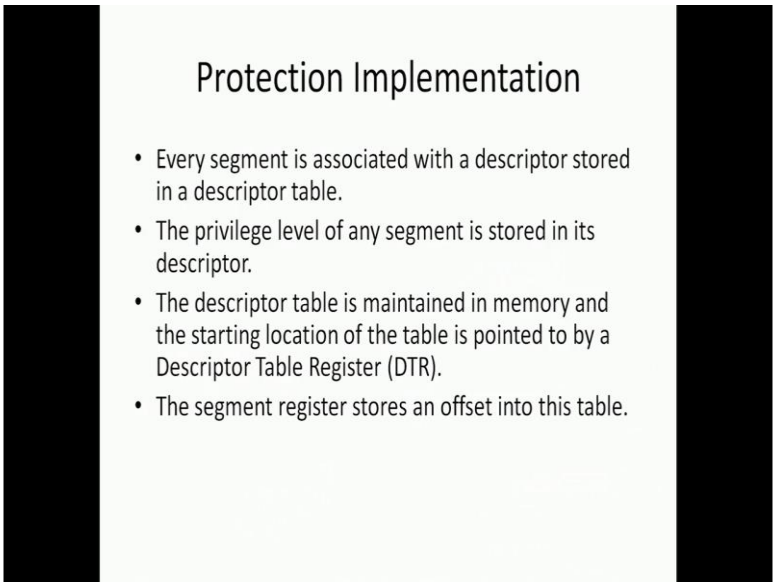
Information Security - II

Prof. V. Kamakoti
Department of Computer Science and Engineering.
Indian Institute of Technology, Madras

Lecture - 12
Architectural Aid to Secure Systems Engineering
Session – 11: X86 Memory Segmentation

Now, we move on to the next session, again on X86 Memory Segmentation.

(Refer Slide Time: 00:18)



Protection Implementation

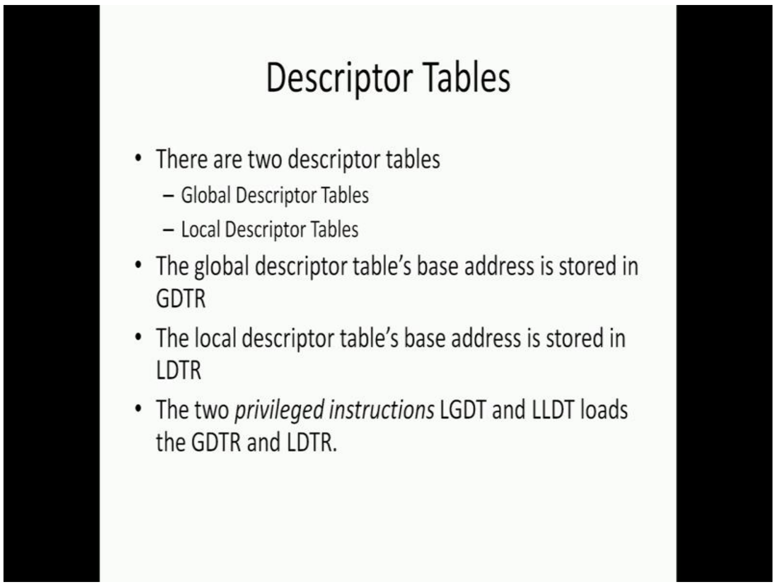
- Every segment is associated with a descriptor stored in a descriptor table.
- The privilege level of any segment is stored in its descriptor.
- The descriptor table is maintained in memory and the starting location of the table is pointed to by a Descriptor Table Register (DTR).
- The segment register stores an offset into this table.

First thing is, how do we implement protection? I am going to read out these 4 points very slowly and I will read it out twice by the time you will also understand what it is. Every segment is associated with a descriptor stored inside a descriptor table. This we have explained, I hope there is no doubt. Again I will repeat, every segment which is going to be created in the memory will be associated with 1 descriptor stored in a descriptor table and the privilege level of any segment is also stored in his descriptor.

Now, by these 2 statements what happens is, every location in your RAM has a privilege level associated with it, because every location in your RAM is associated with a segment here and every segment has a privilege level this essentially implies that every usable segment in your RAM that is currently being used is already associated with the privilege level. So, like this I can get very, very you know low granular, fine granular privileges assigned. Every byte in your memory, I am in a position to assign a privilege level through this mechanism.

Now, this entire bunch of descriptors are maintained in the memory itself in a descriptor table and the start of this descriptor table, where will the descriptor table start that is stored in a register called Descriptor Table Register, because when I say select R number; R means, I say go and get the 5th selector in your descriptor table. I should know where the descriptor table is starting then only I can go to the 5th entry in that and pickup. So, where will I know where the descriptor table is starting in what you call as the Descriptor Table Register. And, the segment register what does it store it stores an offset into this table. The segment register only selects the descriptor, so the segment register is also called as segment selector got this. Now, when we look at the global picture you will understand why this level of complication are, if at all you feel it is complex why this level of complication is necessary.

(Refer Slide Time: 03:09)



Descriptor Tables

- There are two descriptor tables
  - Global Descriptor Tables
  - Local Descriptor Tables
- The global descriptor table's base address is stored in GDTR
- The local descriptor table's base address is stored in LDTR
- The two *privileged instructions* LGDT and LLDT loads the GDTR and LDTR.

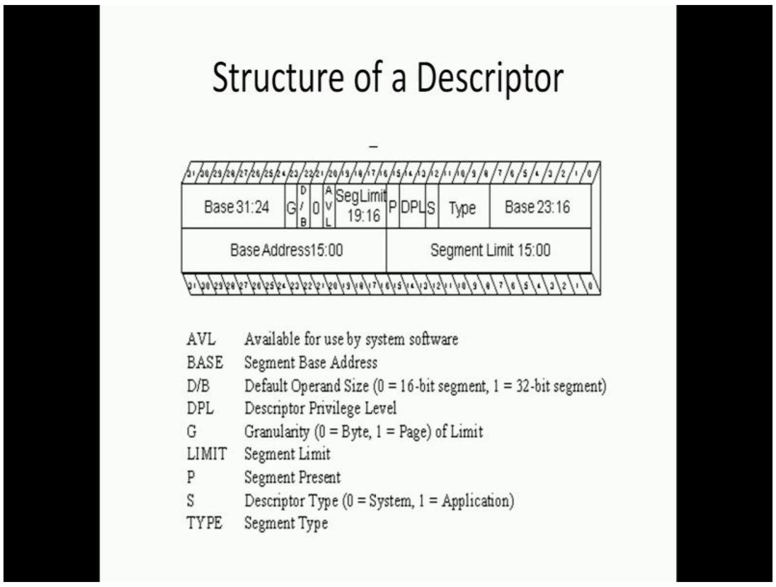
There are two descriptor tables, one is a Global Descriptor Table another is a Local Descriptor Table. Already we had some discussion on, why we need a global descriptor table? Why we need a local descriptor table? But we will elaborate on this as we go. So, if I have a global descriptor table I have a local descriptor table there should be registers which point to the start of the global descriptor table, there should be register that points to the start of the local descriptor table. Essentially, I have 2 registers GDTR which is

Global Descriptor Table Register and LDTR which is Local Descriptor Table Register.

For our understanding, there is one global descriptor table which is used by the operating system. The GDTR will point to that table as and when a process is created a local descriptor table is created for it. So, every time a new process is created the LDTR will be changed to point to that local descriptor table. Let us say you 2 are 2 processes, process one has it is local descriptor table at 5000, process 2 has local descriptor table at 10,000, when you are executing the LDRT will point the value of the LDRT will be 5000 when you are executing the value of LDRT will be 10000. Now, how will I load into this GDRT and LDRT? Please note we are building this security very very slowly. How do I load the value into this LDRT, GDRT? I have to load 1000 I have to load 5000, so I use 2 instructions namely load global descriptor table register which is called LGDT and load local descriptor table LLDT. I use 2 instructions namely, LGDT and LLDT. Now please note, this is very very important the LGDT and the LLDT instructions are privileged instructions, that means only a code executing in privilege level 0, in our case, the operating system has the permission to basically load into this global descriptor table, load into this local descriptor table.

A privilege level 3 code cannot load into the global descriptor table; a privilege level 3 code cannot load into this global descriptor table register. So, when I want to create a GDT, who can create this GDT? Only a privilege level 0 code can create it. For the simple reason because the base address for this LDT or the base address for this GDT can be loaded only by the privilege level 0 code, because we have to use these 2 instructions LLDT and LGDT to load into those registers and that 2 instructions LGDT and LLDT are what privilege instructions and they can be executed by only privilege level 0 code. So, nobody other than the operating system can create a GDT or an LDT. Even if I create a GDT or LDT, I cannot make the LLDT and LGDT to point to that, the GDTR and the LDTR to point it. If I cannot make them point to it then I cannot access, suppose I create descriptor table and I store it at 1000 and now I say in the ES 5, that means I should take the 5th descriptor in my 1000 I cannot take it unless my descriptor table register points 2000 and I cannot load 1000 into it unless I have privilege level 0.

(Refer Slide Time: 07:23)


Structure of a Descriptor

AVL	Available for use by system software
BASE	Segment Base Address
D/B	Default Operand Size (0 = 16-bit segment, 1 = 32-bit segment)
DPL	Descriptor Privilege Level
G	Granularity (0 = Byte, 1 = Page) of Limit
LIMIT	Segment Limit
P	Segment Present
S	Descriptor Type (0 = System, 1 = Application)
TYPE	Segment Type

You are able to follow what I am trying to say. Now, we will next go and this is something again if you ask me a why the answer is this is how you know Intel and AMD engineers found this. Now your descriptor table what will the descriptor table contain descriptors, what will be each descriptor. As such as such I told you the descriptor will have 3 things as of now, what are those? The base address, the limit and the privilege levels right now it as much more here, let us now see.

Each descriptor is actually 8 bytes in size. You see there, 0 to 31 in the bottom and 0 to 31 in the top, so there are 8 bytes; the first byte. So, if you say this descriptor starts at 1000 it will go from 1000, 1001, 1002, 1003, 1004, 1005, 1006, 1007. In that 7 bytes, this descriptor itself will be stored. So, 1000, 1001, 1002, 1003 are the lower half, 1004, 1005, 1006, 1007 are the upper half, that is how it will be stored. The lower 4 bytes of this 8 bytes are in the bottom and the upper 4 bytes are in the top. So, in the lower 2 bytes what will be stored, the first 16 bits of your limit and the first 16 bits of your base will be stored. In the upper half, the first 8 bits of your base will be stored in 1 corner, the remaining 8 bits will be stored in the other corner. It is a complicated instruction at architecture. You do not ask me why, there may be some reason I cannot find that reason. This is how it is stored. So, this is how the designer felt it should be.

There are 32 bit for the base starting of any base address should be 32 bits, so the 4 bits the first bits of the base is stored on the bottom upper half another 1 bite on the upper in this part base 23 colon 12 that you see on the top and the remaining here. The limit as I told you is 20 bits, so I could have 1 MB segments to start with we can extend it to 4 GB I will tell you later, but that 20 bits are stored as 16 bits in the bottom and there is some small 16 to 19 in the top; you see seg limit 19 colon 60, so the 20 bits are accommodated like that. Then, there are several bits. We have finished of segment limit, we have finished off base. Then please note that, the 21st bit on the top is 0 always it should be always 0, why I will tell you later.

Then, there are other types of bits that we will example now. AVL that is the 20th bit, it is available for use by system software. So, the operating system can use it for different purposes it can color segments, etcetera. It can do a red, black coloring of the segments so many things are possible, but we will discuss something about this when we do the operating system course. But as of that that is available, so when the operating system is creating this if it has no purpose for it can put any do not care, any 0 or 1 it does not matter. Then the next thing is D slash B bit. This is default operant size, if it is 0 then it is 16 bit segment, 1 then it is a 32 bit segment. So, what happens is? Why we have this? We have this because Intel and AMD had this legacy at to support legacy, so it has to support 16 bit code because 80, 86 was 16 bit architecture, then we moved on to the 30-32 bit architecture. So, there is legacy. We have to support execution of the 16 bit code. If I have a 16 bit code I go and make this D slash B bit as 0 correct, then what will happen 16 bit code will get, so all the code that I see there all the operands, I see will be treated as a 16 bit operand. If I make it 1 then it becomes 32 bit operand. So, if I have a code that is compiled for 16 bit then that code segment that D slash B bit should be made 0, then automatically it will start executing as a 16 bit. If I make it 1 it is 32 bit, so that is a D slash B bit.

Now, there is a Granularity bit, G bit if I make it 0 then that limit 20 bits I have. Yesterday, I explained in one of the session that 20 bits, each entry suppose I store in this 20 bits 1000, the limit is 1000 and my G bit is 1, right? Then what is this? It will be 1000 bytes, but if I store G bit as 1 then each entry is equivalent to 4 KB. If I put 1000 there and my G bit as 1 it becomes 4000 KB, you are getting this? So the G bit will determine what each entry in the limit essentially means, what is the value? What is the weight of the entry in the limit? If my G bit is 0 and I put 1000, it essentially means 1000 bytes, if the G bit is 1 and I put 1000 then that 1000 essentially means 4000 kilo bytes. So, by this

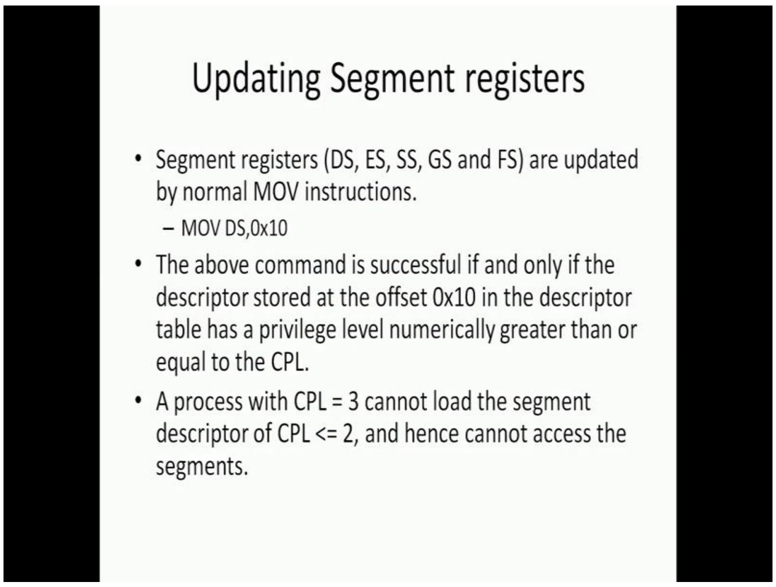
I could get a segment which is 1 MB in size, I could also get a segment which is 4 GB in size, because the maximum limit I can put it as 2 power 20 and then I put the G bit 1 then it is 2 power 20 into 2 power 12, which essentially becomes 2 power 32. There is a segment limit.

Then there is P bit which is called the Present bit. When I create a GDT and I only fill up 3 or 4 columns, I fill 3 or 4 entries. What will be every entry; every entry will be a combination of 0's and 1's. So, every entry can become valid entry. Some where I should say that this a valid entry and this an invalid entry, because if I create a GDT I can store up to 255 will see why, so around 255 descriptors I can store, but I will not fill up all these 255 descriptors I will fill up only 4 or 5 descriptors. The remaining all the descriptors will not be valid, but in the RAM I will have some random values there and each can become descriptor by it with that base and that limit, correct? Do you understand? what I am saying? Yes or no. So, for that what we do? There is 1 bit called Present bit, if that bit is 0 that means the segment is not present, but that the entry is invalid. If that bit is 1 then only that entry is valid. Once I create a GDT, the operating system creates GDT or a LDT it has to go through all these 255 entries, and make the present bit 0 for all so that no entry is filled everything is invalid. Then it fills up one by one and makes the present bit 1; so any structure. So this is 1 principle of architecture.

Any structure that you create in an architecture where it can be partially full, you need to have for every entry whether it is a valid entry or not. Can you give me another example? Cash if I create a cash, cash it can be full it can be half full. So there will be some entries there will be some (Refer Time: 16:24). There is a present bit, valid bit and an invalid bit in cash for example. So, like that we will have a translation look as it buffer many things. All the things where it can be partially full, there should be 1 bit which essentially says which essentially says which is valid or invalid and this P bits (Refer Time: 16:40) let that. There is a DPL which is that 13 and the 14 bit on the top which gives you the descriptor privilege level. I told you right everybody will be given at privilege that DPL gives you the privilege and there are 4 privilege levels so I can have 0, 1, 2 and three. Then there is S bit which says that this descriptor belongs to your Application or this descriptor is a System descriptor. So, there is a classification variety of descriptor, what is a system descriptor? What is an application descriptor? We will see in great detail as we go, but as far as now we have seen 3 types of descriptors already a code segment descriptor, a data segment descriptor and stack segment. These are all belonging to the application. For those descriptor this S bit will be 1. When it is 0, it actually belongs to a system descriptor.

As we proceed I will tell you many more about system descriptors, but just to tell you the LDT is a descriptor table that is described in the GDT. I have a process, I create the LDT; by having an entry in the GDT which will point to an LDT. So, the describing of the LDT itself will come through a LDT descriptor and that will be example of a system descriptor. We will talk about that as we proceed in this session. And then, there is a TYPE, Segment TYPE which is 4 bits in length. We will elaborate on the Segment TYPE also (Refer Time: 18:27). So, every descriptor that is put on the descriptor table essentially has all these entries in that.

(Refer Slide Time: 18:39)



Updating Segment registers

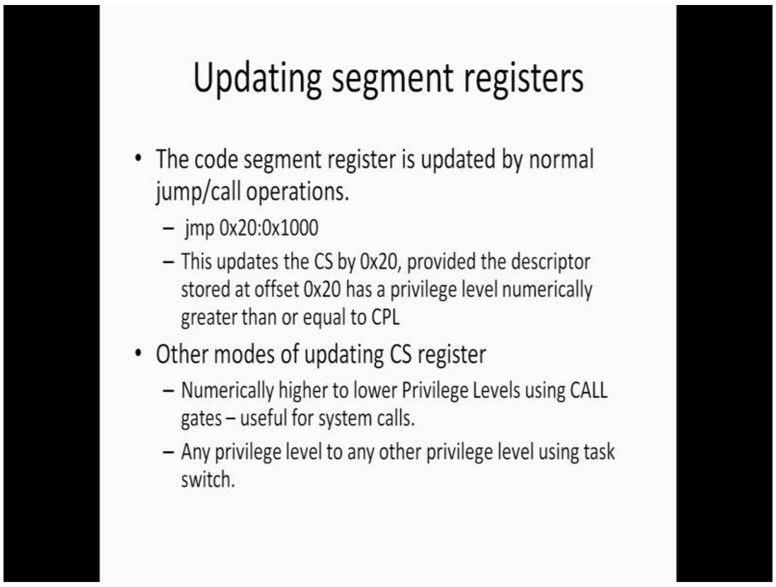
- Segment registers (DS, ES, SS, GS and FS) are updated by normal MOV instructions.
  - MOV DS,0x10
- The above command is successful if and only if the descriptor stored at the offset 0x10 in the descriptor table has a privilege level numerically greater than or equal to the CPL.
- A process with CPL = 3 cannot load the segment descriptor of CPL <= 2, and hence cannot access the segments.

Now, how do we go and load these descriptors? I go and load this descriptor by a simple MOV statement. MOV DS comma 0x10, what does this mean?

Yes, please answer, what MOV DS comma 0x10 means? It is 10; it is hexadecimal first, 0x10 means 16. MOV the value of 16 into the DS, if this is successful then DS will point to the 16th descriptor in your descriptor table. Now, this MOV DS comma 0x10 is not just go into the DS and put the value 10, this is also important from the security perspective. This is all the architectural 8 to security, so what does the architecture do here? The above command namely, MOV DS comma 0x10 is successful if and only if the descriptors stored at the offset 16 0x10 16 in the descriptor table it has a privilege level numerically greater than or equal to the current privilege level. Now, how do I explain this statement? This is how the explanation goes, I am executing MOV DS comma 0x10. I am a process; I have some privilege level, what is my privilege level? How is CPL defined? I am executing using a code segment and that code segment has a privilege level and that is my current privilege level. So, when I am executing my code segment register should point to some code segment right from that only I can get that thing. That code segment will have a privilege level and that is my current privilege level.

Suppose, my current privilege level is say 2 and now I say go and load the 16 descriptor in the descriptor table. That descriptors privilege level is say 1, then I will be says stop we cannot. If that descriptor level is 2 I will be allowed, if that descriptor level is 3 I will be allowed, but if that descriptor level is 1 or descriptor level is 0 the architecture will give you an interrupt. It will catch and throw you to the operating system saying this fellow is trying to do something which he should not do; you go and rectify the problem. This is the architectural 8 2 security. So, if a process with current privilege level 3, that means I am executing through a code segment whose privilege level is 3, CPL is 3 I cannot load the segment descriptor of any privilege level not CPL any privilege level that is less than or equal to 2. And so I cannot access the segment itself. So if I am at privilege level 3, I can only access segments which are at privilege level 3. If I am at a privilege level 2, I can only access segments which are a privilege level 2 and 3. If I am at privilege level 1, I can only access segments which are a privilege level 1, 2 and 3. So this is how the whole thing works.

(Refer Slide Time: 22:40)



Updating segment registers

- The code segment register is updated by normal jump/call operations.
  - jmp 0x20:0x1000
  - This updates the CS by 0x20, provided the descriptor stored at offset 0x20 has a privilege level numerically greater than or equal to CPL
- Other modes of updating CS register
  - Numerically higher to lower Privilege Levels using CALL gates – useful for system calls.
  - Any privilege level to any other privilege level using task switch.

Similarly, how do I create privilege level? how do I go and change the privilege no I told you with this DS I can change DS, I can change ES, I can change SS, GS and FS using this move statement. But how will I change CS? There is no point in saying MOV CS comma something then it will go and start executing from somewhere, so the way I will change CS is by using either a jump statement or a call statement. For example, I can say jmp 0x20 colon 0x1000, this is the way I change the CS. Currently, I am executing some code segment say let me say I am executing code segment 15, what is 15 in hexadecimal 0 x f. Now, I want to go to code segment 32 which is 0x20 hexadecimal 20 is 32. Now, I say 0x20 colon 0x1000 essentially this means, this will update the CS register to 0x20 and that 0x20 will be a descriptor and that will have a base address. Let that base address be what, let it be something. It can be say some 5000, now you go to jmp 0x1000, 0x1000 is what 2 power 12, we just 4 by 0x1000 is what? Find out what is it. So, is it 2 power 12. It will take you to 4096 plus 5000 which is 9096. And this transition can happen only if 0x20 has a privilege level numerically greater than or equal to the current privilege level.

If I am at privilege level 3, I can jum to 0x20 colon 0x1000 if and only if that descriptor that is stored in the 32 30 second entry also has privilege level 3. If I am at privilege level 2, I can do this jump only if the descriptor there has 2 or 3. If I am a privilege level 3 and the descriptor that is stored at 0x20 has privilege level 1 I cannot do this jump. So, by this, what we have ensured that I can go from a lower privilege level to a higher privilege level, numerically lower privilege level to a numerically higher privilege level. But how will I come from a numerically higher privilege level to a lower privilege level; I need to come for several reasons. I am a user programmer, I am doing print F. The print F will be at a numerically lower privilege level then the, that will be in a higher privilege level than me in terms of power numerically it will be lower than me.

There are many many reasons why I need to come to a numerically lower privilege level from a numerically higher privilege level. So, for doing that there are certain mechanisms and those mechanisms give you lot of ways by which you can protect. Yesterday's stack mashing was a movement from a numerically higher privilege level to a lower privilege level without any check. Now, that type of a thing we can avoid because there are many ways by which the x86 architecture provides you ways by which you can stop a numerically a higher-level code to come to a numerically lower level code. If it has to come, then there are lot of checks that you can impose and by that you can do lot of lot more protection in gain of privilege levels.

So, this is another important aspect.