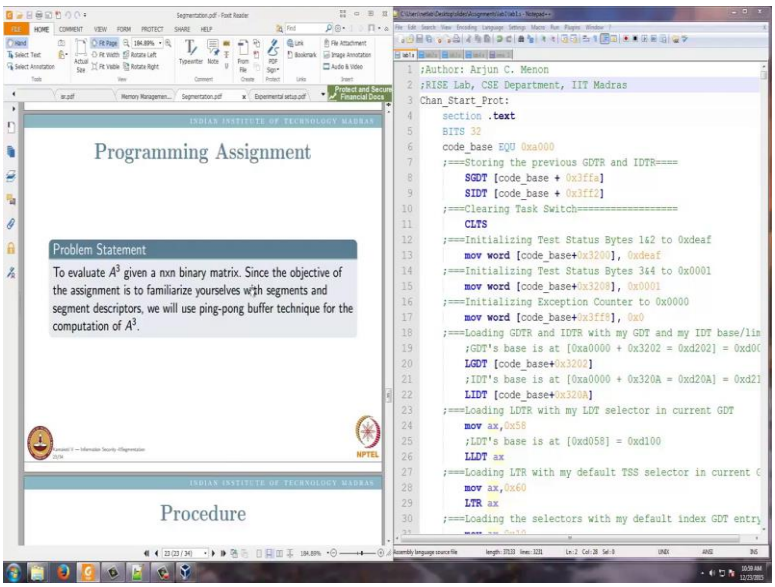
Information Security - II
Prof. V. Kamakoti
Department of Computer Science and Engineering
Indian institute of Technology, Madras

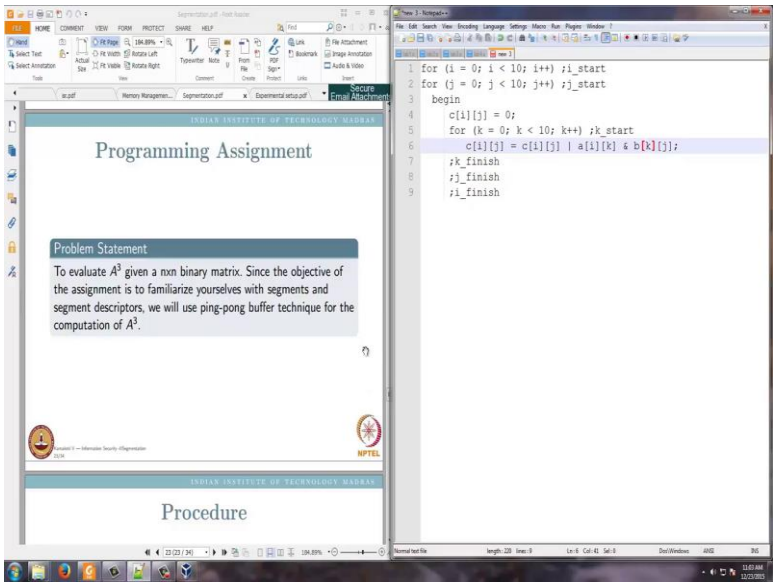
Lecture - 22
Lab1 Part 2 - Week 4

(Refer Slide Time: 00:09)



This is the first assignment that we need to do and in this assignment I am going to teach you a way of writing assembling code and also segmentation. So, let us now start with a programming assignment, where I am going to multiply 210 cross 10 matrices, where each element is a byte. Actually both that matrices that I am multipling are the same matrix. So, I am calculating A square then I will calculate A cube. I will take a matrix 10 cross 10 matrix, which each elements being at byte. I have 100 bytes, I multiply that the matrix with itself, so I get A squared, again I multiply A squared with A to get A cube. And this multiplication is not the normal multiplication.

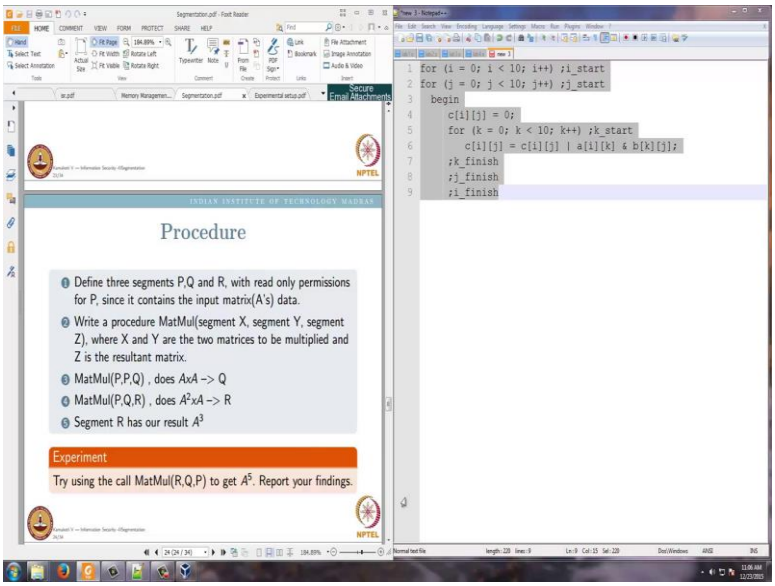
(Refer Slide Time: 01:15)



How do you multiply 210 cross 10 matrix? For i equal to 0, i less than 10, i plus plus, for j equal to 0, j less than 10, j plus plus. I am writing simple c code, c i j equal to 0 for k equal to 0, k less than 10, k plus plus. c i j equal to c i j plus a i k into b k j. Let me just put some labels here, here your k will finish then your j will finish and then your i will finish. Your i will start here, your j will start here, the j loop I mean and your k loop start here.

To simplify our complete code in this case what I am going to do is, I am going do r and then n. So I am going to binary operations here. so instead of plus I am just going to do a bit wise r of the bits. Note that c i j, a i k and b k j all these are byte, these all are 8 bits. So, I am going to do bit wise r of c i j with the output of the bit wise standard of a i k and b k j, this is the code that I want to execute. Now, your A matrix will be stored in one segment, B matrix will be stored in another segment. Your c matrix will be stored in other segment and then we will also use functions calls, basically to demonstrate, how the stack works and how your calling routine communicates with the call routine. As I told you one of the earlier sessions that this had been one of the major vulnerability, so let us understand through assembly programming, how passing of parameters actually work.? We will write a code, where we write an assembly code through which we actually pass parameters, so that we have a real hands-on experience here. So, this is the code for which I am going to write assembly code here and we will now go through the assembly code in detail.

(Refer Slide Time: 04:42)

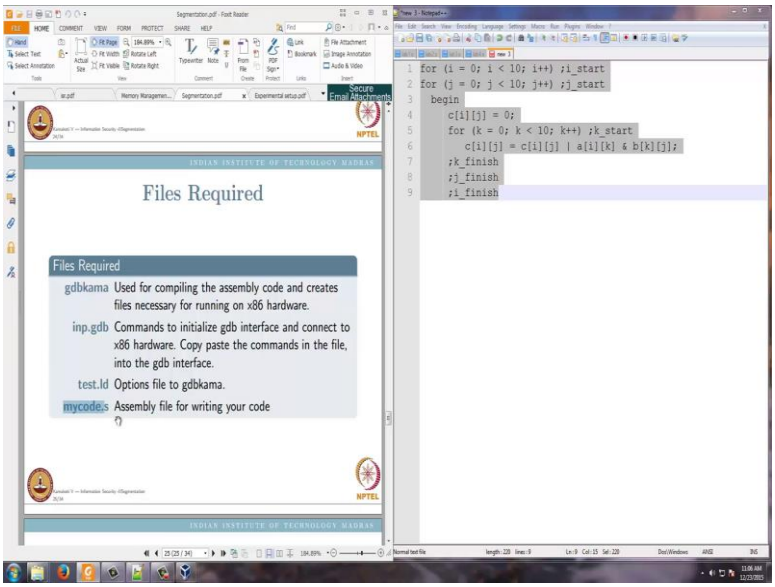


As we had seen in the directory, the procedure is we will define 3 segments P, Q and R. Initially, P will be storing A, now we will multiply the A stored in P with the same thing to get A square which will be stored in Q. So in the segment Q, A squared will be stored. Again will take the A square in Q and A in P compute A cube and store it in R. Just make things little bit to understand, what we have learnt in the theory class, we will make this P as a read only segment, while Q and R as the read write data segments. And every time I do this matrix multiplications I call a subroutine which will take 2 segments as input and 1 will be the source segment the another will be the, 2 source segment as input and it will fill the values in the destination segment. So, this is how I will basically do a function call here.

Normally, when we want to multiply these 2 matrices as a, suppose I define a function for this, I will take the A and B matrix as inputs. Meaning I will take the address of starting of the A and B matrix as inputs here, and I will also take the address of the starting of the C matrices inputs. I will do these operations and store the value of C there in the address that is provided, that means, what I mean by address of A and B matrix? Basically, I am taking the details of the segments, where A and B are stored and also where I should store C and I will store it back. So this how if I want implement these entire matrix multiplication as a function call and I call it from main routine, so the way the parameters are typically passed would be through these addresses and this is what we will be implementing here.

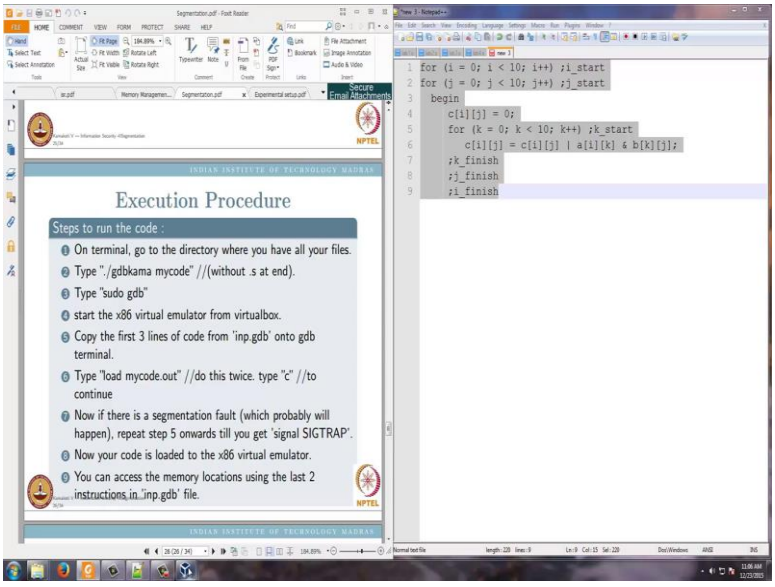
The notion of using call as a repeat will be to basically explain how the stack works because we have already looked at stack machine where we explain, how stack works we will see how it is doing with a practical demo here.

(Refer Slide Time: 07:20)



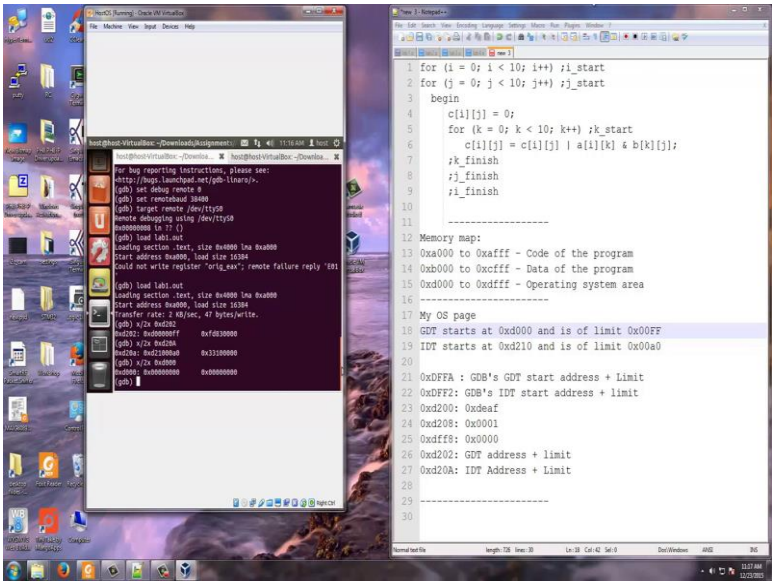
As we have seen there, we need the gdb kama which is there in the directory, we need inp dot gdb, which are the commands, the test dot ld which are the loading options and today we have lab 1 dot S. This is typically what will be using for other assignments, it will be lab 2 dot S, lab 3 dot S, lab 4 dot S and so on.

(Refer Slide Time: 07:42)



The execution procedure we have already seen this, so you can go through these slide again, if any doubt, I have already explain that in (Refer Time: 07:54) detail.

(Refer Slide Time: 07:59)



.

With this let us again go back. Now, we have loaded the entire code here, so let us start. If you open lab 1 dot S, what you see on the right hand side is the thing that you will see. So this is the code, note that there was a dot text. In the pro dot l d file I am going to the next here so more broad s dot l d. There was a dot text and we said load the dot test in 0x8000 that is what you see here. The dot text in the file as you see in your right hand side here, the dot text will get loaded that 8000

Now what are we doing here? So, there is a GDT already running, so that will have its own GDT and IDT. Note that we are currently in provision level 0. We go and store those values of GDT and IDT in code base plus 3 ff , what is code base? 8000, 8000 plus 3 ffa is dffa. So, in dffa we will store the old values of GDT and DFF2 we will store the old values of IDT. What will this GDT contains? It will contain a 16-bit limit and a 32-bit starting address of GDT. We will just a start making note here, so I start from here again in 0xdffa, you are in line number 8 here 0xdffa we will be storing old value old GDT, GDT start address plus limit. Similarly, 0xddf2 will be storing GDT, IDT's, IDT start address plus. Note that these 2 commands can be executed. What is difficulty in executing these commands, note that SGDT and SIDT are privilege instruction and they

can be executed only with privilege 0 code. But when I am starting executing of this code this is in privilege 0, so this will execute these 2 commands for sure. Then we clear the task switch, then these are all something basically to tell that this code is executing. So, 8000 plus 3200 is d200, I store the value deaf, so 0xb200 I store the value deaf. And in d208 I store the value 0001.

And in the location dff 8, I store the value 0 this is line number 70. So these are all 8 bit movement, sorry 4 byte movement. This is word is basically 16-bit that is 2 byte movement, so you may get as 000. After that I load the GDT of mine, the GDT of this code. So, the lab 1 dot S to basically explain you what is lab 1 dot S. Basically, all the code that will have as a part of this assignment, this is the memory map of our code 0xa000 to 0xafff we will have the Code of the program. 0xb000 to 0xcfff we will have the data of the program. And 0xd000 to 0xdfff will be the Operating system area. So when we load the program, we have a small Mini Kernel also loaded. Note that now this memory map you will follow, note that all that you are doing here so far like storing the old GDT, IDT, d20 all these thing are in the page d000 to dff, which is the operating system area. Now, what next step is, the operating system actually stores my address at address my GDT so for executing my code because I will have my own segment, I love I need my own GDT show the 0xd202 will store the GDT address plus limit.

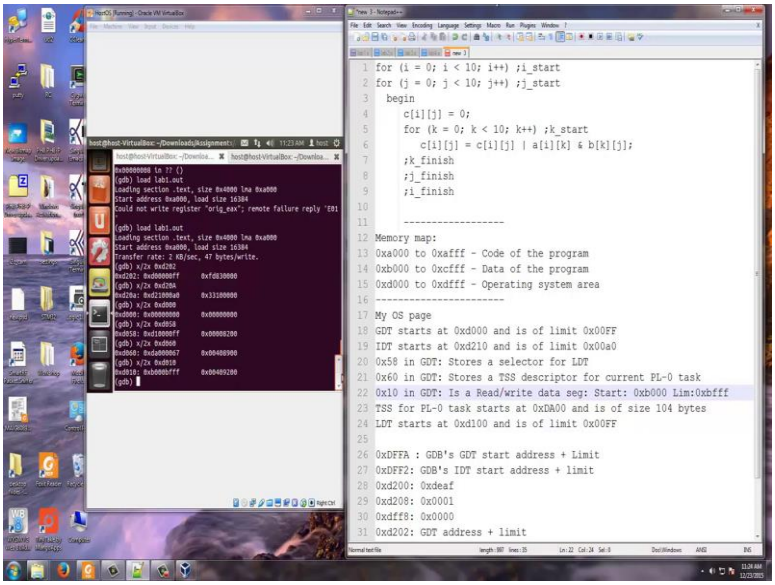
Now let us go and find out so go back to this window here on you left hand side let us say x slash now that is 16 by, so 4 plus 2 6 bytes. So, x slash 2x0xd202. Now you will know that the limit is ff and actually 0 0ff the basis d000 and plus 00. So, there is a 16-bit limit which is ff and 00 ff and there is a 32-bit start address which is 000 d00. So, from this we will get that, I am making another statement here. I will say my OS page, GDT starts at 0xd000 and this of limit 0xff.

Similarly, let us go back to the next statement here your IDT line number 22. IDT base and address are stored at d20A. Your 0xd20A will have IDT Address plus Limit. So the IDT starts at (Refer Time: 17:16), for that we go and say (Refer Time: 17:19) 0xd20A. Your IDT starts at d210 and it as a limit of a zero, so as a limit of 00a0 as I am marking here and start address at a 0000d210. So, your IDT starts at 0xd210 and is of limit 0x00a0.

Now just as a sanity check let us go and find out what should be the. So each GDT is 8

bytes, so I will just put slash 2x0x the 000 it should be a null descriptor. As you see that the first descriptor in your GDT which is stored at you know d000 to d000 1 2 3, so 4 bytes, sorry 8 bytes so 0, 1, 2, 3, 4, 5, 6, 7 is first null descriptor. Now, we want to move ax comma 0x58. So the ax gets 0x58. Now I am loading LDT with ax. What is LDT? It is Local Descriptor Table address. So what is 0x58? It is selector in the GDT.

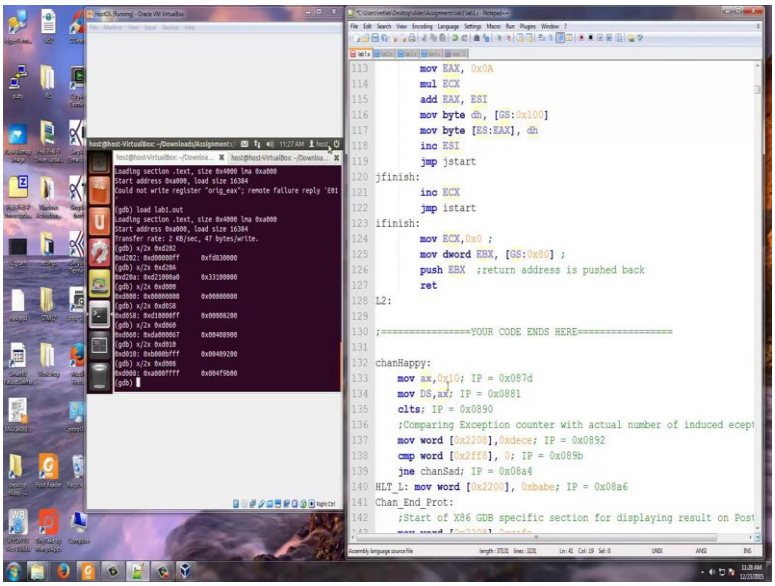
(Refer Slide Time: 19:12)



Now, how will I find out what is the value of, I say x slash 2x0xd2 sorry d058. And note that this is pointing to basically TSS selector which starts at d100 and this 82 basically tells that this is a selector of privilege level 0. Basically, this is a LDT selector of privilege level 0, 0x58 is an LDT selector and the LDT starts at d100 and it is of size 0 0 ff. So this is what I set up the LDT in case I need it. Let me go here, so the 0x58 in GDT stores a selector for LDT. LDT starts at 0xd100 and is of limit 0 0 ff.

Now we see what is 0x60, x slash 2x 0xd060 what is that? Please note that this is basically a task segment descriptor of privilege level 04089 here, and is a 32-bit task state segment descriptor. It starts DA00 and sizes 67. As you know that task state segment has total bytes which is 6 into 6 into 16 96 plus 700 and total 104 bytes. So, just basically is a task segment descriptor which starts at DA00 and the size 67. I should also note here that 0x60 in GDT stores a descriptor to current for current (Refer Time: 22:28) for current PL-0 task. It shows at TSS descriptor. TSS for PL-0 task starts at 0xDA00 and is of size 104 bytes. So, this is what it is currently executing.

(Refer Slide Time: 23:45)



Now, we are moving 0x10 to DS, SS, ES, FS and GS. So what is 0x10? Let us go and say x slash x, x slash 2 x, 0xd010. Please note that this is segment starting at b000 and as I huge limit bfff and it is privilege level 0 and it is a read write able segment. So, we will go back to this now, 0x10 in GDT is a read write data segment start as0xb000 limit as 0xbfff. At end of these all these selector will now can whatever you write will starts from b000 and whatever offset from that and they are all PL-zero segment. Now, what is 8 0xd008, this is a code segment, which starts at a000 and it is of size limit fffff, f followed 4 f's as you see here, there is also small f here, this i huge segment.

Now, you jump to this 0x8 with an offset start minus code base, this is start address and code basis on the top, this is code base. The code base starts at a000. So, what is start minus a000? Basically, we are start minus a000 is total number of bytes consumed by the code above it. So this will be from a000 whatever be the number of base consumed will be this start minus code base. If I say jump 0x8 colon start minus code base, it will take the base a000 and it will add so many bytes which is given by start minus code base and little jump, that means it is going to start it will jump to move d word 0x1 ff0 (Refer Time: 27:26) so it is basically coming to this. Why it is coming to this? Why do I need to do this jump? Because I need to switch from GDT code segment to my own code segment, 0x8. Here, I have switched on to my own data segment, now here I am switching on to my own code segment. Now makes move d word 1ff 0 0, move esp as 1ff 0 and pop fd.

It is initializing all the flags to 0, pop so you know move to what is d word 1ff 0 because the basis b, bff 0. We are now moving to 0xbff 0. We have moving the value 0, d word is the double word, which is 8 bytes and moving values and I am making stack pointer also point to 0x1ff 0. I am doing pop fd, essentially 0 little word initialize all the initial able flags. And now I give control to the code. What we have done so far is what the operating system normally does, so creates memory, load program, it sets up all the things and now it gives control to the code.

So, we have described this entire process. Further, it will execute this code, and after that it comes to this stage where it move ax comma 10, ds comma ax. What is GDT? Is d010 it is a segment starting at b000 and say bfff, then what is does is? It goes and loads the store, please note that we are store and dffa, the previous GDT and the LDT, IDT entries load at back and it does a jump 0x8 colon fin comes here for this GDT. Now the GDT has changed, so this 0x8 will now corresponds to the GDT of the GDB and it will come back. So, this is the way the operating system moves the GDB (Refer Time: 30:25) moves from itself after initializing certain things and you start executing your code, and after you finish your code you actually give back control by getting that GDT and IDT loaded. We do this in 3, you set up all these things with a new GDT and you do this in 3 which will take you back to the GDB part.