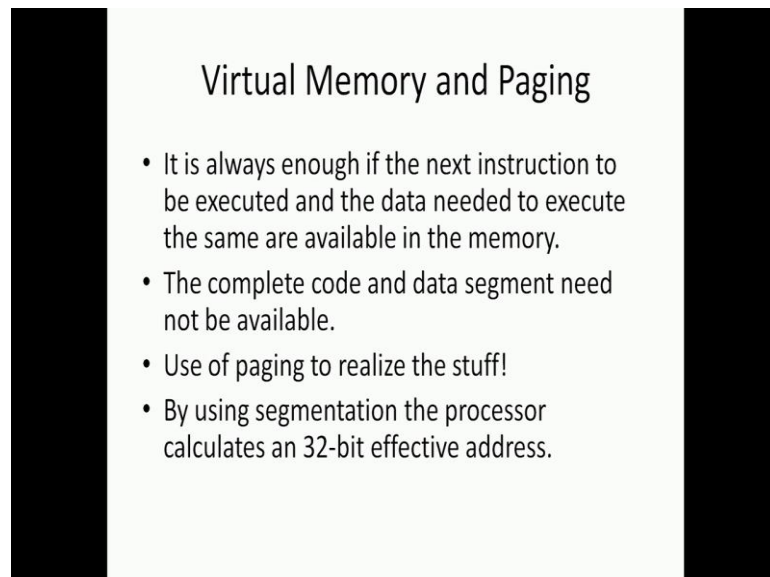


Information Security - II
Prof. V. Kamakoti
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Lecture - 14
Architectural Aid to Secure Systems Engineering
Paging and Virtual Memory

Now, what is this notion of virtual memory?

(Refer Slide Time: 00:16)



Virtual Memory and Paging

- It is always enough if the next instruction to be executed and the data needed to execute the same are available in the memory.
- The complete code and data segment need not be available.
- Use of paging to realize the stuff!
- By using segmentation the processor calculates an 32-bit effective address.

I say, I have a 32 bit architecture that means, I tell the programmer, Hey, you have 4 GB of address space. You have 4 GB of address space, please use this 4GB of address space. You may not get this 4 GB of address space in practice, because multi- users are using the system. But, you cannot tell the programmer, this is a multi-user system; though I assure you 4 GB, just do some 200 MB; do not, do not look for 4 GB. I tell the programmer, you use 4 GB; you create a program that is as large as 4 GB, right. But, we will now execute it. I will take responsibility of executing it, correct. Right. Then, you are a system now, and you have multiple users who are logging into this system. You have multiple users who are logging into this system. Now, what will happen? I will not get 4 GB as a user. I cannot give 4 GB, as an operating system, I cannot give 4 GB to every

user. Can I give 4 GB to every user? No I cannot give it. Somehow with the limited amount of memory I have I have to execute a program that could be as large as 4 GB right. So, what I am telling you is 4 GB is available to you, but physically I do not have that 4 GB. So, I am creating a virtual world for you where 4 GB is available, but physically I do not have that 4 GB correct? Do you agree with me? And, that is why this notion is called virtual memory. Now, virtual memory. So, I make this promise and then I also see to it that you know your 4 GB program essentially gets executed. What is the underlying principle that makes virtual memory possible today? This is the question that we need to answer, right? Please note that the virtual memory as a concept is a success because suppose, I have say a 4 GB program the entire 4 GB need not be loaded into the memory always.

It is enough if at every point of time when the program is executed if the next instruction to be executed and the data needed to execute that next instruction, if these 2 are available in the memory that is enough. It is not necessary that I should load the entire 4 GB of program always for the program to execute and go to completion. It is always enough at every point of time the next instruction to be executed and the data necessary by that instruction to complete execution. If these 2 things are available in the memory, I am happy, yes or no? So, I really do not want anything else. And to store that next instruction and to store the data necessary by that next instruction to go to completion it is not going to be 4 GB. It cannot be even 1 kilo byte in many of the time; you are getting this? So, that this underlying truth or fact is the success behind us having a good virtual memory in place. So, let us go to the next stuff.

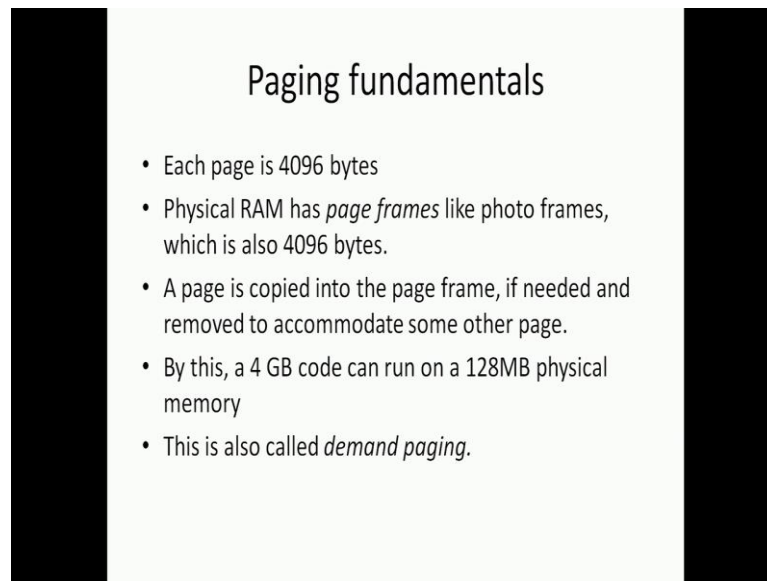
Let us read this slide again after this explanation. It is always enough if the next instruction to be executed and the data needed to execute the same are available in the memory. The complete code and data segment need not be available. And, the notion of paging is going to exploit these two facts to realize our goal of executing a 4 GB program theoretically or even practically a 4 GB program using say 2 MB of memory. So, now let us see first there is segmentation and then there is paging. So, many of the operating system, if you go through the source code

segmentation is trivial; they use everything on paging. And that that can lead to lot of vulnerabilities. Now, you assume a new generation of operating system which one of you will build after taking motivation from this course; I will be happy. I am too old to build a operating system.

But I think young fellows can build one. So, wherein we are using segmentation in its full perspective and then using paging on top of it. Now, what will segmentation do? Let us now start. I am a process. I say, 'Hello, I want to execute'. Now, as a operating system, what do you do? You create a code segment you create a data segment, stack segment you create an LDT for me you put all these descriptors there and give it to me. And then, when I start executing I generate an address. I generate a logical address; that goes through the segment table and generates me another address. Now, it is not the physical address. In the old time when I did not have any paging or anything it is going to be the physical address; it is going to be the actual address in the RAM. Now, it will generate me one address which I call it as an effective address.

This effective address I have to find out if it is available in the RAM or not. In my previous case, where I did not have virtual memory if I want to execute a say 200 MB program the whole 200 MB has to be loaded into the memory. If the 200 MB is loaded into the memory, whatever address I generate if it is a legal address the architecture will actually take care whether it is a legal address that is the address which I can access. Then automatically it will be there in the memory. So, I will go and execute. But in this case it will not be there, it need not be there in the memory, if I am using virtual memory, because I am not loading the entire 200 MB. I will load some part of it. So, somebody has to go and check whether this address this fellow needs is it there already in the memory or not; if it is there in the memory, I will go and execute it, if it is not there in the memory, then I have to go and load it and then execute it, you got the point? So, this is how the virtual memory as a concept works.

(Refer Slide Time: 07:25)



Paging fundamentals

- Each page is 4096 bytes
- Physical RAM has *page frames* like photo frames, which is also 4096 bytes.
- A page is copied into the page frame, if needed and removed to accommodate some other page.
- By this, a 4 GB code can run on a 128MB physical memory
- This is also called *demand paging*.

Now, what are the things that we need to discuss here? We need to discuss the following: first I said, I will not load the entire program, but I will load a part of this program.

What do you mean by loading part of the program? So, the entire program is now divided into what we call as small chunks of memory which we call it as pages. Each page will be 4 kilo byte in size. So suppose, I have 6,000 bytes essentially I have 2 pages. The first page is zero to 4095; second page is 4096 to the 8192. But, the first part of it will be full, other remaining part will be empty. So, I need 2 pages to store a 6,000 byte program. So, the entire program is basically split into what pages.

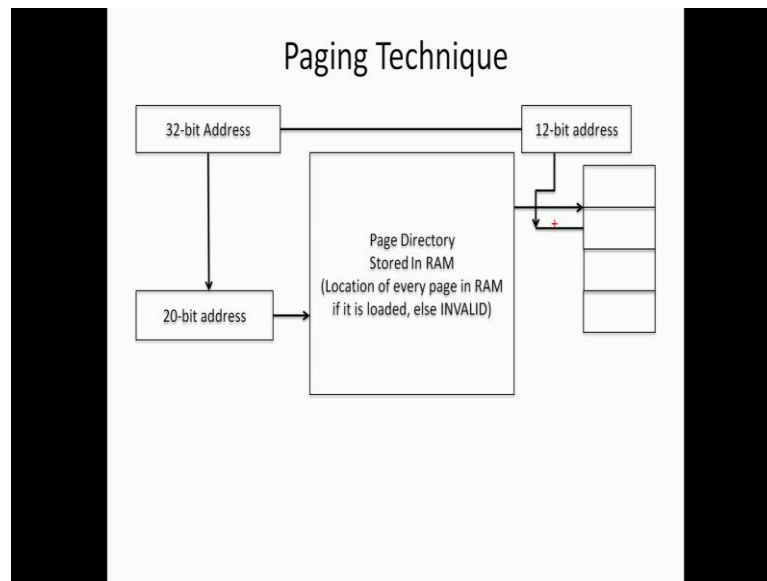
Now, the Intel architecture has made every page as 4 kilo byte, that is 4096 bytes. So, your 4 GB address space that is available how many pages will it have? 2^{20} pages. Each page is 2^{12} bytes; 4 GB is 2^{32} . So, I will have 1 MB pages. So, if I have 4 GB memory that is a logical memory because since it is a 32 bit architecture I could have memory of size 4 GB; that is not actually available, but that is actually possible. So, that is the logical memory that you can have. The actual physical memory would be much lesser than this 4 GB. So, that logical memory can be divided into how many pages 1 MB pages, 2^{20} pages; each page of size 2^{12} bytes.

So, now you have a notion of a logical memory which is 2^{20} pages and you have a physical memory which is of some size; but in this physical memory. Again, we will view the physical memory as pages of size 4 KB. Like how a photo will fit into a photo frame.

Now, these pages will fit into a page frame. So, your physical memory can be split into multiple page frames each of size 2^{12} . So, what you will load from the logical memory to the physical memory is pages. So, I load one page and I put it into the physical page frame. So, pages from the logical memory are loaded into the page frames in the physical memory. So, a page is copied into the page frame if needed and suppose, all the page frames are full and I need a new page to come in I go and throw out an old page and get the new page. Because, my execution can happen not from the logical address space which is represented in the disc; the logical memory is there in the disc. The physical memory is there in the RAM. As a processor, I can execute only code that is stored in the RAM not into the disc. So, I have to move if I want to execute something I have to move it from the disc to the RAM and start executing there. So, suppose I want to execute some page which is not loaded, so I go and take it from the disc and load it into the memory and start executing. But, in this process if all the page frames are full then I have to remove one page and put this new page and start executing it.

So, this is how the whole thing works. With this as the background, let us now read this particular slide again. Each page is 4096 bytes in the X86 architecture. The physical RAM which has to into which we have to load these pages for execution has page frames, like how photo has for photo frames and those page frames are also 4096 bytes. So, a page can fit exactly into a page frame. A page is copied into the page frame if needed and it can also be removed to accommodate some other page in case all the page frames are full. By this a 4 GB code can run on a 128 MB physical memory machine. And, as and when I require I load the page execute it and if needed I will remove it back and to the logical address space. So, as and when I require I get a page and that is why this is also called demand page.

(Refer Slide Time: 13:08)



So, what happens here? I have a 32 bit address. Each page is 12 bits in size. Each page can be addressed in 12 bits. It is 4096 bytes. So, the last 12 bits of this 32 bit address will give me the offset within a page. The remaining first 20 bits will give me the page number. Note that there are 2^{20} pages. So, the first 20 bits will be the, give me the page number and the remaining 12 bit will give me what, the offset within that page.

Now, I have something called a page directory which tells me whether these pages are either there in the memory or not, and if it is there in the memory in which page frame it is going to be loaded. If it is not there in the memory then I have to go and load it. If it is there in the memory then I should tell in which page frame is it loaded because, I have some amount of page frames; I can load every page into any of these page frames. Now it will tell me which page frame it is loaded. So, I use these 20 bits to index into the page directory and that will give me some details about whether that page is loaded into the memory or not that means, whether the entry is invalid or valid. Valid means, it is loaded into the memory. Invalid means, it is not loaded into the memory. And if it is loaded into the memory it will give me the page frame number.

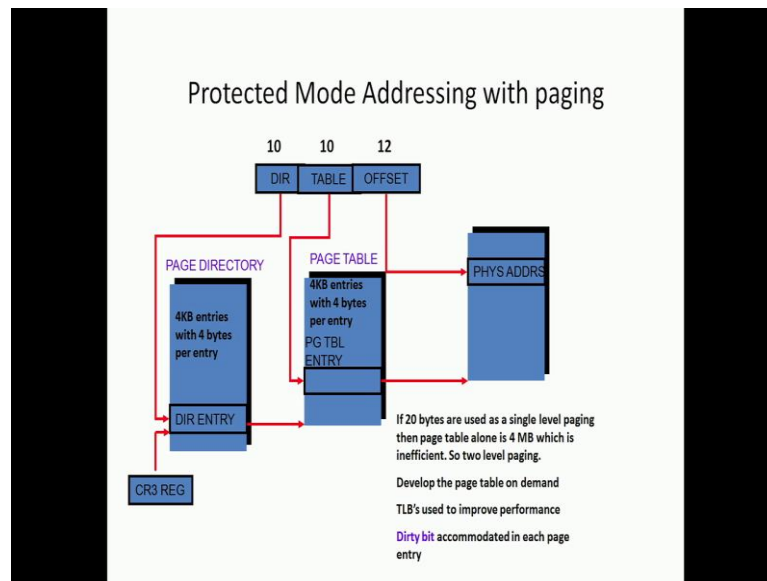
From that page frame number, I will go to that page frame and I will use this 12 bit as an offset to that add that value and I will access the correct address. If the page is not loaded,

then I will go and load that page; the operating system will go and load that page into some of this available page frame. If all the page frames are full it does something called a page replacement policy. It removes one of the page based on some policy that is called page replacement policy. There are many page replacement policies like least recently used etcetera. So, it will use one of this policy to remove it and it will put the new page and validate this page directory saying now this page is available there it is valid. Now, you go and start accessing it.

So, this is how the paging works. And, whenever I need I will go and access it. And, whenever the page is needed it will be loaded, if the page is not needed automatically it will be thrown out. So, by this I can take a large program like 4 GB and execute it on a 128 MB example machine. Now, what is the difficulty with this stuff? Now, what should I store here? I should store the starting address of each of these pages frame and that is 4 bytes, because the 32 bit machine I should know where the page frame starts I have to store the starting address here. And how many pages I have $4, 2^{20}$ 2 power 20 1 MB pages.

And, for each page if I am storing 4 bytes as starting address then how much would be the size of this page directory; that itself would be 4 MB. So, I cannot waste 4 MB just to have a page directory and it is not the case that I will be using this paging always; all the all the entries always. When will this entire page directory be full when all the pages are actually being used. It is not the case that all the pages will be used in average. So, please note that by doing this mechanism, we are now wasting 4 MB of memory; just for page translation which is quite costly. You are getting this it is very, very costly. So, that is the reason why we have moved on to what we call as multilevel paging.

(Refer Slide Time: 17:47)



So, this is what that happens in the X86 architecture and you will be implementing it as a part of a lab; you will be setting this page directory etcetera and implementing it as a part of the lab. So, there is one page directory

And so, multilevel means that the translation actually becomes multi. In this case the previous case the translation was only 1 translation. I give the first 20 bits and that gives me immediately that start of a page if it is there and I just add and go there. So, this translation is only 1 level here. But, note that there is a translation. This is the 32 bit address that comes on the top is given by the program that gets translated to a physical address; that 32 bit address is called the effective address and now the translated address the actual address on the RAM is the physical address. So, there is a translation that is happening here but there is only one single translation because of the single page table. But, when we move on to the next this is going to be multiple transactions that are happening.

So, let us go and say what is this multiple transaction here. So, there are totally 32 bits. Now I split it as 10 plus 10 plus 12. The 12 bit is anyway offset into that page table. The first 10 bits will index into something called a page directory. And, how many entries you need for 10 bits? I need 2 power 10 entries. For with 10 bits how many possible entries

are possible 2^{10} , that is 1 kb. And that will point each entry there will point into a start of another table which is a page table. So, each entry there should store how much a 32 bit address. So, that will be 4 bytes. So, there are 1 kb entry each 4 bytes that is equal to 4 kb, 1 kb entries because I have 2^{10} means 1 kb possibility is there;

1 k possibilities are there and in which each each entry I need to store 4 bytes. So, essentially, I need to store 4 kilo bytes. And each one will point to a start of a page table. Now, the next 10 entries will offset into this new page table; again, how many possibilities I will have for this page table 2^{10} and that will point to a start of a page; to that start of the page, I add this 12 bits and now that gives me this physical address. So, instead of having 32 bit in which one I take the first 20 bits index and get the page table entry page start and then add the 12 bits, I take the first 10 bits, index into a table and that will point to another page table; I use the second 10 bits to index into that table and that will point to the actual page, to that page start I add the offset and access the memory.

Now, note that the page directory and the page table they are also going to be stored in a memory and so they are also going to be internally stored as pages. But both are 4 kb in size. So, they also fit into one page right. Now the other important thing is there is a CR3 register. Yesterday, we talked towards the end control register number 3 and that will tell you the start of the first page directory. There is one page directory. So, CR3 will tell you the start of that page directory. So, to that start you take this first 10 bits and index to that start and that will give you another entry, there to that start entry you index the next 10 bits there and that will give you start of another entry and that page and to that, you index into the next 12 bits there and you get the actual address.

So, at any point of time the page directory should always exist, because that is the starting point. So, that will be only one 4 kilo byte; and as and when new and new pages are new and new memory is explored in the logical I can create these page tables and throw off these page tables. The second level page table all of them need not exist; as and when I am creating a new thing I create another page table and I if I go off, if those addresses are removed from the logical address space the corresponding page table can

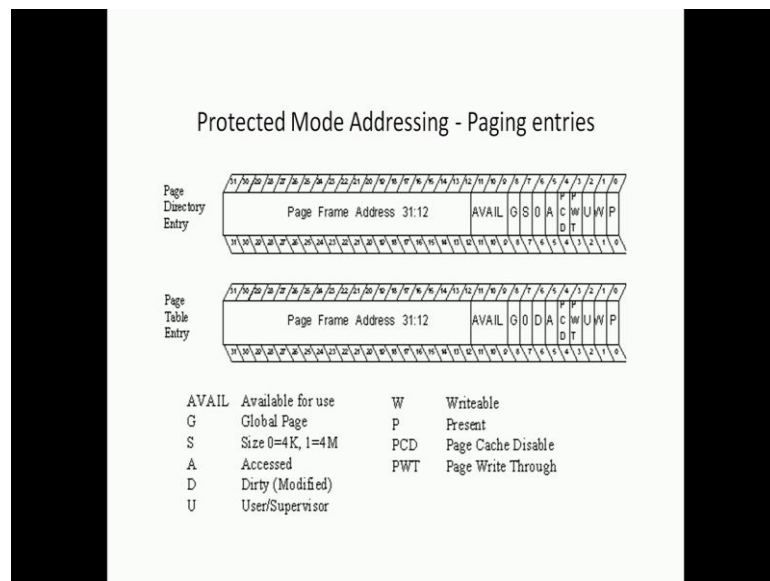
also be removed. So, as and when my logical address space is getting full this corresponding page entries also get full; as soon as my logical address space diminishes this will also diminish. So, this gives you a dynamic you know space translation setup, so that the amount of memory I am spending for this translation is much lesser than is proportional to the amount of memory I am actually using in the logical address space. If half your logical address space is full those addresses is will never be generated and so all these page directory entries can become invalid.

If a page directory entry is valid then only a page table needs to be created, you understand? So, if a page value page directory entry is invalid then the corresponding address never even exists. If the page directory entry is valid, then I go to this page table; if this page table entry is valid, then I go here; if this page entry is invalid, then I do not have a page that is mapping on this. So, by this I am doing a multi-level translation of my 32 bit address into 2 page, 2 level paging and I am able to calculate and the big advantage I get by here is that my paging infrastructure the amount of memory I use for storing these page tables dynamically grow and diminish with the amount of logical address space that I am using; do you get this? So, this is one very important thing. So, instead of using 4 MB of memory. Now I can start with 4 KB and the remaining thing can be flexible. 4 kb needs to be there; that page directory has to be there but the remaining can be flexible. And also, note that CR3 is a control register and that can be updated only by a privileged instruction; 'move CR3 comma some value' can happen whenever I use a control register as an operand it can be due to, it can be, it should be through a privileged instruction only.

So, a privileged instruction can only set the start value of a directory entry. So, that is also very important and that is again from a protection perspective. So, now what happens is I do segmentation; I get the effective address; that effective address is this 32 bit address; that goes through this paging translation; and I actually reach memory. So now I have added one more level; it is not just for protection, but it is for virtualization to provide you virtual memory but I have added one more level of translation. Now, this is going to be now one access to the memory to get the page directory then one access to the memory to get the page table then one more access to the memory to get the actual value.

So, one memory access now became three memory accesses. So, to get to reduce this problem, it is not to avoid this problem or prevent this problem to reduce the impact of this 300 percent degradation, so what has happened is, they use something called a Translation Lookaside Buffer. This is part of an architecture course but we will not deal it here; because it is not for the to understand the functionality it is not necessary. But whenever you do an architecture course you will learn about this TLBs. These TLBs are actually used to improve performance. So, the TLB actually stores the translation. Instead of going through this multilevel this page this is the page there. So, the TLB will automatically give you the translation lookaside buffer, translation lookaside, do not look at the translation. So, if I have the answer I will give you. This page is mapped on to this page frame; I have the answer; I will give you. If it is not there then you go through the whole story. So this is how. So, I will just finish off with what would be an entry in the page directory and what will be an entry in the page table.

(Refer Slide Time: 27:06)



So, the page directory and page table entries are seen here. So, lot of things are there. So note that the page frame every page frame, basically starts at a 4096 byte boundary. Each page is 4 kb in size. So, each page frame starts at a 4096 byte boundary. So, the last 12 bits are going to be zero. So, I start using those 12 bits in a very constructive way. One of the thing is if something is not full always I need one thing called a invalid bit or a valid

bit. So, the first thing that you see is the P bit that is a 0 bit which essentially says whether the page is present or not present. So, that entry is valid or not.

Your entry is valid only if the corresponding page is present. So, in the page directory entry is there a corresponding valid page table present or not that this P bit will tell you. If this P bit is 1 then I go subsequently to the page translation. If the P bit is, then I generate what you call as a page fault. Who will generate? The architecture will generate the page fault and give it to the page fault handler saying this fellow is asking for this page please go and provide it. So, this is the architecture support to paging. So, P 0 means the page is not available; P is 1 means it is present. Then there is W bit which says this page is writable; I could have read only pages. This is also important from a security point of view. I could I want to share a privilege zero data to privilege 3, but I do not want it to manipulate it. So, that entire page I can put one more level of protection by saying it is a read only page. So, this is write-writable. Then there is one U bit which is user or supervisor page.

We will see in the operating system course whether it is a user page or a supervisor page. Then there are some bits like PWT and PCD, which is Page Write Through, Page Cache Disabled; these are all architectural stuff. We will just not bother about it now. But for all practical purpose you will set it to zero. A is an accessed bit. So, whenever a page is accessed the architecture automatically sets it to 1. And why do we need this type of an access bit because for me to implement page replacement policies, at a later stage all the page frames are full now I need a new page. So, I have to replace an old page. So, for me to implement a page replacement policy I may need to have one of the replacement policies least recently used. So I need to know whether this page is used or not in the

recent past. So I use this access bit. So whenever some read or write happens to this page immediately the architecture will set it to 1 and I can go and reset it to 0, after some cycles. But during one period of cycle I can now have the list of pages that are accessed which will enhance my decision which will improve my decision on deciding which page should I remove and which page should I keep. Then there is a G bit which is called a global page or a local page. We will also deal with that in great detail when we go to the

OS course. And then there is some 9, 10 and 11, these are available. So, you can use these bits, the operating system can use these bits to use for supporting the replacement policy etcetera. And then, whether there is a complete page directory entry and whether the page frame address which is 12 bits and the no the page frame address which is the 20 bits and again a page frame address it could be a 20 bits in both cases. So, this is how the page looks like page entry looks like; and you will fill this up, so each of this page directory and the page tables will have several of this valid entries; valid and invalidity is decided by this P bit which is the present bit ok. So, this is how paging is implemented over and above segmentation.

So, all the protections that you get in segmentation is still there; after those things are validated now you go and use paging to basically see the things. So, this is this is a very, very broad description of paging in the X86 architecture. In the subsequent course and also subsequent part of this course in the lab, we will have a deep dive into these things to understand it in a better perspective. And, many of these things I will keep repeating it, because I believe that at least 7 times I have to repeat so that I am sure that it has reached you properly; because these are all much ambiguous topics also. Though they are quite straightforward to understand but when you actually go into implementation you will land up with certain unsolved questions. So I will keep repeating the theories several times at least 2 to 3 times by the end of this course every concept would be (()) so that, you are very thorough with this course. But at the also I will also see to it that we do real practical exercise so that we are in good shape in terms of understanding this entire concept.