

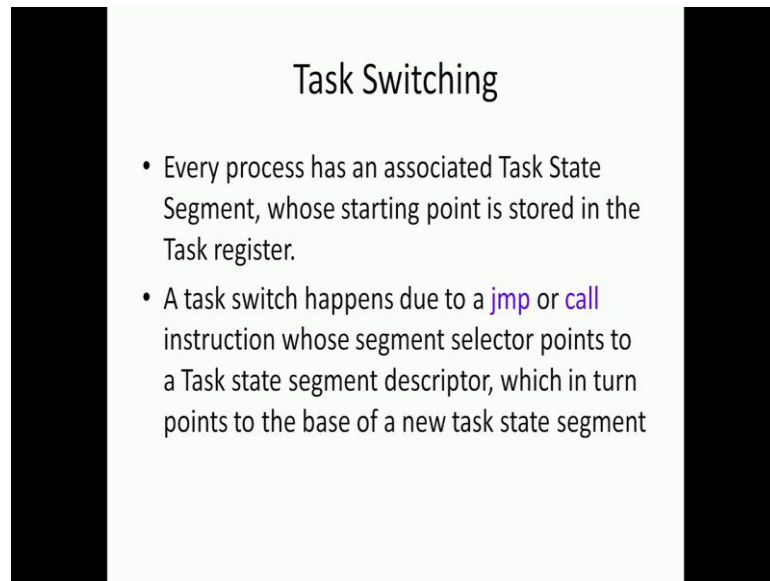
Information Security – II
Prof. V. Kamakoti
Department of Computer Science and Engineering
Indian Institution of Technology, Madras

Lecture No-15
Architecture Aid to Secure Systems Engineering
Session – 14 Task Switching and Interrupt Service

Welcome back to session-14 of this course. So, we will start answering the other questions of how we move from different privilege levels to other privilege levels. What we saw so far, in terms of protection that xi 6 offers, we can go from numerically lower privilege to a numerically higher privilege. But, the other part that is from a numerically higher privilege to a numerically lower privilege, we need to start answering it in bit detail. And, we will study that.

And, this particular part that we are going to address in session-14 is very important, in the sense that this basically dictates the security because this is going to talk about transitions from different privilege levels to other levels. And, one of the important aspects of security that we have been talking so far is from the privilege escalation. If I am going to get an undue privilege escalation, that is a vulnerability. So, from that point of view this particular session is extremely important.

(Refer Slide Time: 01:33)



Task Switching

- Every process has an associated Task State Segment, whose starting point is stored in the Task register.
- A task switch happens due to a `jmp` or `call` instruction whose segment selector points to a Task state segment descriptor, which in turn points to the base of a new task state segment

So, what is task switching? When I am executing as a process, I need to store my context. So, I need to have a place holder for my context. Why do I need a place holder for my context because we are working in a system, which is subject to interrupts. So, when an interrupt comes my process has to be suspended, the interrupt has to be serviced and again my process need to start from exactly the point where I am suspended to go back.

One of the very interesting example for this is, basically the round robin scheduling. In round robin scheduling which the operating system performs, is the very simple stuff. There are 5 processes, end processes. Each process gets one time quantum, they execute. At the end of the time quantum, they are pulled out and the next process is basically given time takes to execute.

So, in this context when I am, I as a process, when I am subjected to round robin scheduling, I will be pulled out at some point of time, and again I will have to go on start executing at the same point where I have left; from the same point, where I have left. Now that means, my status has to be recorded somewhere and again reloaded for me to start executing; and what? That status which needs to be recorded so that I start again, that state is actually called the context of the process. So, the context is that information

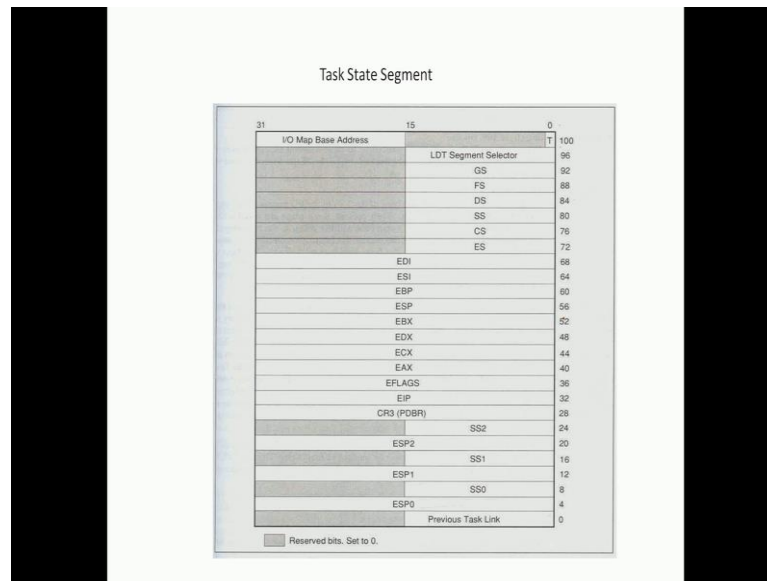
that one needs to store about a process, so that it could be resumed back after sometime from the same point where it stopped.

So, the architecture actually provides space, which is actually used by many of the operating systems. Some of the operating systems whose source code we can see; we can see that there they are using it. There is a place holder where I can store my context. And, that place holder is called a task state segment TSS.

So, with this as a background, let us just go through this slide that I have put on the screen. Every process has an associated task state segment, whose starting point is stored in the task register. So, task state segment is something like 104 bytes, where I store all the information that is necessary for me to restart again; and, where that task state segment is stored in the memory, in the operating system part of the memory typically. And, it is pointed to the start of the task state segment; that 104 bytes, where it is stored. You need to know the start of that segment. And, that is stored in that task register. Like, how I have a global descriptor table register, which stores the starting point of the global descriptor table, how I have a local descriptor table register which stores the starting point of a local descriptor table, how I have a CR 3 register which stores the starting address of the page directory. Like this, I also have a task register which stores the starting address of my task state segment.

So, when I am going to move out, what will happen? I store all the architecture, not the program. The architecture by itself stores all the contents of different registers, etcetera, in to that task state segment. It dumps it there. And then, it pulls me out. Next time, when I want to come from that task state segment, it will load all the registers. So, I start executing from the point I left. So, a task switch that is from one process to another process happens due to a jump or call instruction. Whose segment selector points to a task state segment descriptor, which in turn points to the base of a new task state segment. So, we will now go through this second part of the line, which is very important from a security perspective.

(Refer Slide Time: 06:12)



Now, let us understand what a task state segment is. As I told you, it has something like hundred and four bytes, zero to hundred and four. And, note that there are several things here; many things you will immediately follow, many things you may take some time to follow. What are the things here? All the general purpose registers starting from forty to sixty eight as you see here; EAX, EBX, ECX, EDX, EXPE, BPES, IEDA. Then, your flag register. When you just complete execution what was your flag register? Then your instruction pointer; which instruction are you currently executing? So, that I have to start which instruction you have to execute next. So that, when I go out and I restart, I have to start from that instruction; so EIP, then CR 3.

CR 3 is a control register which stores the start of a page directory. Please note, when we go and study the operating systems, many of the modern operating systems give a separate page, direct paging mechanism for every process. For every process, we will have it is own page directory. And if I switch from one process to another, my page will also directly change. So, we will just leave it at this point, but please have it in your mind that whenever you attend the next level course on operating system. Please, understand that every process will get us its own page directory. But, we will not go beyond that, then it becomes much in to the operating system. But, we also store this starting address of a page directory there. So, these are all the stuffs here.

Now on top of it, you have ES, SS, CS, DS, FS, GS. These are all the segment selector task for your different; stored in your different segment registers. So, the content of your segment registers are already stored in this. So, when I move from; when I am pulled out, the current values of all my general purpose registers, the current value of my flag register, the next instruction that I need to execute, the current page directory, start of the page directory which I am using, the current values of all the segment registers are all dumped here. Do you understand this?

And then, there is LDT segment selector because I can have my own LDT. So that LDT segment. I will come to LDT segment selector slightly later. But, I have my own LDT. Let us even assume that I need the base address of that LDT. That also is stored. Do you get all these things? We will not go on to I O map base address. That is something that again we will try and do beyond operating system scores. But, are you able to understand? Say, from byte 28 to byte 96. Is there any doubt? So, all these things are about my process and all those values are immediately dumped here, automatically by the architecture. In addition, please understand that there are three more stacks SS 0, SS 1, SS 2 and 3 stack pointers ESP 0, ESP 1 and ESP 2. And then, there is a previous task link. This is basically if I do nested task switching, the link will be established.

So, if you studied normally the operating system course, if you have taught that course or if you are studying that course there will be something like process control block, which stores the context of the process. And, they will be in one link list. This previous task link you will sort of establish that link list for. You can establish that link list. It is up to the operating system to utilize it.

So, now why I need three different stacks and three different stack pointer values here? What is SS 0? It is pointing to a; this is a segment selector pointing to some stack descriptor. SS 1 is again some segment selector pointing to some stack descriptor, similarly SS 2. And ESP 0, ESP 1, ESP 2 are the values of the stack pointer, again stored here. Why I need this? This has lot of security implications, when we talked about stack smashing, right, that vulnerability happen because we shared a stack. And, we shared a stack and the vulnerability actually became very problematic because the calling routine was at a higher privilege than the call routine; meaning, numerically lower privilege. So

when the called routine, I went and filled up my own malicious program and changed the return address. Where did I do it? On the stack, right? And, when I went back to the, when I gave a return, then what happened? The called calling routine which is such as a higher privilege started executed that. And, it started executing my program at its privilege. And, that is why I was able to get in to the system. Do you remember yesterday's class? This is what exactly happened.

Now, all these things happened because I am using stack, the same stack. The same stack is used by codes of two different privilege levels. The calling program was at a different privilege level than the called program. Now, the called program went and wrote all its malicious code in the stack and it returned back. And, the calling program essentially started executing this malicious code at its privilege. So, the vulnerability is because two privilege levels are sharing the same stack.

To get rid of this, xi 6 gives you a wonderful opportunity here. Note that if I am a privilege level three code and I am calling a privilege level two code; calling in the sense that I may do a, there can be an interrupt. Or, I can do through a there can; mostly there can be an interrupt. The interrupt is the most important. When there is an interrupt, immediately your interrupt service routine might be in privilege 2 or it can be in privilege 1 or it can be in privilege 0. When the interrupt service routine is in privilege 2, it will not use the stack of my stack. It will start using the stack which is given by SS 2 and ESP 2. SS 2 is the stack segment; ESP 2 is the stack pointer.

So, when the interrupt service routine is of privilege two, it will start using SS 2 and ESP 2 and it will not touch me. If it is using my own stack, after the interrupt service routine comes back there is a chance that I can use that stack values and see if there was something happening there. The interrupt service can do something very confidential. Correct. And, those confidential information can leak through the stacks. You are getting that.

So, when I am a process of privilege level three and my interrupt service is happening at privilege level two, my stack will not be used. The stack of the SS 2, ESP 2 will be used.

If it is going to be privilege 1, SS 1, ESP 1 will be used. SS 1 will store the offset into the segment descriptor table from which I will get the base address, etcetera.

And, this is the; so, I will now define a stack segment. By the TSS 1, we will describe a stack segment. It is a selector. It goes to a descriptor table. There will be a descriptor, from where there is a base address limit everything for the stack segment. And, ESP 1 is the stack pointer to the stack segment. You are getting this. When I go to, when my privilege, when my interrupt service routine is a privilege 0, then it will use SS 0 and ESP 0; so, two codes of two privilege levels will not share the same stack. And by this, some major vulnerabilities can be avoided.

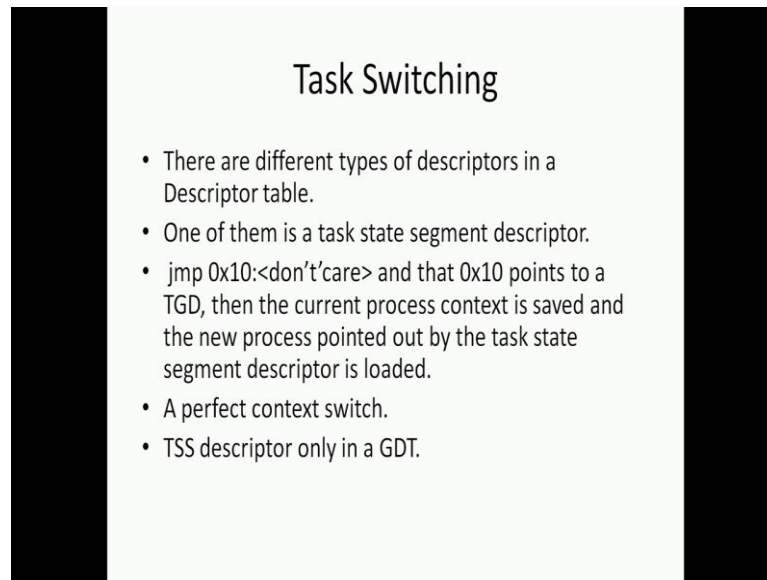
So, when I as an operating system, I ask so; when a process comes to me saying, yes, I want to execute, so what are things I will do? I will create a code segment, I will create a data segment, I will create stack segment, I create LDT. And, I will also create a task state segment like this. And, I will also populate this SS 0, ESP 0, SS 1, ESP 1, SS 2, ESP 2. I will also fill up this EIP with the start address where I have loaded the program and then give control to the program. So, it will start executing. Whenever this is being pulled out, immediately what will happen? The entire context would be saved in this thing, so that, I could restart whenever I want to restart it. You are getting this. You are able to appreciate how task switching happens.

The most important thing in this task switching is from a security perspective, it is very important that there are different stacks assigned to every process. And, if I want an interrupt service routine of a different privilege to service a process which is of other different privilege, I need not share the stack. The architecture itself, please understand, there is architectural aid to security. The architecture itself will see that it will switch the stack. You use some different stack, rather than this. And, that will give us a lot more protection in terms of isolation. Are you getting this? So, this is why yesterday's stack smashing, the stack smashing that I thought in different previous sessions are very important.

So, you appreciate from that point of view. Even though stack smashing may not happen now, there could be lot more important security measures that are put to get rid of stack

smashing today. But still as an example, it works good because this explains many of these concepts here.

(Refer Slide Time: 17:32)

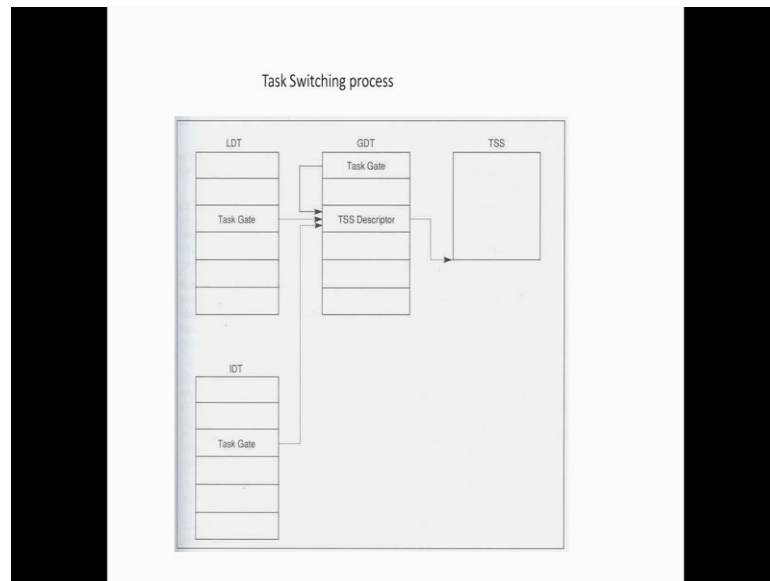


Task Switching

- There are different types of descriptors in a Descriptor table.
- One of them is a task state segment descriptor.
- `jmp 0x10:<don't care>` and that 0x10 points to a TGD, then the current process context is saved and the new process pointed out by the task state segment descriptor is loaded.
- A perfect context switch.
- TSS descriptor only in a GDT.

So, now what is this task state segment? How can we handle it? So, let us take a few minute for this and we will talk about this. There are; as I told you, there are different types of descriptors in a descriptor table. So, I also told you there could be something called a user descriptor and a system descriptor. This task state segment, it is a segment. So, there will be a descriptor which is describing this state segment. And that is called a task state segment descriptor and that is a system descriptor.

(Refer Slide Time: 18:30)



Now, this is how it looks like. Please look, there is GDT. In that GDT, there is a TSS descriptor that will point to the start of a task state segment. What is a task state segment? This is the task state segment. The starting point of this will be pointed out by a descriptor, which can be part of your GDT alone. It cannot be a part of your LDT. A task state segment descriptor can be only in a GDT.

Now, there could be something called task gates, which will point to this TSS descriptor. This task gate can be in your LDT, it can be in your GDT, it can also be in your interrupt descriptor table. I will shortly talk about interrupts, very shortly. So, your task gate can be in your LDT, it can be in your GDT or IDT.

Let me say that your task gate is at some third location here. So, how do I do task switching? I just say jump 0 x 10. So, 0 x 10 is the second descriptor. So, this is the second descriptor. When I say jump 0 x 10 in a GDT or whatever, when I jump on this task gate, when I jump, when I execute JMP with this selector as the number, so this is the third, 0, 1, second selector. If I say jump 2: whatever, I do not two colon, I can have any random value. The moment I say jump 2 colon; that means, it is going to come and see what is this second selector in your GDT or LDT, correspondingly. So, when I see that I go and realize that is a task gate. The moment I realize that it is a task gate, I know

that there is going to be a task switch. That means, I am going to be pulled out and something else going to happen. The moment there is a task gate, that task gate will point to a TSS descriptor. That TSS descriptor will point to a TSS. You followed?

Now, I as a process, who is executing. There is a process A who is executing jump two colon, do not care. My process there is a T R for me. Task register for me, that task register will be storing what my task gate's value. See my task gate can be in, say, sixth location here. And, my TSS descriptor that will point to another TSS descriptor gate, which will be another TSS here; so, what happens is whatever is available in this TSS descriptor will be loaded. After, whatever is there in my context will be loaded into this TSS, the older TSS descriptor. And, this new TSS descriptor content will be loaded into the registers. I am sure I have not explained it fully. So, I will again do it step by step.

So, I am process and I have, I am executing. And, I am executing jump two colon, do not care. The two corresponds to the second entry in the LDT or GDT. Let us say it is the LDT. The two is a task gate. The moment it is a task gate, I know that I am going to be switched out and the new process is going to start executing.

So, that two is, that descriptor here is pointing to a task state segment descriptor. This task gate is pointing to a task state segment descriptor. Correct. This task state segment descriptor will point to a task state segment. So, the moment I say jump two colon, whatever do not care, it will come to this; the architecture will come to this TSS descriptor. It will come here and it will load the content of this TSS into my, in to the corresponding parts of your architecture. For example, TSS will have EAX. The value of the EAX stored here. This also store EAX. So, what will this store? EAX, EBX, all these values it will all be loaded into the corresponding to the EAX, EBX, ECX, EDXS, S. All the segment selectors will be loaded. Note that, EIP is also there; so, the instruction to be executed that will also be loaded. And then, this program will start executing.

And, there is a task register. The task register what will be the value now? It will be two. The new value of the task register will be two. You followed. You got it. So, when I say jump two colon, do not care, it comes to the second descriptor here. It finds that it is a task gate. Immediately, it knows there is going to be a task switch, who the architecture

will know. Immediately, it will go. This task gate will point to a task state segment descriptor. Then, it will go to a task state segment descriptor and it will load the content of the new process completely, including the instruction pointer; which will tell you which instruction you have to execute. That is why I said jump two colon, do not care; because whatever I put there it has no value. What this EIP, whatever EIP is there, that will be loaded and the new process will start executing. And, your task register will store now the value two.

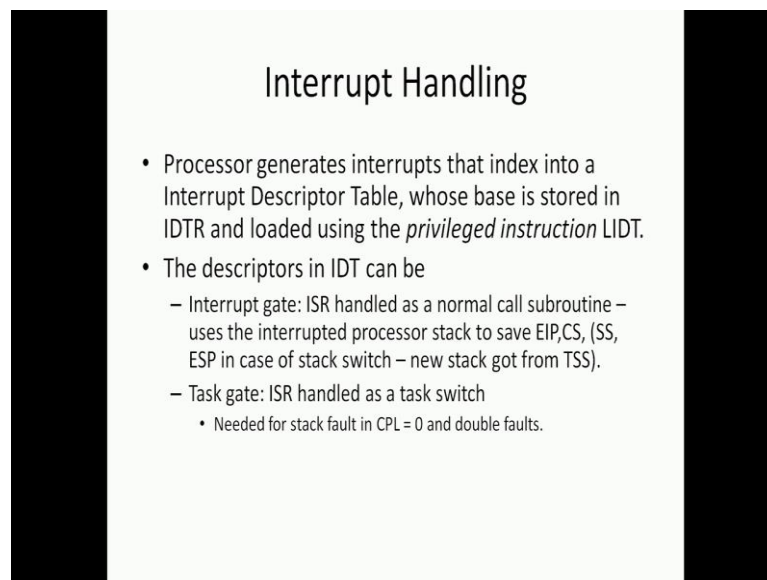
So, before doing all these things what will my old process, what will the task register be there for the old process? See, I am a process, I am executing two colon something. And, when two colon something is executed, immediately the new process start executing in the manner. before executing the two colon, the jump two colon, what would be my status? I am another task I would have had another task register, task register value. That task register value will be pointing to what? Some other task gate, some other task state segment descriptor there; so, the moment I realize that this is two colon something, the moment I realize that this is a task gate, immediately I will go to my task register, I will find out where my task state segment is available, I will go and dump my values completely and then this values will be loaded fresh. And, all these things are taken care by the program, by the architecture.

So when the architecture, so I am process a, I am executing jump two colon something. I go to this 2, I find it is a task gate. That is, architecture finds it is a task gate. The moment it finds that it is a task gate, immediately it will go to the task register, it will find out where the task state segment of the process is, the old process, it will take all the values of the different registers etcetera and dump into the task state segment. Then, come to this two, go to this TSS descriptor, go and find all these task state segment here and then load it into the registers now and start executing the process. And by this, what happens? I have done a task switch from one level to another level. do you all understand this?

So, I have a task gate, then TSS descriptor, then I go to TSS. So, there is 2 levels of transition by I need to do, before I do the context switch. Do you understand this? that is why when you study operating systems in many of the books, you see that thread

switching is much faster than process switching. There is a statement that is put. Thread switching is much faster than process switching. Why? This is one of the reasons; because at least I need to go through multiple levels of thing. First, that task register should find that task gate and that task segment selector and then that task state segment, put all the contents there, then again come to the new task gate, come to the new TSS descriptor, get the new values loading and then start executing. So, so many things have to happen before a task switch essentially takes place. And, all these are supported by the architecture. And, that is why task switch is faster than; if everything has to be done by software, it becomes much slower. You are getting this? Yes or no? Followed?

(Refer Slide Time: 28:40)



Interrupt Handling

- Processor generates interrupts that index into a Interrupt Descriptor Table, whose base is stored in IDTR and loaded using the *privileged instruction* LIDT.
- The descriptors in IDT can be
 - Interrupt gate: ISR handled as a normal call subroutine – uses the interrupted processor stack to save EIP,CS, (SS, ESP in case of stack switch – new stack got from TSS).
 - Task gate: ISR handled as a task switch
 - Needed for stack fault in CPL = 0 and double faults.

Now, the same thing is about interrupt. See, there are 2 types of interrupt are of. So, we call a generic name called exceptions. Exceptions mean something that should not happen that in the normal course of the program and that has happened. So, that is why it is called an exception. The literal word meaning of exception also is valid here. exceptions are of two kinds. One is called trap, another is interrupt. trap is, I fall into a trap means I do some nonsense and I fall into it. That is what I called it as trap.

interrupt means somebody else is stopping me. So, a process or a programming execution can stop executing due to 2 reasons. Reason number one, I do something

which I should not have done and I fall. for example, divide by 0 or segment over stack overflow or segment overflow. All this thing I am not supposed to do. But, I have done it. So, this is a trap. I fall into my own trap. That can be some timer interrupt coming, right, or some other peripheral interrupt coming, which is I am not responsible for it. So, those are called interrupts.

So, generic name for exception, for these types of things is called the generic name is exception. And, the generic name is divided into 2 parts. One is trap, another is interrupt. Trap is something that the program does to get it out. by divide by 0, I am thrown out. Segmentation over flow I am thrown out, page fault I am thrown out, stack over flow I am thrown out, double fault I am thrown out. So, many reasons will be thrown out. Alignment check fault I am thrown out. On the other hand, I get an interrupt from an external source like a keyboard interrupt or a timer interrupt. Those are called as interrupts. So, exceptions are of 2 types; traps that are generated by the program; interrupts that are external to the program.

Now for some reason, the literature says everything as interrupts. Interrupts include trap and interrupts. Now, how is the interrupt handles? It is exactly handled like the other descriptors. The many of the or all most all the contemporary architecture have something called vector interrupts. What is vector interrupts? for every interrupts, there is a number assigned to it. So, there are 255 possible interrupts that you can define. And, some are already defined. That is. So, this forms a vector of size 0 to 255 or 256. So, 0 to 255; and, each number in that corresponds to some interrupt. The 0 to 31 is already defined by the architecture. For example, 0 is divided by 0, 2 is debug, 7 or 8 is double fault, 8 is double fault, I think 7 is some stack fault, 13 is general protection fault, 11 or 12 is page fault. So, all these things are some of them are already defined by the architecture.

Beyond 32 to 255, you can define your interrupt. for example, I add a peripheral device. for the peripheral device, I give a series of interrupts. So, if I come through those interrupts that I know that the interrupt is coming from that peripheral difference. So, I know which device driver should I use for a given interrupt. for keyboard, there is a keyboard device driver; for a graphics stack, graphics stack device driver, p c a express,

per a device. For different devices, I will have my different device drivers. So, this is the interrupt. So, if I have say 255 interrupts or 256 interrupts, for in every interrupt I should know where the interrupt service routine is located. And, what is the privilege level in which that interrupt service routine will execute. I need to know that. So, that is given in what you call as this interrupt descriptor table.

In interrupt descriptor table, they will tell you there are 0 things. One thing is it will tell you that this is the code segment, it will give you a selector for a code segment, go and execute selector for this code segment. That means, go and execute the tenth segment in the GDT or the tenth segment, mostly LDT may not be used, but you can use LDT also. But, 10th segment in the GDT; 10th code segment in the GDT; so, it will tell you one segment selector which points to a code segment. Then, it will give you an offset within that code segment. So, go and start executing from that offset from that code segment. And, it will also give you a privilege level. That code will have a privilege level. So, if I am going to execute some something which is not. So, I am privilege level three code and a interrupt service is going to be a privilege level one code, then what will happen? Importantly, the stack will change. The stack one, ESP 1 will be used there. So, that is a most important thing. So, I will generate an interrupt. That interrupt can be divided by 0 means, the 0th interrupt. So, I will go into the 0th location in your IDT that will give me one code segment and that will also give me an offset within that code segment. So, I go to that code segment and I look at that offset; from there I will start executing it.

Similarly, if I generate say, I did it one; interrupt one; so, this first entry in your first, 0 first, no, the second entry. Second entry in your IDT will give me a code segment from which I will get the base. It will; the same thing will also give me an offset. So, from that offset I will start execute it. So, when I generate an interrupt, I go to a different code segment and start executing from some offset within that code segment. And, if that code segment is of a different privilege, then there will be a start that is wished, so that, there is now some amount of security that I begin here. In addition, I can also start a new process itself. If I put a task gate here, then there will be context switch. In the previous thing, what I told you? it is only a stack switch, but then the interrupt service routine is essentially working like a function. It is working like a function call. And after that, you return back and you start executing your own process. But, in some cases I will do a

complete task switch. So, the interrupt service routine is a complete new task. It will finish. And if you want, it will come back to this old task. You got this point.

So, when I generate an interrupt, I can handle the interrupt service routine like a function call; that means, my context is still there, the interrupt service routine will work in my context with, but with different stack in case where the privilege levels are different and then it will return back to me and I will start execute it. So, my context is not stored somewhere and retrieved back. You are getting my point. So, that is how I handle interrupt service routine like a function call. But, I can also handle interrupt service routine like a task switch, where completely I move off and the new task starts executing.

The interrupt service routine is not a function, but it is a new task by itself. And then, after the task it may call me or it may not call me. Now, so in the IDT in the interrupt descriptor table as ever you will be actually implementing this in the lecture then you; many things should become much more clear. But for our understanding, in the inside the IDT, there are 2 types of, you know, entries. Entry number one is an interrupt gate. It is called an interrupt gate. What will the interrupt gate do? it will just treat that interrupt like a function call. It can also be a task gate as shown in this figure in which it will treat this as a complete context switch. So, when an interrupt comes depending upon for that interrupt, so every interrupt as a number 0 to 255. Depending on that entry in the interrupt descriptor table, if it is going to be interrupt gate, then what will happen? I will just treat this interrupt like a function call. If it is going to be a task gate, then I am going to treat this interrupt like a task context switch.

One very simple example of interrupt gate is that, when I do a printf, for example, then it is like a function call. But, please note that I am going to in operating system routine. It is not my routine. It is at a different level. but if the scheduler is going to pull me out, but due to a timer interrupt, then it is going to be a complete context switch; because a new process. At the end of this interrupt, it may not even come back to execution. Somebody else will come into execution. So, it can become a context switch. So, there are 2 ways by which interrupts are handled.

So with this as a background, let us just go and see this slide. In interrupt handling, processor actually generates interrupts that index into a interrupt descriptor table. So, please note that every interrupt has a integer value. And, when the process generate this interrupt that actually index into a interrupt descriptor table. And, I should know; index means, I should know where the base is. So, the base is stored in a register called IDTR. It is interrupt descriptor table register. And, the way I can load that IDT R, I have to initialize that value to point to the interrupt descriptor table. The way I load that IDTR is using a privilege instruction called LIDT. Like how I had LGDT, LLDT, I have now LIDT which will load that particular value into the interrupt descriptor table register.

The descriptors in the IDT can be of 2 types. Type number one is called a interrupt gate in which the interrupt service routine handled like a normal call sub routine. And, the interrupted processes stack to save all your EIPCS. And SESP in case of stack switch, new stack is got from the task gate segment. I also explain what a stack switch is and why it should happen. Your interrupt service routine is at a different privilege level than the actual process that is executing, while the interrupt was coming. Essentially, there will be a task stack switch. If the next type of a descriptor inside an IDT can be a task gate, if it is a task gate, your interrupt service routine is handled like a task switch. So, for example, if I have a stack overflow at privilege level 0, I cannot use an interrupt service routine there. Please, understand.

For that, I need to have a task gate to handle this particular scenario. Why? I am a PL level zero code, right, and there is a stack fault. Stack is over flowing. I need to have a double fault handler. I need to do an entire stack switch. I need to do it. I cannot handle this using a interrupt gate. I need a task gate. Why? I repeat the question. There are two types of descriptor in the IDT. One is interrupt gate and a task gate. Why do we need task gate? why do we need interrupt gate? That is almost clear. Task gate, I said that the complete interrupt service routine is handled like a task switch. The interrupt service routine will be become a new task by itself. And, this is needed for stack fault, mini stack over flow fault, when your privilege level is 0.

So, the first question, first, yes very good. So, the first thing is when I am at privilege level 0 and I have a stack fault (Refer Time: 42:36) stack is overflowing. First and

foremost, your, the stack switch will never happen; because you are already in privilege level zero. So, it will not take another privilege level zero stack. So, it will start using your own stack. And, if you are going to have an interrupt gate to handle this, then what will happen? This fellow, the, as because of this interrupt, more things will be tried to push into the same stack; because what will happen in the interrupt gate. ISR handled as normal calls of routine uses the interrupted processes stack to say EIPCs, etcetera. You got this. So, what will happen? I am having a privilege zero code. I have a stack over flow. And, I put only interrupt gate there. Then, what happen? more things will be pushed into this stack and that will cause more over flow. So, this is actually called a double fault.

See, double fault is not that when I interrupt, service routine itself is creating another fault. When I am switching, see, when I am switching from the normal routine to the interrupt service routine in that process, I am switching at the; while switching, I do lot of things. for example, I push something into the stack, etcetera. When I am doing that process, in my movement from a normal process to the interrupt service routine, in that in between if again some fault occurs, then it is called a double fault. In this case, it is going to happen. If I have an interrupt gate there and I am a privilege level 0 and have a stack overflow, what will happen? I will start pushing. If I have interrupt gate, I will start pushing more things into the stack, which will create another over flow. Even before I start executing the interrupt service routine, I will create another fault. And, that is why it is called a double fault.

It is not a nested fault; it is a double fault because nested fault is something I come to nested fault. Now, I cannot handle. Now this double fault, if again goes on to a stack fault, then it will again create one more fault. Then, this will go into an infinite loop. You got this. The hardware will go into an infinite loop. It is not the software creating an infinite loop. please understand here. It is the hardware itself will go into a infinite loop. It will not go and it has no way of executing the next instruction at all. You got this. There is a very subtle difference between a software infinite loop and this infinite loop.

So, what we need to do is, when a privilege level 0 gets into a stack fault I cannot use the stack any more. So, I need to go and do a complete context. This own the process and

stack may know ISR to go and find out why this calamity has happened. So, that is one simple example where ISR, where a task gate should be a part of your IDT. Are you able to follow this? If I have an interrupt gate, it would have landed up in exception.

Now, last several years in several interviews or selection of students, I have been asking, what is a double fault? The answer I get, “sir, when an interrupt service routine is executing, the interrupt service routine generates an exception, then it is a double fault.” No. It is not a double fault. When I am moving from the normal routine to the interrupt service routine, interrupt service routine is yet to execute its first instruction. When I am in the process of movement, I generate a fault. Then, it is called a double fault. I go to the interrupt service routine and it starts executing. When it is executing, some other instruction creates a fault. Then it becomes a nested fault. It is treated like a normal fault. You got it. So, this is the subtle difference between a double fault and a nested fault. It is not an interrupt service routine, while it is executing creates another interrupt. That is not double fault. When I am moving from the normal routine to the interrupt service routine, at that point when I create another fault, then it is a double fault. And, one example for that is given here. You got it. good.

(Refer Slide Time: 47:22)

Interrupt Handling

- Processor handles a total of 255 interrupts
- 0-31 are used by machine or reserved
- 32-255 are user definable
- 0 – Divide error, goes to first descriptor in IDT
- 1 – Debug
- 8 – Double Fault
- 12 – Stack Segment fault
- 13 – General Protection Fault
- 14 – Page Fault

Now, I told that there can be 255 interrupts, in which the first 32 0 to 31 are used by machine. And, they are reserved 32 to 255 are user definable. That is for every peripheral, you can define interrupt. And then, there are advance programmable interrupt controllers which you can study. So, there is lot of architecture that can go into how to handle interrupt quickly. And for especially real time system, you need this interrupt to be very fast in response. So, there are lot of interrupt controllers that are basically in hardware to see that this becomes very fast. So, that is another aspect of architecture.

Now, in terms of this interrupt handling, the first 32 are reserved by the machine and these are quite important for us to note; because if at all I am going to catch vulnerability, I need to catch a reducing interrupts. Please understand that. So, 0 is divide error and it goes to the first descriptor in the IDT, the kth fault will be kth descriptor. So, if I have a divide error what should I do? the hardware will first take it, will find it is 0, it will, hardware will go to the first entry in the IDT. And, what will be there in the first entry in the IDT? it will give your code segment and an offset. So, it will go and start executing that. And, that will take as I said it will just print divide by 0 and it will exit the program. The printing divide by 0 is done by an interrupt service routine, corresponding to the divide by 0 fault. One is debug; 8 is double fault, 2 to 6 are reserved, 2 to 7 12 is stack segment fault, 13 is general protection fault, 14 is page fault. So, these are all some of the 32 faults that are defined by the machine.

So, what we have seen here in general is about task switching, in which I can switch from any privilege level to any other privilege level. But to switch, I need to go through several checks and balances and need to have, I need to go through a task gate. I need to go through a TSS descriptor. And, similarly for the interrupt I may handle it directly or I can go through a task gate which goes through a TSS descriptor and then it goes. It essentially performs the task switch.