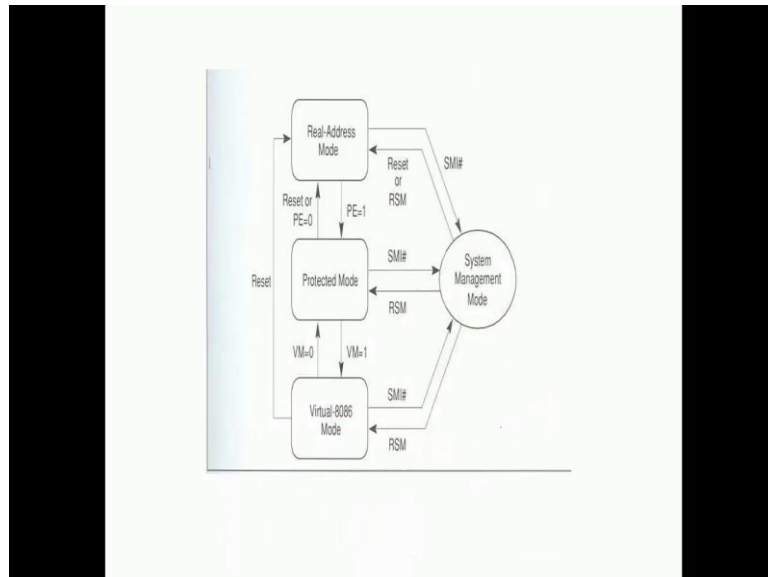


Information Security – II
Prof. V. Kamakoti
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Lecture – 11
Topic: Architectural Aid to Secure Systems Engineering
Session – 10: X86 Protected Mode Details

Good morning and welcome to the session-10, where will start looking in more detail about the x86 Protected Mode Operation.

(Refer Slide Time: 00:29)



This figure actually talks about, how the x86 processor works. When you boot the system as you see reset on your left mode side, when you reset the system, it actually goes into a mode called the Real Address Mode. The Real Address Mode is one, where you have fixed side segments, you cannot have paging and all the protection mechanisms that we are talking off are not possible. This mode was basically introduced to have some compatibility with the earlier systems like the 8086 and before, so that is a real mode. Then, if you remember yesterday's slides there was a bit in one of the controlled registers namely CR 0. If you go and set that bit to 1, it actually moves into protected mode. So, PE equals to 1, as you see in the slide will take it from real address mode to protected

mode. So, when the system boots, it is in real mode. Then the operating system or your bios or any one of this has to go and set this P to 1. So, it comes into protected mode.

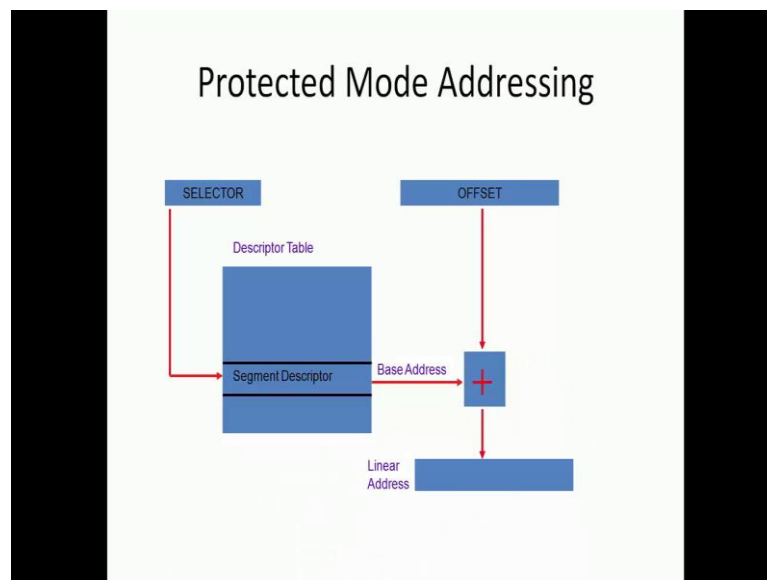
So, in my opinion all the important features that will enable us to build lot amount of security in the operating system and the compiler level are available only in the protected mode. There are lots of text books which talk about real mode. In this course, we will deal only about the protected mode and once in the protected mode, if you go and reset or again you make your bit PE is equal to 0, it goes back to real mode. Now, if you want to maintain legacy, if you want to support the past there are lot of things that you need to do. One of the other things that happened was, when I am executing in the protected mode there were some old 8086 programs that need to be executed for maintaining legacy.

So, Intel introduced another mode called Virtual 8086 mode. When you are in the protected mode and you set this VM flag to 1, you go into Virtual 8086 mode and when VM becomes 0 you go back to the protected mode. This is the simple state machine of how, the Intel architecture works in three different modes. One is the real address mode another is the protected mode and within the protected mode, I can switch to Virtual 8086 mode and come back. In addition, there is something called a System Management Mode, which also has to be looked very seriously but we will look at it in depth when we look at the operating system course.

There is something called SMI, this is a System Management Interrupt and in any one of these modes, either in the real address mode or the protected mode or the Virtual 8086 mode. If SMI is raised, the system switches to the system management mode and then you press the reset at the stage or do the RSM reset for this management, you come back to the corresponding mode from which you left. So, the system management mode is more like an interpreting interrupt service that you do to go and come back. So, there are lots of things that you can do in the system management mode but one has to understand system management mode to see that they build operating system and software much nice, so that there is no security vulnerability because of system management mode. But understanding of that would also crucial and as I mentioned the just earlier we will do it as a part of our operating system part of the information security.

In this course, we will be dealing a lot about the protected mode, Virtual 8086 mode is rarely used, real address mode is also rarely used or it is not necessary from this course perspective. So, we will deal more about the protected mode in this course.

(Refer Slide Time: 05:08)



What is protected mode? Protected itself, the adjective says that there is some protection and this is the protected mode. Yesterday, we talked about segmentation. So, what the operating system can do is basically have a lot of segments, many segments and it can give some segments to the program. So, the entire address space, you have 4 GB of address space can be split into several small, small segments and each segment could be allocated to a program. So, every program needs three or more segments, it needs a stack, it needs a code, it needs data. Then it could have some extra segments, it would have many segments at any point of time we have six segment registers. So, one register will point to the code segment, which is the executing part of program. One segment register will point to the data segment another would point to the stack. Then you have three more segments which can point to extra segments, if needed at that point of time.

Now, each segment has an attribute. What is an attribute of segment? There is a start address for the segment; there is a limit for that segment. There are many more things in that segment, we will go and deal one by one but as far as this slide, I want you to

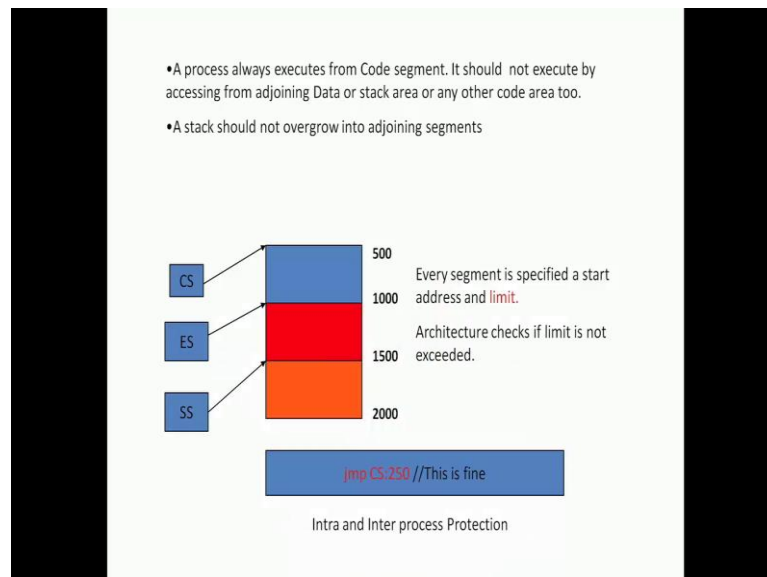
understand that every segment has certain attributes and two of those attributes are the start address and the limit. So, for every segment we need to store these attributes somewhere and that is stored in what you call as a descriptor table. The descriptor table is also inside a memory and in that descriptor table, what we store is for every segment we store its attribute and what we call that is a segment descriptor. So, inside the descriptor table, I will be storing a segment descriptor. What does the segment descriptor have? Currently it has the base and the offset. So, yesterday when we looked at one of the assembly programs, we dealt with stack machine. We said `move 0, move EAX comma 0, move EAX comma 8` and we said that 8 is going to get added to the base of the segment, so that you can get the correct address, where the particular variable is stored. So, the base of the segment is stored in the descriptor.

Now, I have something called the descriptor table as it is shown in the slide here and what I said `ES colon 8`. What will the ES store? That register ES will store, which descriptor in the table it is pointing to. For example, as I told in the context of multi process context switching. In a particular segment register, I can make the segment register point to any of the segments, which segment I am pointing to that will be for every segment I have a descriptor in the descriptor table. So, what will ES store is, out of all these descriptors that descriptor to which I point, I will have the offset to that. So, what ES stores is a selector within that descriptor table, when I go to that index into the descriptor table I get the base address. If I have a segment that is stored at 200, in the descriptor table I will have an entry pointing to 2000 and let it be the seventh entry for example. What will the selector have? It will have the value 7. So, from the value seven, selector means whatever is stored in the segment register. So, if I say `move EAX comma ES colon 0` yesterday, that ES will have seven. From that 7, I will go into the descriptor table, I will find that the base is 2000 that 2000, I will add with that offset and I will go to the address.

When I look at a memory address, it has a segment register, colon, offset. That segment register will store the entry in the descriptor table, you go to that entry there you will get the base address, that base address you add it with this offset you will get the actual address which is there in the memory. You are getting this should I repeat. This is called a linear address because now you are looking at the entire memory as one contiguous

array as memory.

(Refer Slide Time: 10:40)



In addition, why cannot I store that 2000 itself in the ES? I do not want to store it because I want to store several things about a segment. There are lots of things that I want to talk about a given segment. In addition to the base, I want also want to store something called the limit. So, for example, there is a program that is executing as you see in the slide. There are three segments for it, one is a code segment, another is a data segment and another is a stack segment. The code segment is pointed to by CS, the data segment by ES for just for your change and the stack segment by SS.

For every segment, note that CS is 500 bytes, ES is also 500 bytes and SS is 500 bytes. So, the limit of 500 will be stored in the descriptor, along with the base the limit also will be stored. So, when I am trying to access that segment and the offset is greater than that limit, if I am going to access, say 1001 as a code. I am going to execute the instruction stored in 1001. Is it correct? It is not correct here because my code segment is only 500 bytes in length, whatever the code is, it should be within that 500 bytes. If I am going to start executing something in 1001 that means, I am trying to execute some data here, which is wrong. You are getting this? So, that limit of 500 will also be stored in the segment descriptor. When I am trying to access that is, my offset is greater than the limit

then immediately, the architecture will raise an exception saying that you are trying to exceed my limit.

It is the responsibility of the architecture because operating system cannot catch you. Operating system is also a program; it has to execute to catch you. There is only one CPU and you are executing on that. You as a program, you are executing on that and if you make a mistake, who can catch you? The operating system cannot catch you because it is also a program and it is not executing. Now, it should be a program that is executing to catch you. Operating system does not know executing, so it cannot catch you. So, the architecture takes the responsibility of catching you, who sets this limit? The operating system sets the limit and then allow you to execute then it loses control, the architecture monitors whether you as a process are adhering to whatever rules that are set by the operating system and when you do not adhere to that rule, when you violate that rule, the architecture catches you and hangs it over to the operating system. Is this clear?

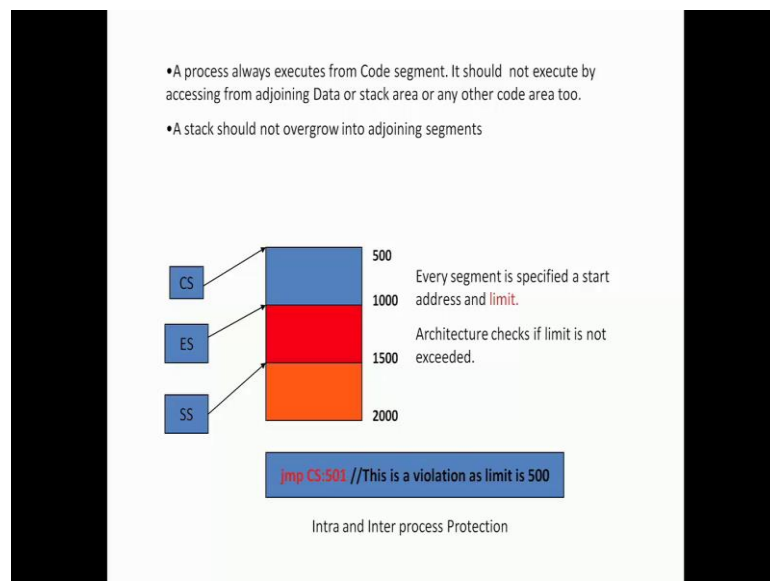
This is how architecture can help the operating system impose certain rules and regulations, which are very, very crucial for security purpose and this is exactly what is happening here. Similarly, the data is starting from 1000 and it is going to 1500. Now, when I have a limit that is more than 500, the data segment maximum size is 500. If I try to access data little above 500, what will happen? It will catch segmentation; it says there is a segmentation error, segmentation fault, seg fault. When you execute a program, this is one manifestation of that seg fault I am trying to exceed the limit that I have put here.

Similarly, the stack cannot over grow and stack is very, very important. It keeps on; you keep on pushing and popping from the stack. Stack is very dynamic, when you start having many function, your stack will grow as a function calls get executed and returns back the stack will become smaller. So, the stack will dynamically grow and it will become smaller. So, there is a growth and diminishing on the stack. So, stack is a dynamic data entity and so that is a very important reason why we should put a limit for the stack otherwise it will go and write into a data in this case. It can write into your data or it can write to something else above 2000, if it is growing down or it is growing up. So, first you know taste of security we see here, where in I am protecting my own data and my own stack from my code segment and vice versa. So, each of these, I am now by

this notion of segmentation I can create an isolation between code, data and stack. The stack cannot over grow into data or code, the code cannot go and execute something in data or stack, the data cannot write into stack or code. From each of them I am getting a mutual isolation here, are you understanding this. So, who said? Who gives you? Who allocates? You are a process; I need 500 bytes of memory, take 500 to 1000, who allocates this? The operating system allocates. The memory management module of the operating system allocates 1000 to 1500 for your data, 1500 to 2000 for your data. Who allocates all these things? The operating system allocates. Once the operating system allocates then it also puts a limit and this it stores in the memory as a segment selector in the descriptor table. It also notes your CS, ES and SS pointing to those entries in segment or descriptor table, where you have the segments described and then it allows you to execute.

Now, you cannot change these values but you can just execute and when you go and exceed the limit, immediately it catches you and gives you back to the operating system in terms of an interrupt. So, if you do this there is a seg fault that happens. The seg fault will take you to interrupt service routine and that interrupt service routine is an operating system entity. So, you are caught and given to the operating system. Here, say jump CS colon 250 is fine because my limit is 500.

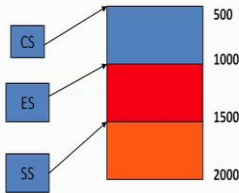
(Refer Slide Time: 17:52)



Jump CS colon 501 is a limitation because 500 is your limit.

(Refer Slide Time: 18:00)

- A process always executes from Code segment. It should not execute by accessing from adjoining Data or stack area or any other code area too.
- A stack should not overgrow into adjoining segments



The diagram shows three memory segments stacked vertically. The CS segment is blue and ends at address 500. The ES segment is red and ends at address 1000. The SS segment is orange and ends at address 1500. Arrows point from labels 'CS', 'ES', and 'SS' to their respective segments. To the right of the segments, text explains that every segment is specified a start address and limit, and that architecture checks if the limit is not exceeded.

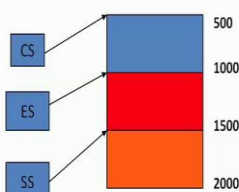
```
mov [ES:498], AX //This is fine
```

Intra and Inter process Protection

Move ES colon 498 comma AX, AX is 16 bytes. So, move the entries in 498 and now what is 498? 1498 and 1499 into this EAX register. This is fine.

(Refer Slide Time: 18:23)

- A process always executes from Code segment. It should not execute by accessing from adjoining Data or stack area or any other code area too.
- A stack should not overgrow into adjoining segments



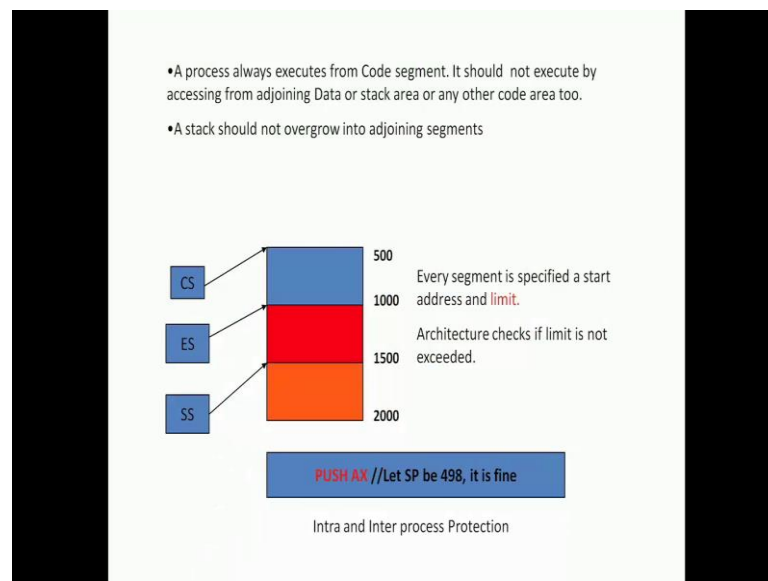
The diagram is identical to the one in the previous slide, showing CS (blue, limit 500), ES (red, limit 1000), and SS (orange, limit 1500) segments. Text to the right explains that every segment is specified a start address and limit, and that architecture checks if the limit is not exceeded.

```
mov [ES:498], EAX //This is a violation!!!
```

Intra and Inter process Protection

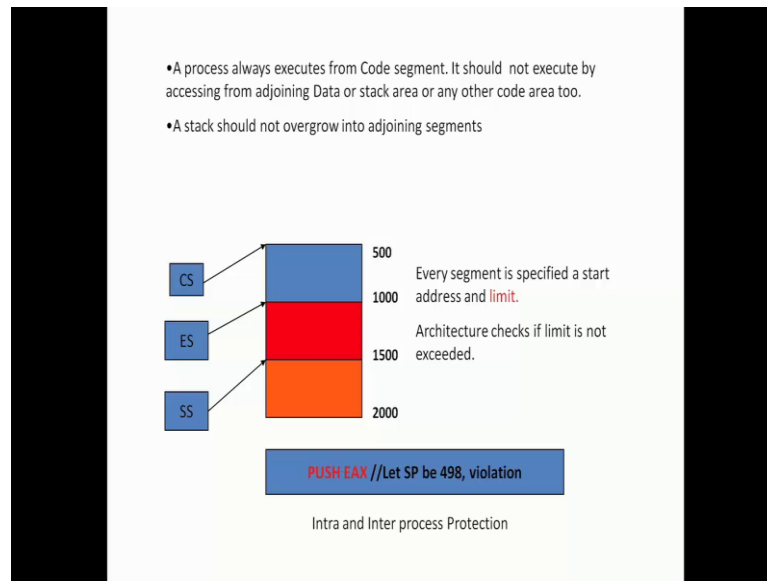
Suppose I say move ES colon 498 comma EAX, EAX is a 32 bit register. So, 498, 499, 500, 501 also I need to move. So, this is a violation. It is not just, I test the offset with the limit, I should also set from that offset how much data I am moving that maximum address also I should test. In this case, it is not enough if I test 498 with 500; it is I should set 498 plus 4 with 500. So, these are some of the simple things where you get caught.

(Refer Slide Time: 19:08).



When your stack point that is at 498 and I push AX, it is fine because it will now that stack pointer will go to 500, which is within the limit.

(Refer Slide Time: 19:19)



But when I push EAX then it is not fine because it go 498, 499, 500, 501 it will cross that. Here, in these two instructions, what we see? The explicit register is not stored, the address is not store; it is there in the stack pointer. So, that is an implicit addressing that is happening. If you read the books, there is something called implicit address, implicit because when I push, I do not have a direct reference to ESP but it uses ESP. So, the value of ESP, the stack pointer you increment the stack and push the value. That is how you do push, decrement the stack if you pop stack pointer. So, here ESP is an implicit addressing mode. So, if I say push EAX then it is going to give me a violation. Are you able to follow?

(Refer Slide Time: 20:10)

- A process always executes from Code segment. It should not execute by accessing from adjoining Data or stack area or any other code area too.
- A stack should not overgrow into adjoining segments

Every segment is specified a start address and **limit**. Architecture checks if limit is not exceeded.

POP AX // Let SP be 2, it is fine

Intra and Inter process Protection

With my SP is 2, pop AX is 5 because this will become 0 as there are only 2 bytes and it will become.

(Refer Slide Time: 20:20)

- A process always executes from Code segment. It should not execute by accessing from adjoining Data or stack area or any other code area too.
- A stack should not overgrow into adjoining segments

Every segment is specified a start address and **limit**. Architecture checks if limit is not exceeded.

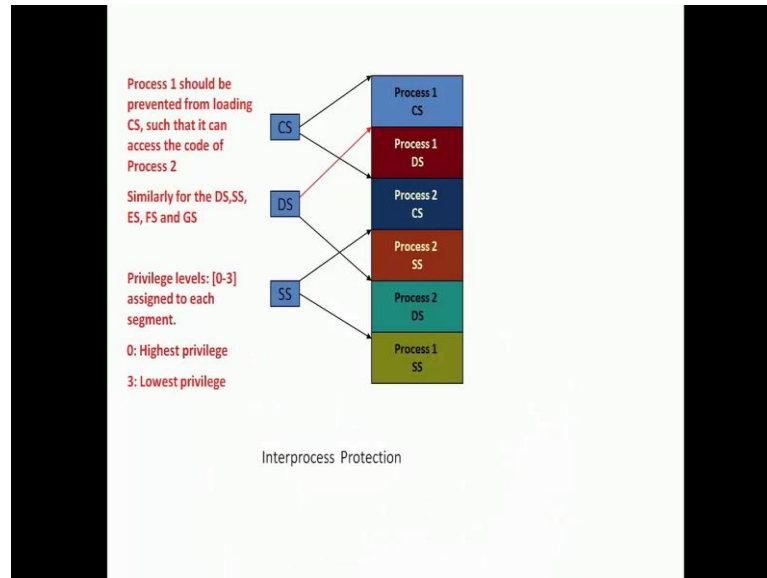
POP EAX // Let SP be 2, Violation!!!

Intra and Inter process Protection

But pop EAX is a violation because SP will now become negative 2, 1, 0 minus 1. So, again these are some of the things which are interesting about limits and always this limit

violation happens when you are very close to that limit and all these are called corner cases, where your program can fail.

(Refer Slide Time: 20:47)



So, there is a strong interprocess protection here because let me say there are two processes, process 1 and process 2 and each have their code segment, data segment and stack segment and process 1 should be prevented from loading CS, such that it can access the code of process 2. When process 1 is executing, please note that the code segment is pointing to process 1 code segment as when I am executing process 1, if I am able to load the code segment corresponding to process 2 or if I am able to load that index in to the descriptor table, which is pointing to the code segment of process 2 then I can access process 2. I can access process 2 data or whatever. So, process 1 should be prevented from loading the code segment corresponding to process 2 or loading the data segment or loading any other segment corresponding to process 2.

So, in this context I talked about intra process protection, my code and data and stack could not access between each other like I cannot execute on my data or stack data cannot write on to stack. Those things we have already seen here but in this context now I want to protect one process from another process, the code segment of process 1 should not execute code segment of process 2. So, this is called inter process protection. So,

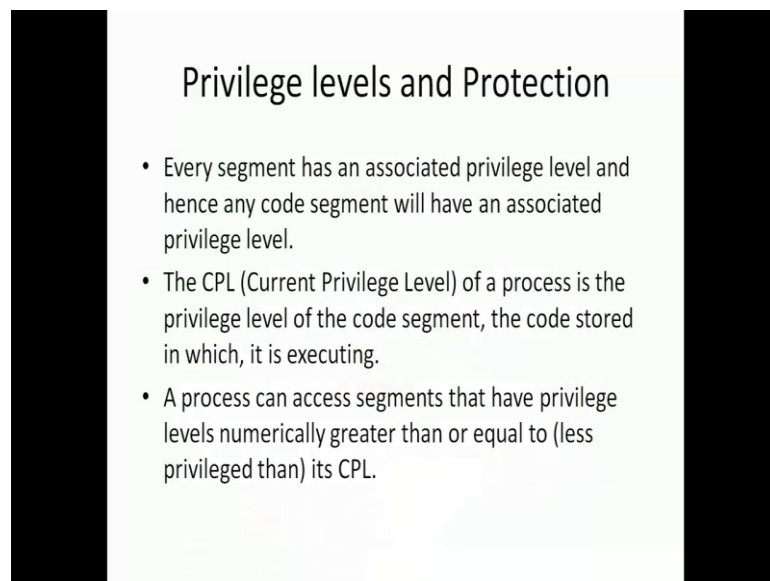
what we saw in the previous slide was intra process protection. What we are seeing in this slide is inter process protection. The way by which we are going to prevent process 1 from accessing process 2 data stack and code is by introducing something called a privilege level. There are four privilege levels in x86 architecture. What are these 4 privilege levels? Privilege level 0 1 2 and 3. The 0 privilege level has the highest privilege, it is called something like a super user privilege, the one has the next highest, two and three is the lowest privilege. So, the numerically higher the value of a privilege level less powerful it is. Please understand there is a reverse mapping here, 0 has the highest privilege, 1 has, next 2, 3. So, why did Intel or the x86 community think about 4 privilege levels? Because, they wanted the operating system to be a layered operating system for reasons of security, for reasons of security an operating system need to be layered. The innermost layer which we call as a kernel should be highly privileged, the next layer should be a little less privileged and then so on so forth. The application layer or the user layer can be privilege level three, where it has a very limited power.

When the operating systems create a segment, you are a process, you come and say I want to execute. So, it will create a code segment, stack segment and data segment, it creates a privilege level for you. So, you say I want execute, operating system will say you execute in privilege 2, you execute in privilege 3, you execute in privilege 1. So, the operating system assigns a privilege level for every process. How does it assign the privilege level? It assigns the privilege level by putting that privilege level in your segment descriptor. So, 0 is the highest privilege and 3 is the lowest privilege. In your descriptor table, when you see this segment descriptor, what is there now? There is a base address, there is a limit address and there is also a privilege level. So, when you ask for a code, stack and data, 3 descriptors are created in this descriptor table. Let me say, it is stored in the locations 1, 4 and 5. One corresponds to code, 4 correspond to data and 5 correspond to stack then the operating system. What will be stored in the descriptor table? The base of your code segment, the limit of your code segment, the privilege level of your code segment similarly, for your stack and for your data.

So, three descriptors will be created in the descriptor table, let us say they are stored at offset 1, 4 and 5 that is entry number 1, entry number 4 and entry number 5 of the table corresponds to your code, stack and data and what will be stored in your CS register, it

will store 1, it will point to entry 1. Here, in your DS register, it will store 4, SS register it will store 5. So, whenever somebody wants to use the DS register, he will go to DS segment register, there he will find 4, so you will come to the fourth entry in the descriptor table, there he will find the base, the limit and also the privilege level you understand this. So, slowly we are building up what should be inside the descriptor table.

(Refer Slide Time: 27:19)



The slide is titled "Privilege levels and Protection" and contains three bullet points. The slide is framed by two vertical black bars on the left and right sides.

Privilege levels and Protection

- Every segment has an associated privilege level and hence any code segment will have an associated privilege level.
- The CPL (Current Privilege Level) of a process is the privilege level of the code segment, the code stored in which, it is executing.
- A process can access segments that have privilege levels numerically greater than or equal to (less privileged than) its CPL.

Now, because of this privilege level what sort of protections are we going to get? Before we end up this session, I basically want to talk about what we call as a current privilege level CPL; I am a process I am executing. Where is my code stored? In a code segment that code segment has a privilege level. That privilege level is called the current privilege level. So, the current privilege level is the privilege level of the code segment that I am currently executing. So, when I am executing, in my CS if I have a value 1; that means I go to the descriptor table look at the index 1 and there will be a privilege level stored and that privilege level is my current privilege level. Who assigns this privilege level to you? The operating system assigns it to you. Now, if I am at privilege level 1, I cannot go and access any data or stack of privilege level less than one because privilege level 0 is more powerful than privilege 1, but I can go and access data or stack greater than privilege level 1. I cannot go and access less than 1. If I am at privilege level say 2, I can access 2 and 3, I cannot access 1 and 0. If I am at privilege level three that is, if my

current privilege level CPL is three, I cannot access 2, 1 and 0. I can only access 3 level segments. So, current privilege level and this explanation do not solve the problem that we have in mind. It does not solve it fully but this is certainly a building block where finally, we will have to understand how two processes are completely isolated from each other that we will understand over the next subsequent sessions. Did you follow this?

What we have done so far is that we have introduced a notion of a descriptor table, which has some descriptors each describing a segment and what is stored in your CS, SS, ES, FS, GS are not the base address of the segment but a pointer to the descriptor corresponding to the segment which it is pointing to and in the descriptor you have base, you have limit, you have privilege levels and we also introduce the notion of a current privilege level which is the privilege level of the code segment I am currently executing and depending upon the CPL. Now, I can have my entry in to certain segments my access to the certain segment can be restricted, you are getting this. So, that restriction also, we have seen as a part of the session.

Any Doubts?

Student: How the size of segment is fixed?

It is fixed by the operating system, looking at the size of your executable and data. So, if you look at ELF format, where you know you compile a program. When you look into the program there, after the compiler compiles it, it will put certain attributes of your program there. One of the attribute is the size of your code segment, data segment and stack segment. So, when you want to execute, the operating system will look at your program and then it will go and find out from the header of the program, the sizes and based on that it will fix a size of your code segment, stack segment and data segment. That is why you know, one of the reasons I know Windows program cannot execute in Linux because even this format might be different across operating system. ELF is a very known format when you are going to do your assembly language, you will use the ELF format to assemble your program and execute in the lab and you will do that. So, that is how the operating system finds the size of your particular segment when it is assigned.

Student: Yes, sir. Actually, I have one doubt.

Yes.

Student: You said, descriptor table will be in side processor that means it is a finite sized. So, does that put any restriction on number of processes that are processed?

Exactly, very nice. Yes, the descriptor table is finite in size. The reason is that, as we proceed in the next stage, number one, there are 2 types of descriptor table that we will see. One is called the global descriptor table another is called a local descriptor table. I will have one global descriptor table where I can store up to 255 entries but then I can create a local descriptor table for every process that is currently in execution. So, in the local descriptor table, I could have 255 entries again for that. So, for a given process, the number of segments that it can store is limited to 255, the 256 for the first segment descriptor is always a null descriptor. So, I can use up to 255 descriptors per process but the number of processes, I do not have a limitation because I can create several local descriptor tables and I can also do some sought of swapping in and swapping out at GDT level. So, I can keep increasing my number of processes. So, that is why there is no limit on the number of processes because of the number of descriptors I can hold because I create for every process I create another local descriptor which it can use and so I can have several such local descriptor tables.

Now, you saw one descriptor table, there will be one descriptor table exclusively for the operating system because operating system is also a program in execution, it is also a process but it is the super process or the most highly privileged process. So, it will have one descriptor table and that we can call it as the global descriptor table. Then every process you create the operating system will create a descriptor table exclusively for that process and that descriptor table is called a local descriptor table that will be in existence when the process is alive and when the process completes that local descriptor will go off. This is how; we manage the numbers that we are talking of there.

Why do we have only four privilege levels, it is something like why you are named Kamakoti? Because that day I was born, my aunt felt that Kamakoti is a good name and

she named me. Similarly, many questions of this why is very important question is that day when a designer wanted to design the system or the group which designed the system felt that four is sufficient. Like same question was asked by Prof. Kalyan Krishnan, when he taught architecture when I was a student here in 1992 and he asked why this is 32 bit? Why this is 16 bit? So, it makes lot of why's there. Then we all said that this, this. Finally, he said nothing, that day that designer wanted to do only 16 bit and that was only possible that time, so he made it 16 bit. So, many questions in architecture had these types of answers like that day it was felt that may be 16 bit was only possible. There were technological limitations also to give some.

So, this is again an interpretation, why it was 4? The reason would be that if you look at a traditional operating system that is called a micro kernel or the nucleus and then there is some layers above that which will have these device drivers, etcetera and then top of it there will be one more layer which have all the system development software's like, compilers and all development environment and top of it there is going to be application and all these 4 have to run at different points of time. So, when you look at an operating system, when we classify the processes that are running on an operating system are based on these privileges. We see there is a need for 4 privileges, I cannot think of something larger than this as of now, that precisely also can be the reason why people look that four privilege levels. So, this is a retrofit interpretation or justification, may be it is true or may be not but this is an answer, the best answer that I could give now, any other doubts?

Student: Address.

Yes, please.

Student: The address, whatever the address we were discussing. Can we use it in the virtual address page?

No, it is in the real that is; we have not still brought in virtual address memory at all. So, this is actually the physical address, we have not introduced paging so far.

Student: When we are talking about the process, it is at the operating system level.

No. No, still we are not talking about the process. We are now talking about a process which is working in an operating system, which has not enabled virtual memory, which is now only working with segmentation. We will introduce virtual memory later but this is not the final environment where the process works. But now, today when I am talking of this session I am not still brought in paging. We are now looking at an operating system; a fictitious operating system may be, which is working only in protected mode no paging, no virtualization, no virtual memory already introduced. So, now, we are looking at real RAM addresses. So, what is stored as a base address in the segment descriptor, it is a real address in the RAM, it is directly mapping on to the address in the RAM. The offset that I have put also this plus offset will directly map in to the RAM. Now, there will be one more stage that is introduced after this translation and before going to the RAM that is the virtual memory, which we will deal in the next session, some sessions later.

Now, what we have discussed so far is only about protected mode, no virtual memory executions. This linear address that you see on the slide here is the actual address on the RAM. If I say 1004, it is 1004 on the RAM, this is directly sent to the memory address register in your RAM. Now, when virtualization virtual memory is introduced, this linear address will undergo another translation and it will go to the RAM. That part, I am going to introduce a little later. So, the processor can work in 2 modes, the Intel processor can work in 2 modes, one without paging and one with paging. In the without paging mode, this is how it works. In with paging mode, there will be another one more translation that happens after the linear address before it goes to the memory.

So, I hope I have explained what we are trying to do.