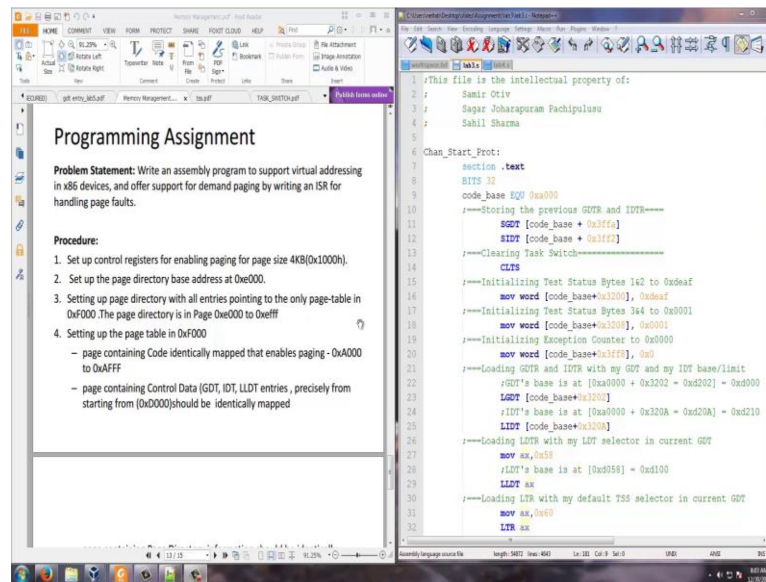
**Information Security – II**
**Prof. V. Kamakoti**
**Department of Computer Science and Engineering**
**Indian Institute of Technology, Madras**

**Lecture – 28**
**Lab3 Part 1 - Week 5**

(Refer Slide Time: 00:09)




Welcome to the session, where we are going to talk about paging. Paging as I had mentioned in the previous lectures, they are one of the most interesting architecture innovation. This allows a programmer to have a view of a 4 Giga byte, if it is in the case of 30 bit architecture but executing a program as large as 4 Giga bytes in a very, very small memory and the reason for that is, the statement that I made. For me to successfully execute a program and take it to completion, it is enough if at every point of time, the next instruction to be executed and the data required by the next instruction to go to completion. If these two are available in the RAM, the random access memory it is sufficient for us to complete program.

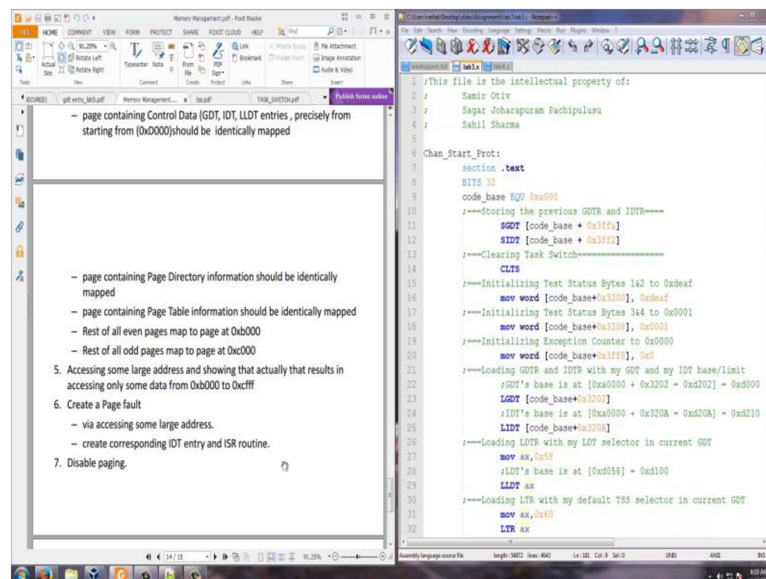
This is what we started and from that evolved the concept of paging and in this session we did talk about paging and how it is enabled specifically in the x86 platform in detail. Now, we will see a lab which will basically talk about all the things that an operating

system needs to do to enable paging and once that is done then we will also go into subsequent ways by which you can demonstrate paging. So, these are all some of the steps as you see on the left hand side in your screen, some of the steps taken by the operating system.

First thing is, it needs to step up all the control registers for enabling paging. Specifically, we will now talk about 4 KB pages, which is hexadecimal it is 0x000; we will step up the page directory at 0xe000. So, let me just recall till about a000, it is all the GDT kernel in your guessed machine and then from a000 to c000, it is going to be a000 to c000, a000 to d000dfff, it is going to be some pages of code and data and from e000 we have some available space. So, we will set up the page directory, in the page 0xe000 to 0xefff and then a page table starting from 0xf000 to 0xffff. We will do the following the page containing the code that is 0xa000 to 0xafff, will be identically mapped.

Similarly, the page containing control data namely the GDT, IDT, LLDT, will also be identically mapped and the page containing the page directory itself which is 0xc000pfff it should also be identically mapped and the page containing the page table that is 0xf000 to 0xfff that is also to be identically mapped and what we do is, all the other even pages we map it to 0xb000 and all the odd pages we map it to 0xc000.

(Refer Slide Time: 03:25)



So, the entire setup is, I have one page directory, all the entries in the page directory will point to a same page table. What is the page table, which is be pointing from 0xf000 to 0xffff and every entry in that page table, all the even pages will be mapped down to 0xb000, all the odd pages will be mapped down to 0xc000, except those pages which contain the code, which will be identically mapped. The page which contain the control data that is 0xc000 to 0xcfff that is also be identically mapped and then we will have the pages having the page directory and the page table themselves they will also be identically mapped. Why they need to be identically mapped is already explained during the part of the lecture.

For example, if the page containing the code if it is not identically mapped, namely 0xa000 to 0xafff, which actually enables paging then what happens is, the movement we enable paging then the next instruction to be executed will be translated and it will go to some other page. So, it is very, very important that we mapped at 0xa000 to 0xafff that is, the page which is enabling paging on to itself, so that once you enable paging somewhere in the page the next instruction to be executed will also belong to the next instruction in that page itself. This is very important, we have already demonstrated in the class.

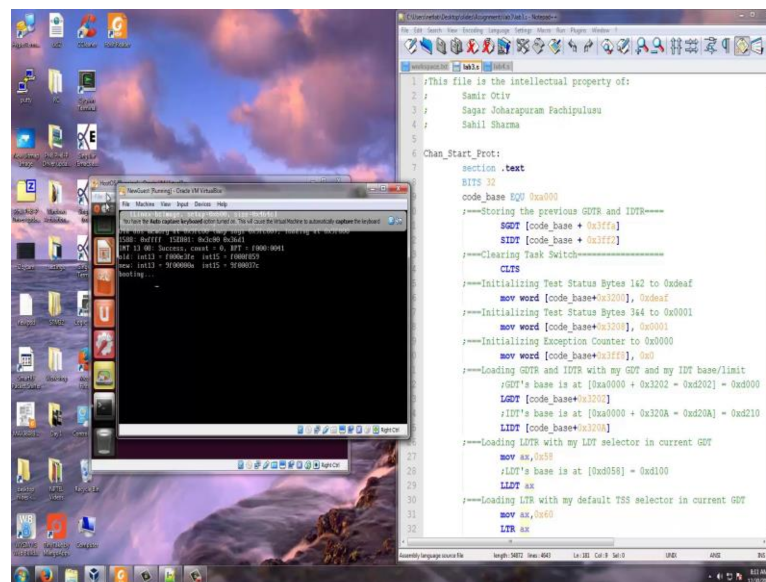
Now, after doing all these things after establishing this page directory, this will be completely done by the operating system. So, when your program is going to be loaded, all these things should be setup then what you do in the program is to demonstrate how paging works. So, you actually access very large addresses for example, you go and write into a very large address, but that will be into a page either b000 page or the c000 page so you write into one of those two data pages.

Similarly, you will jump to a very large address but that will be the next instruction. So, this is all these things will basically demonstrate how paging works. The next thing that we would like to also tell you is that we will also create a page fault, that is we will go and make one of the entries as the page not present and that will create a page fault and then we will take you to a interrupt service routine which is handling that page fault essentially that will the page fault handler in the operating system. What it will do is that in practice it as to go look for the page in the disk, load it here and if needed it has to do

a replacement page replacement policy. So, many things it has to do and that does things should be covered in an operating system course and after doing all these things it will make the page as present and then return back the control.

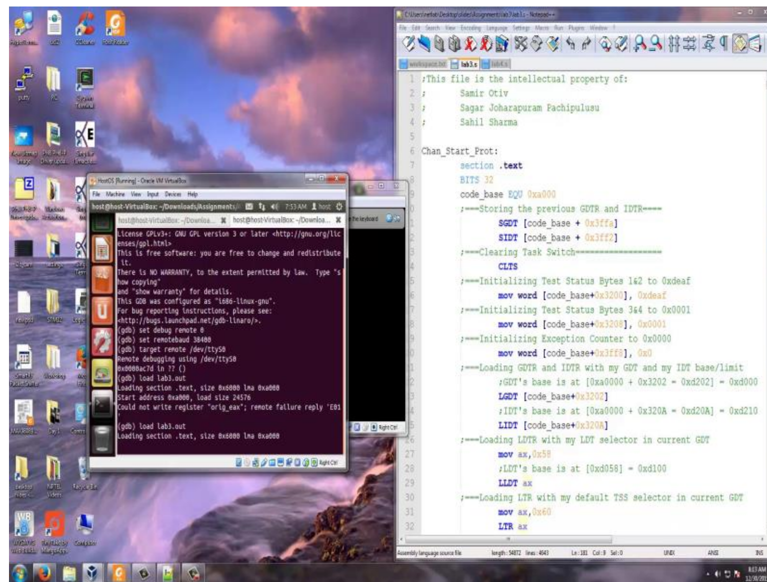
So, what this ISR routine will do is to actually go and make that page as present and it will allow you to continue. So, essentially by doing this we know how a page fault occurs and how do you transfer control to a page fault handler and then may be some functionality of the page handler and return back and start executing then and finally, we will go and disable paging before we come out of the program. So, this is what we do as a part of this complete programming assignment.

(Refer Slide Time: 08:02)



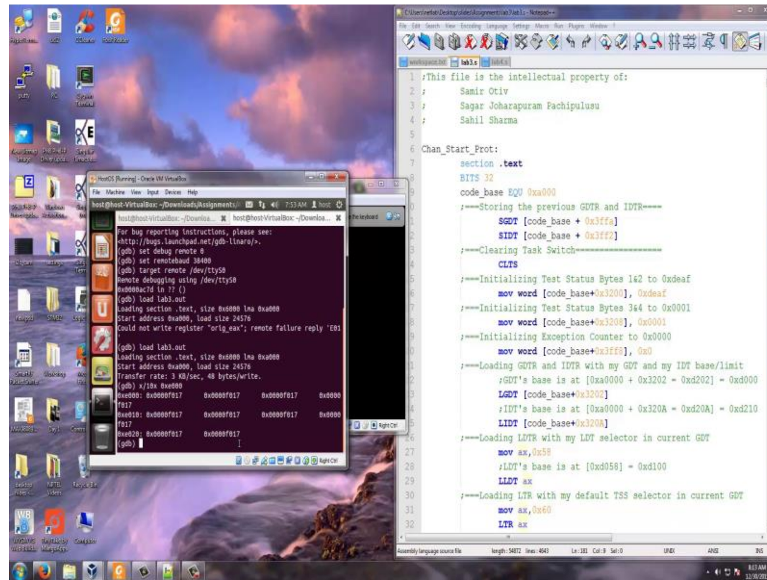

Now, we will go into the code, so this is the host machine, the guest machine is already running.

(Refer Slide Time: 08:07)



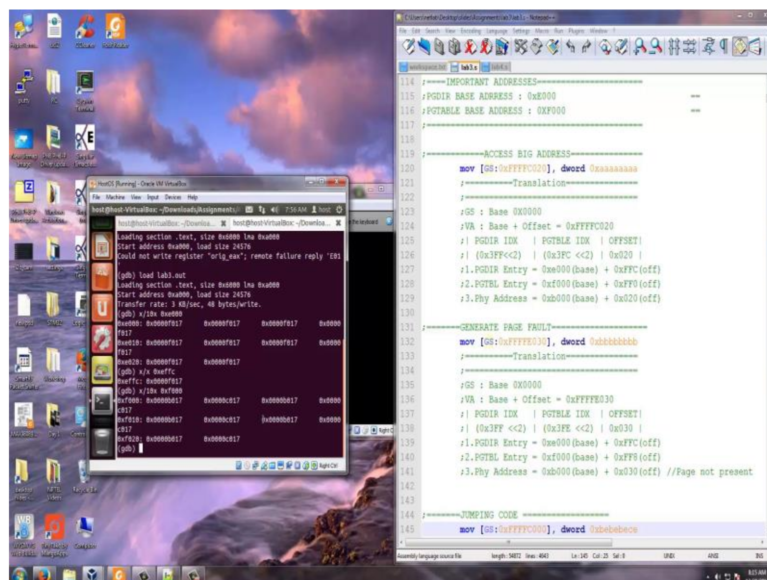

Now, we are in lab we have to go to lab 3 and you have this everything here. So, what we do is dot slash GDT comma lab 3 is compiled now, now we will go and do new tab; we will create a new tab here. So we are going here, so we will now go and say if; so these are the typical commands now we have come to this stage now we will go and load lab 3 dot out. We will again load lab 3 dot out and now we will go and see where all the pages are loaded.

(Refer Slide Time: 09:50)




First and foremost let us go and see where is x slash say some 10 x of 0xe000. All the entries in e000, as you see points to f000. So, this is how the page directory entry please note that the last 12 bits are used for privilege and other things so please note here that all the entries are pointing to f000 and there will be one entry in which the thing is not present, the page not present because we wanted to create the page fault and that entry is

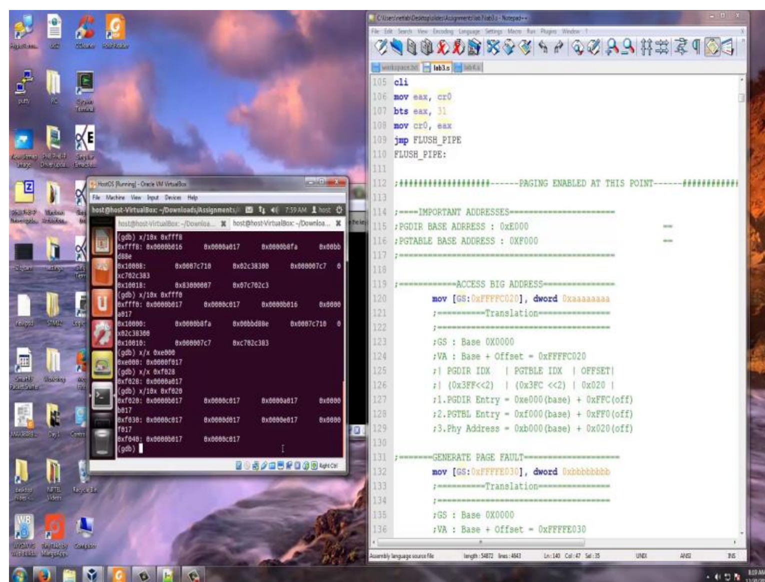
(Refer Slide Time: 11:09)



Let us go and see, so we are seeing the code here. So, that is I have made it as x slash to x or even x slash x, let us go and see 0xeeffc and so this is now pointing to f000 entry. So, inside f000 entry let me go and see 0xff, let us see what is there is f000, so x slash 10 x. Now note that pages are pointing to b000, c000 all the even pages that is page number 0 is pointing to b000, page number 1 is pointing to c000 and so on. Now let us go and see one page where I want to create a page fault, please note that 1, 7 essentially mean that page is present, all these pages are present.

Now let us go and see x slash x, 0xfff8, this is alone p016 so just to make it a little more. So, as you see that, this is alone p016 or is less than these are all p017, p017, but this is alone p016. So, that page alone is not present; we will use this particular entry to create a page fault which is there in fff8.

(Refer Slide Time: 13:19)

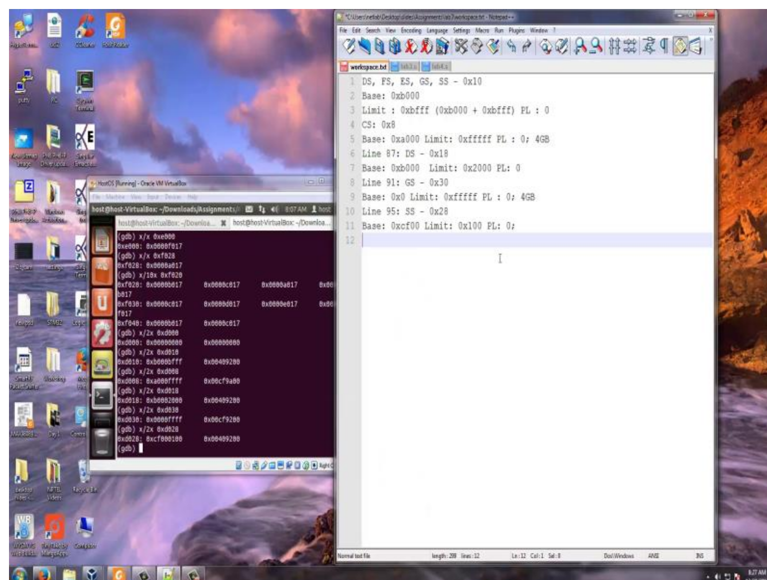



If you look at a code on your right hand, side you see that that has been given here. So, I am generating some huge address which will map on to this fff8 and there you find out that the page is not present, this is basically how I generate a fault. Now, have to also tell you that some of these pages are identically mapped so let us go and say where is a000 to be mapped here which entry would be a000. Let us go and find out which entry a000 will

be, so let us go and see where; I just leave it as a one minute exercise for each one of you where will a000 be mapped.

The first entry would be all 0s because the first 10 bits are 0, if I take the address 0 at a000, the first 10 bits are all 0s so that means, so I now go the first thing is, I go to e000, I use the first entry that the 0th entry, the 0th entry now points to f000. Now in this f000, I have to index with a, what is a? a in hexadecimal is 10 and so each entry is 4 bits so I have to go 40 bytes into this. What is 40? 40 is 28, 28 is 2 into 16 is 32 plus 18 hexadecimal it is 28. So, I am going into the page directory, page table which is at f000 and going to the 28th entry and note that the 28th entry is now, a000 17 it is mapped identically mapped. Now just see that, we will just do some temp just show you how it is going to work, we will just see x slash 10 x; now you see that b000 is mapped on to be b000, c000, etcetera the odd and even page, now this is the 28th entry, it is mapped on to itself. Please note that, the next page b000 is mapped on to itself here and then the next page c000 is mapped on to itself. Now, you also see that the d000 page is mapped on to itself this is the controlled page, then e000 is mapped on to itself, then f000 is mapped on to itself and again you start doing b000 and c000, so the mapping is pretty clear here.

(Refer Slide Time: 16:40)



We will now go and start analyzing this code that you see on the right hand side. So, what you see here, till this we have already seen and please note that at this point of time, I have my a all my ds, ss, es, fs, gs pointing to 0x10, which is one and my code segment is at 0x8. So, let us start having a work space here so now I am in line number 46, when I reach line number 46 your ds, fs, es, gs, ss all are pointing to 0x10 selector, your GDT is already there in d000. So, let me go and see x slash, 2 x, 0xd000 and sorry 0x10, this all d000 10 and so this is b000. So, here the base is b000 for this and limit is 0xbfff; that means, 0xb000 plus 0xbfff, so large limit and your code segment is pointing to 0x8. So, let us go and see what is there in 0x8, d 008; this is the GDT. This is a code segment, this is of privilege 0. So, base is a000 and limit is fffff very large limit and then privilege level is 0 as you see here, we can just go and interpret.

I hope by now you are all very clear with segmentation. So, this basically gives you that privilege level and here also the privilege level is 0 for all. So, this is basically where we are when we come here and your stack pointer currently set is 1 ff 0. Now, what we do here, now let us go and start doing this code; now I am coming here, I am moving move ax comma 0x18 to ds what is 0x18. So, x slash 2x, b018 and that starts with b000 and a limit of 2000 and a second privilege 0.

So, now at line number 87, we have now made ds point to 0x18 and that has now limit sorry base b000, limit is only 2000 and privilege level is 0 and these are the things that we are and it is of type two so it is a (Refer Time: 20:29) segment. Now, gs we have made it as 0x30, gs we have made it as 0x in line number 91. So, in line 91; I am making gs as 0 x 30, that essentially means x slash, 2 x, 0 x, 0 x, d 0, 3 0. Now this is a segment with base as 0 and limit as ffff, 5 fs and limit as 5 f and so this essentially and also the granularity bit is set to 1.

So, this can essentially cover pl is 0 and please note that the granularity bit is also set to 1 so this can cover 4 GB. Interestingly also see that if I; just see the mouse here, if you take the code segment again there also the granularity bit here is set to 1. So, the limit is interpreted as you know 4 KB, so even here this is a 4 GB page. So, from base I can go as long as base plus four GB (Refer Time: 22:30) meaning it will wind up and here base

plus 4 GB, we can go to 4 GB here. So, these are the two codes and now I make the stack point to 28 so, this is line number 95, as you see here I make the stack point to 28

So, 995 I make the stack pointing to 28 so; that means, its bases where as going to see here its base is c f00, its limit is 100 and its privilege level is of course 0 and it is just a normal page, it is not a magnified page, granularity bit as you see here this 4 basically indicates that your granularity bit is 0. So, this limit is interpreted as 100, note that the stack is in a page, which is identically mapped because c is mapped identically on to itself so this also very important. Now, let us go to the next line so now, we also set esp as 0x00. So, esp now point to be a limit of the stack so when I am pushing, it is a growing down stack, so it will start working correctly.