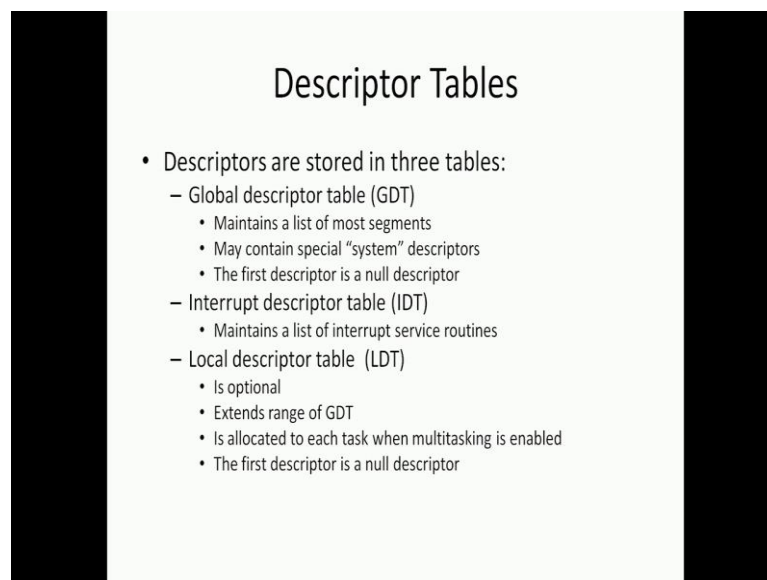


Information Security - II
Prof. V. Kamakoti
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Lecture- 17
Architectural Aid to Secure Systems Engineering
Memory Segmentation Deep Dive -2 [Descriptor Tables]

This particular session is a Deep Dive into the Descriptor Tables. So, what we saw earlier was about the Segments Descriptors. Now, we will move to the descriptor tables.

(Refer Slide Time: 00:32)



Descriptor Tables

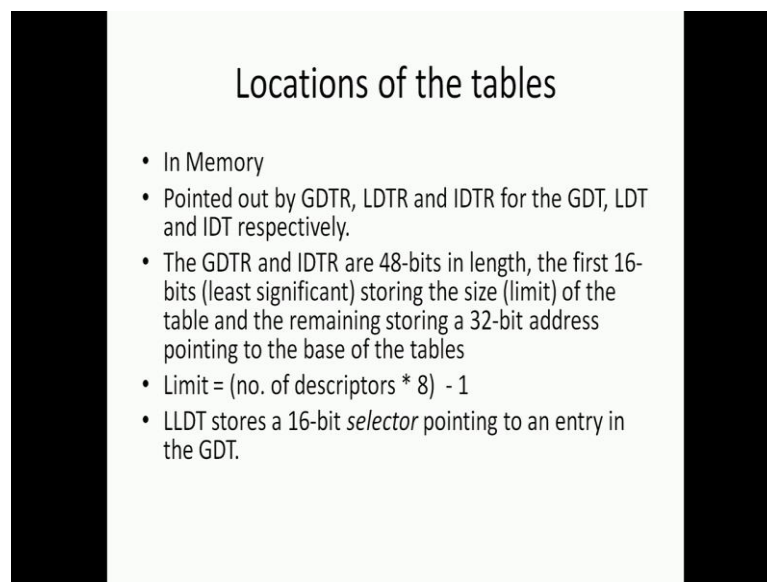
- Descriptors are stored in three tables:
 - Global descriptor table (GDT)
 - Maintains a list of most segments
 - May contain special “system” descriptors
 - The first descriptor is a null descriptor
 - Interrupt descriptor table (IDT)
 - Maintains a list of interrupt service routines
 - Local descriptor table (LDT)
 - Is optional
 - Extends range of GDT
 - Is allocated to each task when multitasking is enabled
 - The first descriptor is a null descriptor

So, there are 3 descriptor tables; the Global descriptor tables - GDT, the Interrupt descriptor tables - IDT and the Local descriptor table - LDT.

The Global descriptor tables are by default necessary. It maintains a list of most segments, it may contain certain special system descriptor as we have seen like the TSS and always the first descriptor is a null descriptor, I cannot use it for anything. The Interrupt descriptor tables maintains a list of interrupt service routines as descriptors, we already seen that. The Local descriptor table, it extends; that is, it is optional, but today

many of the operating system use it, it extends the range of GDT, right, something more than GDT I can store and it is allocated to each task when multitasking is enabled, so every task has its own LDT. And again, the first descriptor in the LDT is the null descriptor. We cannot store any value in the first descriptor in LDT, we are storing some value in the first descriptor of IDT because that is divide by 0, but here we cannot store any value in the first descriptor of a LDT; be it you know; be it whether it is a LDT or a GDT I cannot store anything in the first descriptor.

(Refer Slide Time: 01:52)



Locations of the tables

- In Memory
- Pointed out by GDTR, LDTR and IDTR for the GDT, LDT and IDT respectively.
- The GDTR and IDTR are 48-bits in length, the first 16-bits (least significant) storing the size (limit) of the table and the remaining storing a 32-bit address pointing to the base of the tables
- Limit = (no. of descriptors * 8) - 1
- LLDT stores a 16-bit *selector* pointing to an entry in the GDT.

So, where are these table stored? They are actually stored in the memory and the starting address of this is stored in the GDTR, LDTR and IDTR for the GDT, LDT and IDT, respectively. Now, what is stored in the GDTR? We have not seen that so far. In the GDTR and IDTR we store 48-bits in length. The first 16-bits, least significant 16-bits it stores the size of the table and the remaining storing at 32-bit address pointing to the base of the table. So, the GDTR essentially has as totally 48-bits; 32-bits will tell where the table starts, 16-bits will tell me what is the size of this. So, 16-bits means what? 2 power 16, I have 64 kilo bytes, but each thing is 8-bytes. So, now you calculate what will happen? With these 16-bits how much I can go 64 kilo bytes, each is 8-bytes. So, how many segments can I store? 8 kilo bytes? 16-bits is the limit, so if 16-bit is the limit; that means, I can go to 64 kilo bytes each selector is 8-bytes. So, this will be how much? 8

kilo bytes, correct? Yes or no? So, 8 kilo bytes; 8 kilo bytes minus 1 segments descriptor I can store it is not 255 only if the interrupt it is 255, but for a GDT and LDT we can go up to 8.

We can go and adjust this 16-bit. So, I need not say every time it should be by default 16-bits. So, I can go and make it say I will only store 5 descriptor or 10, I can make it even 40 10 20, right. It need not be occupying the entire 8 KB of; you are getting this. A limit field is you what you set. So, if you want to create an LDT of size 10, you go and make that limit field as what? 80, because you need 10 descriptor. So, the limit field you can set on your own, but then you have to be very carefully in setting the limit bit, so that exactly, that much is.

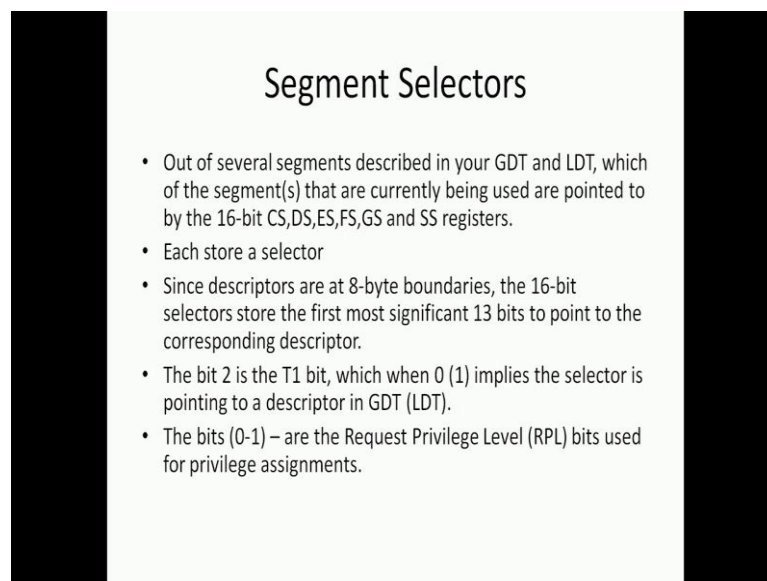
So, what is the important thing about this LDT, this limit field in both GDTR and IDTR? If I start accessing something more than that limit automatic, then there will be general protection for it and that it is very very important for us from a security prospective. I cannot go and try and access something beyond this limit, right. Otherwise, I know were the LDT is stored so I can look at some offset. So, I have 16, 8 KB thing. So, if I can get some location very close to that and put may own entry there, I am gone. So, I fixed a limit, when I create an LDT for you I show what is the base and I also fixed a limit. So, that if anybody starts accessing beyond that limit, then what will happen? there will be a fault. So, when I give an offset even for your move instructions, move DS comma some offset. That offset is checked with the corresponding LDT or GDT limit and if it exceeds that limit, there itself 1 general protection follows. Do you understand this? And that is very very crucial.

So, LGDT we are slowly going to the very tough part of this course the LGDT and the LIDT. What is LGDT? Which will the instruction that will load into the GDT? What is the LIDT? Load into the IDTR. They will point a memory location which is 48-bytes in length. So, you go there to that memory location and you find 48-bytes; the first 16-bytes load it as limit, remaining 32-bytes load it as the base. Do you understand this?

But, in LLDT what you do? It will point LLDT some selector. So, it is a 16-bits selector pointing to an entry in the GDT, right. And, in the GDT there will be a local descriptor

table descriptor, right and that will point to a start address of the local descriptor and also thing. So, LLDT alone will point to a selector in the GDT, that selector in turn will point to; will give you a start address and the; It will point to a selector, that selector will give you a start address and the limit. In the case of GDT and IDT, you point a memory location which is 48-bits in length, and there you will have what? First 16-bits will be loaded as limit and the remaining 32-bites, so that is how this works; you got it, right?

(Refer Slide Time: 08:42)



Segment Selectors

- Out of several segments described in your GDT and LDT, which of the segment(s) that are currently being used are pointed to by the 16-bit CS, DS, ES, FS, GS and SS registers.
- Each store a selector
- Since descriptors are at 8-byte boundaries, the 16-bit selectors store the first most significant 13 bits to point to the corresponding descriptor.
- The bit 2 is the T1 bit, which when 0 (1) implies the selector is pointing to a descriptor in GDT (LDT).
- The bits (0-1) – are the Request Privilege Level (RPL) bits used for privilege assignments.

So, now we will go into lot of this privilege checking things. We will again, we will be retreating several things again and again so that we have a good clarity. So, out of several segments described in your GDT and LDT, which of the segments that are currently being used are pointed to by the 16-bit CS, DS, ES, FS, GS and SS. So, in your GDT and LDT I could have n number of descriptors, I could have as much as that limit by 8. I have a limit that divided by 8 will give me the number of descriptors I should have. But, I could have more than you know 10 or 20 descriptors. As a program at any point of time I will be using one of those descriptors for code and some of those descriptors for data and one of those descriptors for stack. So, there will be one. So, I could have 10 code segments inside that LDT. I can go between 1 code segments to another.

One of those code segments will be used currently, because your CS will point to exactly 1 code segment. One of those segments will be used first stack, because SS points to one of those stack segment. Then for the data I can use any one of ES, FS, GS and DS. Now, since descriptors are at 8-byte boundaries, the 16-bit selectors store the first most significant 13 bits to point to the corresponding descriptor. Since, it is 8-byte boundary the last 3 bit are anyway 0, so the first 8 bit are stored. The bit 2. Out of the 8 bits that the 3 bits, the bit 2 is the T1 bit which, when it is 0 it implies the selector is pointing to a descriptor in GDT, when it is 1 it implies the selector is pointing to a descriptor in LDT. And then, there are 2 bits other than this which is 0 1 and they are called RPL; this is called request privilege level bits, which we will discuss about this in a greater detail when we go into task switching, etcetera.

Now, there are 2 privilege levels. For this when I am trying to see there is a descriptor privilege level, which is stored in the descriptor and there is in the selector, there is also a privilege level which I am calling as request privilege level; you are getting this? There is a descriptor privilege level which is stored in the descriptor and I am trying to access the descriptor, I using a selector to access the descriptor. The selector also can have a privilege level it is called a request privilege level. So, we will not bother about RPL today, but tomorrow we will take it up it is very interesting to note how RPL can be used.

So, lets us now go and do one more deep dive again because, we have talked lot of things in bits and pieces. So, lets us take one more deep dive and find out how we go about loading segment selectors into segment registers.

(Refer Slide Time: 12:11)

Loading Segment Selectors into segment registers

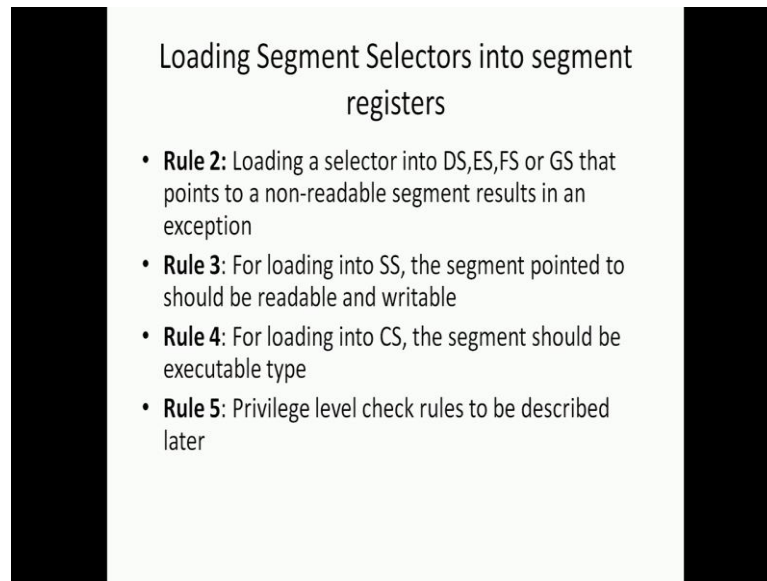
- Whenever segment registers are loaded, the following rules are checked by the processor and if violated an exception is raised thus giving high degree of memory protection
- **Rule 1:** Index field of the selector within limits of the GDT/LDT to be accessed – else raise a General Protection Fault exception.

So, we will go into these details now. So, these are all the rules that we will like to follow when we are loading a segment selector into a segment register. Whenever segment registers are loaded, how do you load a segment register? You do move DS comma something, move SS, ES, FS, GS. For CS alone you use a jump or a call. Whenever a segment selector is to be loaded into a segment register, there are lot of checks that are happening and these checks are very, very important from a security prospective. So, we will again spend some time now to again retreat on these important things.

So, I will read out this rules and explain this rules. Whenever segments registers are loaded, the following rules are checked by the processor that is by the architecture and if violated an exception is raised thus giving high degree of memory protection. Once an exception is raised your control again goes back to the interrupt service routine and hence the operating system.

So, what is rule one? The index field of the selector should be within limits of the GDT slash LDT that we are trying to access. Otherwise, you raise a general protection fault which is some fault number 30 you see. So, where do I set the limit for this? When I do my LIDT or when I do my LGDT or LLDT, I set the limit that so many descriptors alone can be there; if I exceed that then there is a general protection fault.

(Refer Slide Time: 14:27)



Loading Segment Selectors into segment registers

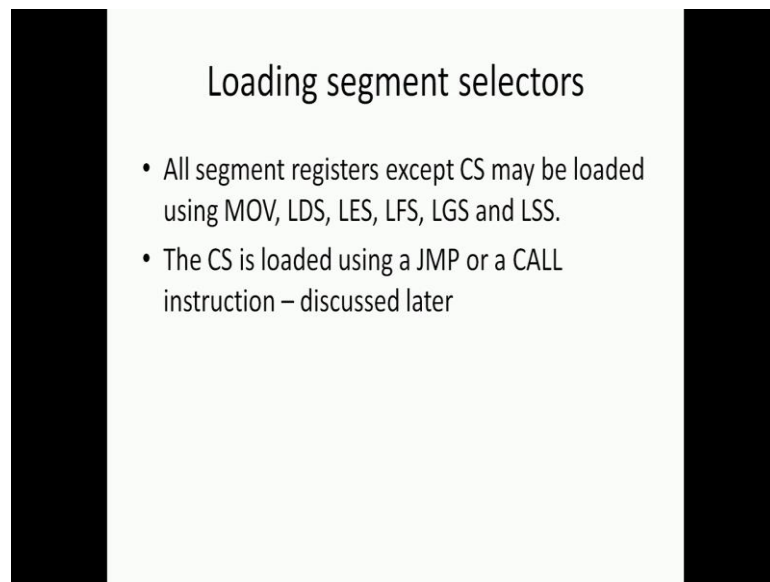
- **Rule 2:** Loading a selector into DS,ES,FS or GS that points to a non-readable segment results in an exception
- **Rule 3:** For loading into SS, the segment pointed to should be readable and writable
- **Rule 4:** For loading into CS, the segment should be executable type
- **Rule 5:** Privilege level check rules to be described later

Rule 2, when I am loading a selector into DS, ES, FS or GS that points to a non-readable segment, it results in an exception, right. So, when I am loading what are non-readable segments? Execute only segments. So, I have different types of segments here, I will cover that in the; I think I have done something in the previous things. So, as you see here suppose I have 1 0 0, this is a code execute only segments. 1 1 0 it is a conforming code execute only segment, you are getting this. So, these segments if I start loading it in SS, DS, ES, FS, GS I cannot even read, not even write. So, if I have an; I cannot even read it, it is a only execute only segments. So, I should not be in a position to even read it, correct. So, if I have an execute only segments, so this basically explains this; so, when I am loading a selector into DS, ES, FS or GS that points to a non readable segments that immediately results in an exception. For loading into SS, the segment pointed to should be not only just readable, it should also be writable.

So, it is not just non-readable. Non-readable is not just readable alone, it should be first readable and also writable otherwise, it will give an exception. So, please note, that there are lot of checks the architecture does before it allows you to start using a segment and if you rigorously follow these rules your operating system has to be secured. The main point, why today there is vulnerability is many of the operating system have forgotten, this rules are not used this rules, correct. You follow what I am trying to say, right. Now,

the rule 4 is, when I am loading into CS the segment should be an executable type segment. That is executable, executable read. Non-conforming executable and non conforming executable read; but it should be a executable segment. So, if I write a data segments into CS then automatically it will give you a fault and then the privilege level check rules are to be describe later, we will describe that in more detail, but already we know that if I am at a privilege level k; I can load only my current privilege level is k, I can load only something greater than k.

(Refer Slide Time: 17:32)



Loading segment selectors

- All segment registers except CS may be loaded using MOV, LDS, LES, LFS, LGS and LSS.
- The CS is loaded using a JMP or a CALL instruction – discussed later

So, all segment registers except CS may be loaded using move, move instruction; MOV DS comma something. Or, it can be using LDS - the load data segment; load ES, load FS load GS, LSS these are all some of the things that are present today. And, the CS is loaded using a JMP or a CALL, we have already described that in the previous notions and now we will discuss that in a much more boarder detail when we do the assignments.

(Refer Slide Time: 18:10)

Local Descriptor Table

- Is defined by a system descriptor (S=0) in GDT which is pointed to by the LDT.

The 64-bit descriptor in GDT

Base Address	0000	Limit	p	0000010	Base Address	Limit
31-24		19-16			23-0	15-0

Now, what is this Local Descriptor Table? I said LGDT and LIDT, I told that it will; LGDT some memory location. In that memory location what will be there, it will have the 48-bit address, similarly, LIDT; that 48-bits means, 16-bit for limit and 32-bit for the base. But, now we look at LLDT. LLDT will point to some selector inside the GDT, right. What is that selector? That selector will have a 32-bit base as you see and that 20-bit limit with a present bit and there is some bit pattern 0000010, this is how that 64 bit is organized. The 0000010 is nothing but a system bit, is system bit and the type bits. So, this is a system descriptor, unlike a user descriptor that we have seen in the past. This is the system descriptor and it has a type and that will be the type and then there is the present bit and then some bits are hard coded to 0. So, this is how a 64 bit descriptor in LDT, of a LDT will.

So, what this will give you? It will give you a base address plus it will give you a limit which is 20 base. It will give you a base address, which is where the LDT will start and it will give you a limit which is 20 bits here; 20 bit means, I can go up to 1 mega byte; 1 mega byte and each is 8 bytes, so 1 made by 8 what? I could have 256 kilo bytes sorry, 125 kilo byte, 125 into 10 power 2 power 10 LDT entries; LDT entries. So, this is how LDT goes.