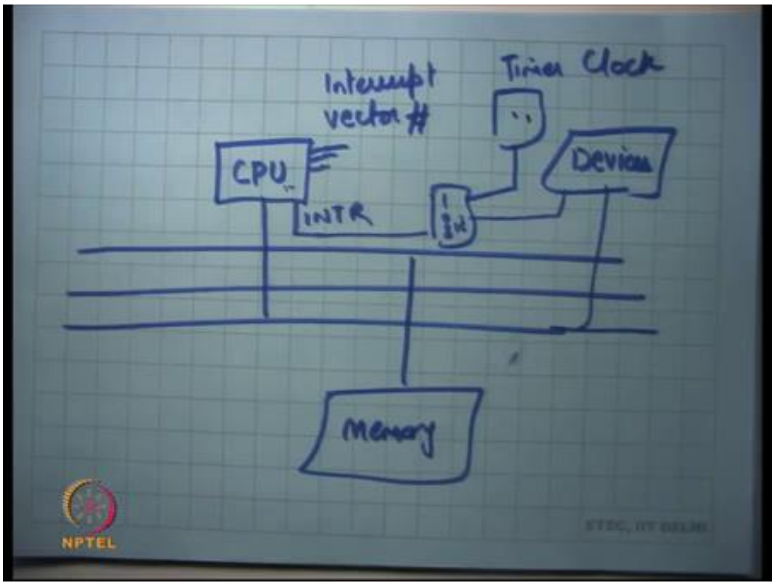
So, before we before you understand that let us also understand how does the; how does the processor actually change the privilege mode, change to the privilege mode right. So, I am actually executing the process and I am executing in privileged unprivileged mode and let us say an interrupt occurs right, so an interrupt occurs, and the OS needs to now run right. So, and the OS needs to run in privileged mode, because the OS will need to access its own data structures to be able to do scheduling.

For example, access all the base and limit values of different processes and then decide which one to pick and put it in there right. So, to be able to do this there is another table called the interrupt descriptor table IDT – Interrupt Descriptor Table all right, which has which is a table of you know 255 entries, each of these entries basically says has a CS colon EIP pair all right.

So, let us say CS2: EIP2 and so on and the 0, this is 1, this is 2 or let us say this is 10, 11, 12 just to just arbitrarily choosing these values right. What this means is that if an interrupt occurs; if interrupts number 10 occurs, then switch the instruction pointer and the code segment to this these values stored in the descriptor right.

So, for example if I say that a timer interrupt is going to occur at interrupt number 12, then whenever a timer interrupt occurs the execution is going to switch the code segment and the program counter EIP to these values.

(Refer Slide Time: 17:00)



So, once again let me just let me just say what I am what this means. So, let us just draw the hardware diagram. So, let us this is the CPU, and this is memory, and these are devices. The CPU has a pin which is called the interrupt pin all right and that interrupt pin is connected to some logic all right, and then that logic and that is logic just multiplexes much lots of devices and allows them to set interrupt the CPU right.

So, let us say there is a disk device as keyboard, this mouse as network, there is there and there are other devices on the motherboard and they are all going through some logic and ultimately they are connected they also have a connection to the interrupt port of the CPU right. So, when this line gets set the CPU gets interrupted right. So, whatever its was doing as a regular execution, it gets interrupted.

What does it mean for a CPU to get interrupted, it basically means started switches execution to some based on this interrupt descriptor table in this way all right? So, if so, apart from the interrupt pin there is also you know, they are also pins which specify the interrupt vector number.

So, let us say you know a device once attention it says I want to assert the interrupt pin, the interrupt gets raised with the CPU, but before it raises the interrupt pin the interrupt vector number has been set appropriately all right. The interrupt vector number determines which of these entries gets activated all right, let us say entry number 12 gets activated which basically means that the execution should get interrupted which means, whatever it was doing should get interrupted.

And the next instruction that should execute should be at this address CS3: EIP3 all right, so that is a semantics of the CPU. Yes.

Student: Then who sends the interrupt vector number?

Who sends the interrupt vector number, the hardware device all right? So, you have programmed so there is this logic that is you know sometimes programmable often programmable, you can say that you know this device has been connected to this interrupt number, this device has been connected to this interrupt number and so on. And so, the OS is can also program this device and now when the device actually asserts an interrupt that particular vector number gets sent to the CPU all right.

Depending on the interrupt number the corresponding program counter gets set right, this program counter is also called the handler of that interrupt all right. So, in other words when an interrupt occurs the handler gets called. So, now what happens is what the OS will typically do is that it will install these handlers a priori. So, for example it will say that on a timer interrupt execute this code that code will probably live here in the address space of?

Student: OS.

Of the OS right. So, what which means that the CS3 will be a segment selector of the OS I mean the descriptor that will point to will be a descriptor of the OS all right. And so, and this descriptor is allowed to have the last two bits as 0 which means ring level 0 all right, so that is one way then OS can actually rest control from of the CPU from the running application.

So, each time and so or the OS will typically install a timer handler, a timer interrupt handler. The timer interrupt handlers code will actually live in the OS address space, it will set up the interrupt descriptor table in such a way that the pointer the handler of the timer interrupt points to the handler in the OS address space appropriately. And irrespective of what I am executing as soon as the timer interrupt occurs, the process is going to switch to this code segment which may which is most likely a privilege code segment and it will start executing in OS mode right.

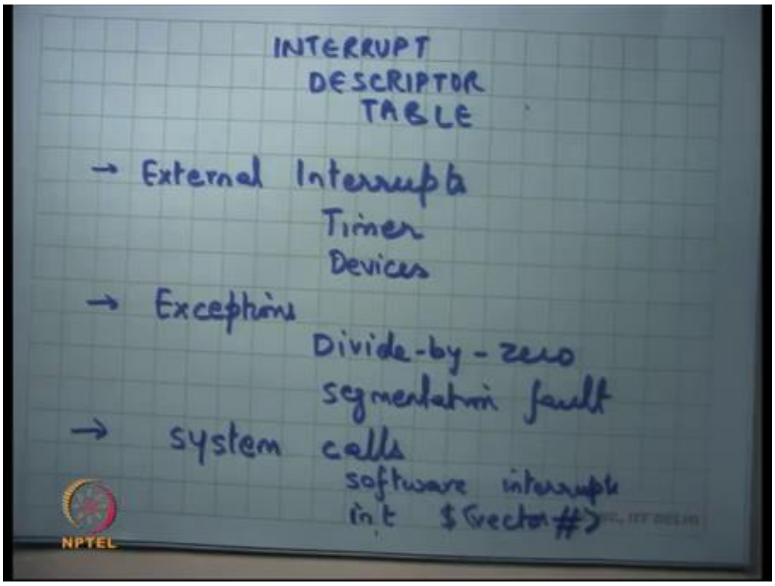
Here is an example, where CPU switches from unprivileged mode to privileged mode or ring level 3 to ring level 0. I said that a processor a process cannot just lower its ring level, but ring level can be lowered by other events like an external event like an interrupt got set right. So, in this hardware figure one of the devices will let us say a timer clock all right.

And the CPU can program the timer clock to say, I want an interrupt every 100 milliseconds right. So, the CPU has told the timer clock I want an interrupt every 100 milliseconds, because I want to you know execute after every 100 milliseconds to take stock of the situation; who all are running, who all are waiting, who should run next, etcetera, etcetera right this is called scheduling right.

So, this the OS will program the timer clock, how can it program the timer clock using the IO address space that we have already talked about right; either in out instructions or memory mapped IO, where you can just write two physical address space and you can get to (Refer Time: 22:41). You program the timer clock to in generate and interrupt every time quantum, let us say the time quantum is 100 milliseconds that interrupts gets in rated after that kind every time quantum and the OS gets to run at every time quantum right.

The OS gets to run in privileged mode obviously all right, the OS when it is running wants to be in privileged mode, so that it can do whatever it likes right. Irrespective of whether the processor was running in unprivileged mode or privileged mode as soon as the timer interrupt will occur, you will start running in privileged mode and you will get to do whatever you like to do all right.

(Refer Slide Time: 23:26)



INTERRUPT DESCRIPTOR TABLE

→ External Interrupts
Timer
Devices

→ Exceptions
Divide-by-zero
segmentation fault

→ system calls
software interrupts
int $<vector#>

Similarly, so we have looked at the interrupt descriptor table notice that the code segment in the CS value, the selector value of the code segment in the interrupt descriptor table has to point to one of the valid entries in the global descriptor table all right. And because of this perms field in the global descriptors, I am able to do this right. So, for this to be a valid value, I mean this selector is going to point somewhere in the table right.

And so there has to be an entry for the OS address space in this table, but I also have to make sure that the process itself is not able to change its entry to that descriptor right and so the perms field in the descriptor allows you to do this differentiation right. So, a perm 0 field can only be accessed if you know if you are executing in ring level 0 and you can only execute in ring level 0 if you for example, get an interrupt all right.

So, we saw the interrupt descriptor table. The interrupt descriptor table is so we have seen it, we seen that its used for external interrupts in particular we saw how it could be used for the timer device all right, but it could be used for any other device as well as for example, disk, network, printer whatever all right.

So, let us say you know your program is executing the printer has finished some job and once you just interrupt the CPU, it will send an interrupt the OS if it's same OS would most likely set up the handler to live in the OS address space, because its only the OS which is allowed to talk to the; talk to the printer right the device the processes are only allowed to make system calls to the OS. So, the OS the printer handler is going to get called and the OS can do appropriate handling for the printer.

The handling way mean just say you know just make (Refer Time: 25:40) some internal data structures and then return control back to the process all right or the handling could mean you know, notify the process that something is happened. What are some ways of notification notifying the process?

Student: Signals.

Signals all right, so that is one way of let us say notifying process all right. So, other devices it can also be it is also used for exceptions all right, so for example divide by zero all right. So, what happens if a process is running in unprivileged mode, but it makes it calls an instruction which actually it amounts to a division by 0; it executes an instruction which amounts to a division by 0.

The processor is does not know what to do, because the division by 0 is undefined. So, it is going to raise what is called an exception; raising an exception is internal to the CPU as opposed to the previous example, where we saw that an external device actually asserting the interrupt pin raising of an exception is internal to the CPU all right.

But the effect it has is roughly similar, which means you know every exception is allocated a particular number. So, let us say you know the divide by zero is allocated number 9 just making it up, it is not the real number, but let us say its allocated number 9. So, the 9th entry in the interrupt descriptor table should point to the handler of a division by 0 exception right.

So, typically what will happen is if a process executes divide by zero and exception is going to get generated, the exception handler must have been set up by the OS in such a way that. The handler actually points to OS address space and the OS is divided by zero handler gets called in privileged mode. The OS is divided by handler can do multiple things, it can either just straight away kill the process saying that you know he was not allowed to do that or alternatively he can send a signal to the process saying you had you did a floating point exception right.

So, we also saw SIGFPE in Unix which was just doing a signal to the process right. Similarly, if there is a segmentation fault what is a segmentation fault, there is a memory instruction where the address is not legal right. So, in this example all addresses between 0 and max are legal.

So, if he if the address that he tried was trying to dereference is between 0 and max its legal, but if it is he tries to dereference a negative address or he tries or you know anything if it is an unsigned, then the only possibility is if it tries to dereference an address which is greater than max, then the processor is going to raise an exception right.

How does the process know that it is an exceptional condition? The segmentation logic has this assert right, so that assert is going to fire saying that look you are less than you are you actually been trying to be greater than limit right. So, the processor is going to say there is something wrong it is going to throw an exception, once again that exception will have a certain number depending on the type of that exception.

So, let us say the segmentation exception number is let us say 14. So, the 14th entry should be set up by the OS at that it points to the segmentation handler for handler and the segmentation fault handler may choose to either kill the process or he may want to do something else or he may actually send a signal to the process itself like the SIGFPE that we have sawn seen on Unix all right ok, so that is those are two uses of interrupt descriptor table.

The third use is actually the same thing is actually also used often for implementing system calls right. So, we saw that system calls are basically functions that are implemented by the operating system and called by the process, but clearly the system called function, the system call itself has to execute in privilege mode right, it cannot execute into unprivileged mode right.

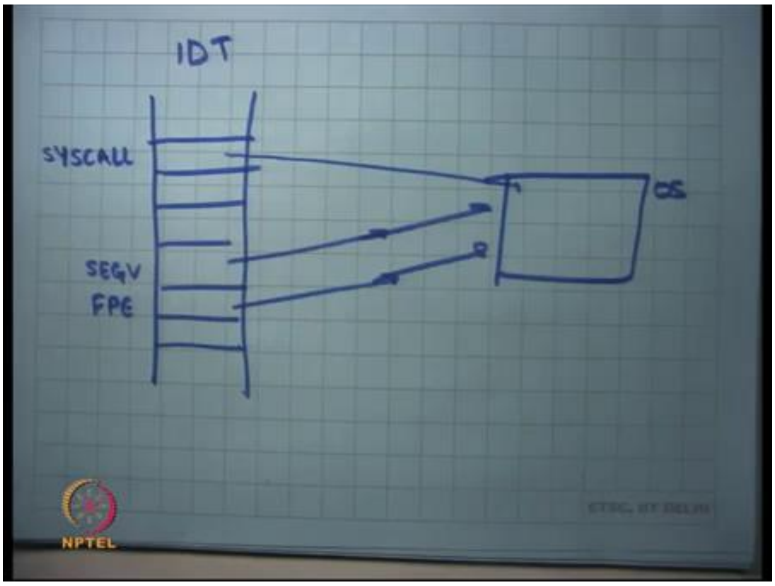
If I for example, make a read system call and the read and the and the file descriptor is actually pointing to a device, then I need to talk to the device right and that talking to the device may need privilege. And so, the system call itself needs to be implemented at privileged level, whereas the caller is in unprivileged level. So, the caller is unprivileged the callee is privileged, and so I cannot just implement it as a regular function call as you have seen last time right. So, one way to implement system calls or what is called software interrupts all right. So, software interrupts are nothing but instructions of the firm int dollar some vector number right. So, this int is the opcode of the instruction says interrupt.

So, apart from this which is you know somebody from outside is exerting the interrupt pin this which is I made an exception like it divide by zero or a segmentation what or others. Here is an example where the software deliberately and explicitly says raise this exception all right. So, here is an instruction that just emulates the raising of the exception number of the vector number specified as its operand right. So, I can say raise, the raise exception number?

Student: Two.

70, 80 right. And so, the 80th entry in the interrupt descriptor table will get activated and that particular handler is going to get called right.

(Refer Slide Time: 31:42)



So, how will system calls get implemented in this way let us say this is the interrupt descriptor table, and you know here all these entries. Let us say you know some of these entries are pointing to let us say this is for the segmentation fault and let us say this is for the floating-point exception. And they are pointing to appropriate handlers in the OS space right. And then you know there is this special number which the operating system can define as a system call number.

So, let us say this is the sys call number which will also point into the OS address space right. So, basically one way to implement system calls is that the process sets up the arguments somewhere let us say it sets up the arguments on in the registers, and then execute this instruction.

This instruction is going to execute a routine inside the OS and that the that routine should assume that the application is trying to make a system call right. And so, what does that routine is going to do is it is going to look in the value in the registers for the operands and exactly what you want to do.

So, for example, one of the operands could just be what system call you want to call whether it is fork or exec or read or write or open that could also be specified in the register right. You set up a register saying this is a systemic call I want to call, and then I just execute this instruction int that number, the OS handler gets to run, it checks the value of the register, and based on that it executes the corresponding functionality.

Student: (Refer Tim: 33:29).

Yes. Why do we need software interrupts, why do we or I think the question is also why do we need system calls? Why do we need system calls? Well, because the process cannot do everything itself; it needs some things that only the OS can do for it right that is the OS abstraction right. For example, I want to fork a process right. So, I need to tell the OS that please fork a process for me.

So, OS has an intermediate agent that is working on your behalf. You only make request to the OS that look, this is please do this for me. And the OS checks whether you are allowed to do this or not, and then depending on what he finds he is either going to do it for you or he is going to say no, I am not going to do it right. So, this there is a separation of privilege and the system calls are a way of actually bridging that gap between tool privilege levels right.

And the question really is how you implement the system call. So, we have already seen why system calls are needed. We are really talking about why, how you implement the system call right? Well, the system call also needs, so there is some hardware support in this in the mechanisms that has described so far. The hardware support is that there is an interrupt descriptor table which has these pointers use which are which are these long pointer, long jump pointers which are which has a computer code segment and a program counter.

And then there is a global descriptor table which uses those selectors to basically say exactly which address you should go to. And then the application just similar it is an interrupt and tells the OS that this is what I want to do in some sense. So, it is basically overloading the same interrupt mechanism to also do system calls and also do exceptions right. So, this is just this is one x mechanism that can do all these common things for you.

Student: Sir, all around the other system calls also in the interrupt descriptor table like fork and all.

Are all the other system calls also in the interrupt descript table? Actually, there is just you know you only need one vector number for the entire system call space that you have right. You can specify what system call you want in the register. You said a certain resistor value and that is going to tell you whether you wanted fork or exec or whatever right. Because if there are 300 system calls and I am going to exhaust my id space, you just have one entry for this system call and then you specify it as an argument what system call you want.

Student: All interrupt handlers are asynchronous?

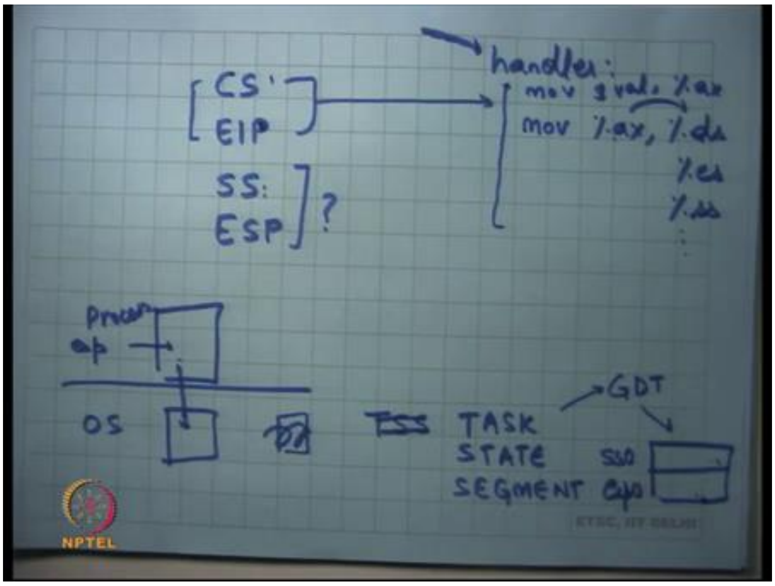
All interrupt handlers are asynchronous, what does that mean?

Student: Means when interrupt handler is called for this is running and interrupt handler is running separately.

No. So, good. So, now the question is how the interrupt handler executes, and in what environment does it execute. Firstly, the interrupt handler execution, so I was executing a certain instruction in a certain address space, an interrupt occurred or you know a soft interrupt occurred or an exception occurred, all these three are the same, I am just going to call it the interrupt occurred right.

So, I am executing a certain instruction in a certain environment and then interrupt occurred, I just switch my CS and EIP nothing else changes. So, I am still executing in the context of the original process in the same address space right, that is not completely true.

(Refer Slide Time: 36:56)



It also changes two more things sometimes which are, so it changes CS, EIP and it also changes SS and ESP in a conditional manner ok. So, on an interrupt four things get changed potentially, one is the code segment and EIP will definitely get change because you point want to point to the new handler. And what is also needed as I am going to discuss next is basically change the stack pointer you also want to change the stack. So, you going to operate on a new stack all right, and why that is needed etcetera I am going to discuss very soon right.

And SS: ESP basically determines the position of the stack. So, an interrupt can potentially also change these values, apart from that it changes nothing. So, the same address space, same everything. Of course, the handler the first thing it may want to do is you know reload its segment registers. So, the first instruction for example in the handler could be move some value to the ds register, so that in the next memory accesses which go through ds actually are OS values ok.

Student: Sir, is this stack called convention?

This so, the new stack that comes in is called kernel stack, but you know let us just hold on that one that thought for me ok. So, let us see, so a CS and EIP handler is, so let us say this is a handler, and let us look at a typical code for a handler right. Firstly, we have established that the handler lives in the OS address space. And the CA and the descriptor for to hcs points should already mapped in the gdt right.

The first thing the handler may want to do is let us say set setup ds right, some move, something some value to ax, and then move that value to ds right. That may be the first thing I want the OS may want to do because you know very soon I am going to start executing code which is going to do memory accesses and I want that those memory accesses should actually be for the OS address space right.

And so, for them to be for the OS address space all my segments should actually be pointing to the OS is segment descriptor right. So, I may want to do it for other segment descriptors also for example, and so on right. I could also do it for let us say the stack segment right, but there is a chicken and egg problem here right.

So, let us say I am a process and here is the OS, and I executed and, in some interrupt, occurred let us say the software interrupt hardware interrupt does not matter and the OS gets through run. The OS before it starts running the first instruction needs to save some context of where it was when it was interrupted, because you know when it is going to return it needs to restore that context. In particular what context as I need to save?

Student: Stack pointer, stack pointer.

The stack pointer ok.

Student: (Refer Time: 40:28) instruction.

The instruction pointer, the old instruction, anything that it is overwriting it needs to save the old value right, because it needs to restore the old value. So far, we have only seen that it overrides these two registers. So, those are the two resistors that it needs to save ok. So, it is overwriting CS and EIP. After it is done executing, it will want to restore CS and EIP to its original value, so that it can continue from where it is left right. So, we need to say that somewhere. The question is where does it save it?

Student: Sir.

Yes.

Student: So, handler is making the change to register to be need to these to that is well.

If the handler is making the change in the resistor it is happening in software. So, if the software writer is actually making a change to the resistor, he should be making; he should be making cop, he should save that in software, but the hardware does not need to save it ok. So, for example, the handler is overwriting ds before he does that, he may want to move he may want to save ds somewhere, but that is completely a software mechanism. Whereas, the saving of CS is a hardware mechanism because CS gets overwritten by the hardware right.

So, there is something is that the hardware is doing for example, its overwriting CS and EIP, and everything else in the software is free to do whatever it likes. Now, this is the typical thing what a software will do, but it may or may not do it and depending on whether it is actually going to make a memory access or not.

Student: Sir, but when we context switch between this and processes when we store all the registers or now process?

All right ok. So, I mean you can do all that. So, you have to save all the registers of the process that is true, but all that can be done in software all right. So, the handler itself will have logic to actually do this saving and restoring, but we are just talking about you know the process by which the handler gets called, even that process is actually clobbering some registers, and those registers have to be saved by hardware, because by the time the software gets run it will have no idea what the old values were right. So, some, some values need to be saved by the hardware.

And this in this case that we can clearly see the old CS and EIP need to be saved by the hardware right. Now, question is where does it save it right? So, one typical place where you usually store these things is the stack right. So, you know the process may have some value of ESP, and one response to the question could be, let us just push CS and EIP to the user stack right.

So, just encrypt just decrement ESP and stuff CS and EIP on the stack, on the top of the stack. But the process is untrusted you know a process could actually set up a ESP to zero and then call this instruction called int something. And the OS is going to now try to or that hardware actually is going to try to stuff CS and EIP at address 0, where that nothing lives or some other invalid address right.

And so what is going to happen is the process is going to the processor is going to go into recursive faults exceptions right, because it tried the hardware tried to push something to an invalid address and that is going to cause another exception. And now the exception handler is going to try to execute, and again you know you are going to try to for that to execute again going to try to push the exception, the exception handler is CS and EIP into the stack which is again, so the stack is invalid you know you are basically halted the system completely right.

The way you wanted to design a system is that these processes should not be trusted right. A process should not be able to bring down the system definitely not. So, I cannot trust any pointer that the process provides me. ESP is just a pointer that the process is providing me right, and it is supposed to hold the processes tagged, but if the process is malicious, it can just set it to something wrong. Or if you if it is just you know if it is also possible that the process actually running out of stack and in which case also, I do not want to be held responsible for that.

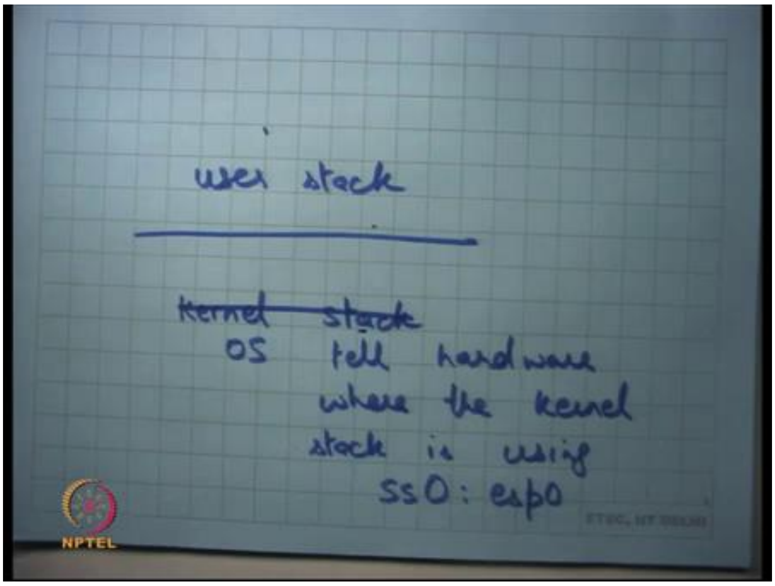
So, what the OS needs is it probably needs a stack of its own right. So, it needs some state space which is going to say it is this is my space, and this is where I am going to save these things before I execute right, and that is where I am going to restore things from when I execute ok. So, another, so the processor provides another data structure or hardware structure to do that which is called the task state segment all right.

The task state segment you know is actually also a selector which points to the GDT, but eventually it points to a data structure which contains two values SS 0 and ESP 0 ok. And what this means is that anytime you switch from privilege level higher than 0 to privilege level 0 load SS and ESP with these values before pushing the old values on the stack.

Student: (Refer Time: 46:02).

Right. So, let me repeat. Anytime I am going to switch from process to OS, before I attempt to push anything to stack, I am going to overwrite my SS and ESP with these values right. So, the x86 structure allows you to space you know there is some memory structure that allows you to specify that this is SS0 and this is ESP 0. And each time you transition privilege level from unprivileged to privilege SS will get loaded with SS 0, ESP will get loaded with ESP 0 right. And then the old values of SS and ESP, and SS and EIP will get pushed on this new stack all right.

(Refer Slide Time: 46:56)



user stack

~~kernel stack~~

OS tell hardware where the kernel stack is using SS0 : esp0

In other words, let us simplify it. There are two stacks; there is a kernel stack and there is a user stack. The user stack is untrusted. When I make a system call and most likely I am basically executing the stack pointers pointing to the user stack or it may point to something else also, I do not care. Before I execute even the first instruction, or before I even try to push anything on the stack, I should switch stacks to the kernel stack right. We are clear on that.

The question is this operation of actually pushing has to be done by the hardware. So, the hardware needs to know where the kernel stack lives right. So, the hardware need. So, the hardware provides the data structure, where the OS can setup values set to specify that this is where the kernel stack is.

And so, if you ever transition load this kernel stack before trying to push values on it right. So, the OS tells hardware where the kernel stack is using SS 0 and ESP 0 pointers a value all right. So, which segment what offset? Once again SS 0 should have been the value of SS 0 should be de referenceable inside the GDT right.

So, using this the kernel tells the hardware that this is my kernel stack, right now I am executing in user stack, but if there is for any reason a switch to privileged mode, the first thing you should do is load the stack. The second thing you should do is push CS and EIP of the user into this stack and not the user stack. And also push the old user’s SS and esp onto this stack, because you are also overwriting the stack, so now that needs to get pushed now.

Student: But sir till that time that previous stack also will be lost (Refer Time: 49:13).

Right, I mean the hardware has some temporary buffer to basically you know make sure that it is not lost, so that is all hardware implementation right. We just have to look at the semantics of the hardware. But the semantics of the hardware is on an interrupt or on a switch, it is going to switch the stack, it is going to save the old CS and EIP, and it’s going to save the old SS and esp.

So, save the old instruction pointer and save the old stack pointer and now it can run right, because this value of SS0 and ESP 0 was set by the hard by the OS itself, it is a trusted value, I can trust it right. No, process can modify it. Agreed?

Let us stop here.