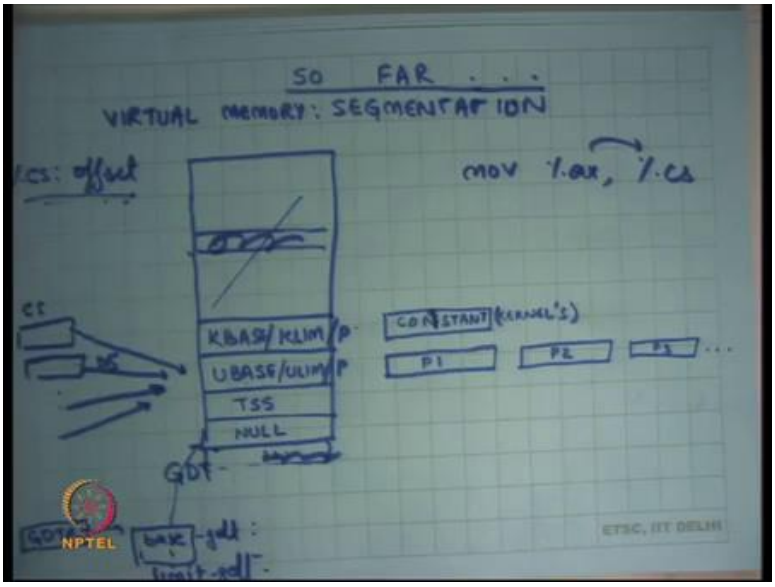
Operating Systems
Prof. Sorav Bansal
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture - 10
Segmentation Review,
Introduction to Paging

Welcome to Operating Systems lecture 10.

(Refer Slide Time: 00:33)



So, far we were looking at segmentation as a way of implementing virtual memory right. This notion where there is a separation between a virtual address and a physical address and there is some translation logic that goes to convert a virtual address with physical address. This system is called a virtual memory system. And we saw segmentation as one way of implementing virtual memory right, where the translation was rather simple, we just added base to virtual address to get a physical address.

And, we said the way it works is that there is a table which is stored in memory which is called the global descriptor table which contains these descriptors right. And this table can be as large as 2 to the power 13 entries, where you know each of these is one entry. Each of these entries contains a base address which will be a physical address and a limit which will say you know what is the size of this particular segment?

So, each descriptor defines a segment and the program itself is going to you know, is going to dereference one of these descriptors using the segment selectors which are these segment, which are stored in these segment registers that are cs ds es fs etcetera.

And, you know they are going to dereference these descriptors here. And we also said one typical way of organizing the segment table is the following. You know you have some extra entries in this to store the task segment and null entry; we can ignore this for the time being. But there will be you know two descriptors, one will be for the kernel, the kernels base, and the kernels limit. And the other one will be for the user, users base and user limit right at any point of time.

So, at any instant of time you will have these two entries know in a typical operating system which is using segmentation to implement virtual memory. And, what happens is typically that the kernel base and kernel limit remain constant throughout the execution of the system right, because they refer to the kernels space. And the kernel always maps itself in a certain address space and in the physical memory and the kernel is always you know is always mapped, and it's always the same contents of the kernels that are always mapped.

But user base and user limit descriptor keep changing over time because you know context switches happened between one process to another process. So, you know for at one instant it could be the P1's UBASE and ULIMIT which are loaded in this descriptor. After a context switch it will P2's UBASE and ULIMIT which are getting stored in this over it overwritten in this particular descriptor right.

So, the kernel remains constant and the users the processes descriptor or the processes the address space which is for the user is defined by the by which process is currently running. And that keeps, that gets swapped on every context switch and that happens by the kernel. Both of these needs to be mapped at any point of time because you know as we have seen control can transfer from user to kernel if some event occurs like an exception or an interrupt or a software interrupt and in which case immediately you need to dereference the kernel side of that space.

For example, the CS in that in the IDT table could be a kernel could be must be pointing to the kernel descriptor and. So, both of these are mapped the way you prevent the user from accessing the kernel address space is basically by having another field here, which

is the privileged level they call it P. So, one of them will say you know I can only be accessed in privileged level 0, and the other one can say I can be accessed in either privileged level 0 or 3. So, you know it does not. So, that way this you know I when if the processor executing in unprivileged model only be able to access the dereference through this descriptor and not others, alright.

Also, typically now I said that there are 6 segment resistors CS, DS and so on right and each of them could actually be pointing to a different descriptor here, potentially right. CS could say I want to point to descriptor number 2, DS could say I want to point to descriptor number 3 and so on right, assuming you know privileges are respected etcetera.

And, so that can potentially allow one process to have six different segments in physical memory and so that can solve some of the fragmentation problem that we discussed last time. But this kind of an organization where you divide your address space into multiple segments actually complicates the programming model, because now the compiler needs to worry about exactly how does this address live in this segment or that segment etcetera and it needs to generate code appropriately right.

So, typically what is done is that all of these actually mapped to the same segment to have this, to give the abstraction of a flat address space right. So, the compiler can assume a flat address space and all of these actually mapped to the same sort of descriptor and so you the process just sort of sees one flat address space from 0 to max right. So, it simplifies the programming model, but it has the problem of fragmentation as we discussed last time. Now, the entire process has to live in one contiguous segment ok.

Also there was there was a question last time about you know the GDT being too large you know, 2 to the power 13 entries is too large and I am only going to use 5 entries out of these two to the power 13 entries. So, actually you know I emitted a piece of detail, the GDTR actually does not point to the base directly, it points to another memory location which contains the GDT base, let us call base GDT and limit GDT.

So, base GDT points to the base of GDT and limit says for the size of GDT. So, that way you know you do not need to waste all this space you can say that you know my GDT is only 5 entries or 10 entries and so. So, the hardware actually gives you a way of

saying that specifying the size of the GDT, and the maximum value of that size can be 2 to the power 13 ok. So, you are not really wasting space in that sense alright. So, question.

Student: Will there be any point in time when we will be using all the space of GDT?

Will there be any time? Can it happen that I use all the space in my GDT? Depends on your kernel model, in the model that I have I have described so far, it is not possible right? With a number of descriptors that can have is actually constant.

Right is just these 4 registers descriptors for example. So, it is not possible in this model in another model it may be possible in which case the kernel should have logic to handle it ok, but let us just let be this model and say no it is not possible another question.

Student: Is the KBASE and KLIMIT constant for all the entries?

Are the KBASE and KLIMIT constant for all the processes is that your question?

Student: Yes.

Yes.

Student: The same entry.

Yeah. So, KBASE and KLIMIT do not change on a context switch ok. So, on a context switch the kernel remains as it is, it is the processes address space that you are changing.

Student: Sir, one more question.

Yeah, one more question.

Student: Base GDT and limit GDT can likely find to a physical address when we do not need a base GDT.

Yeah, so, these are these are actually physical addresses alright. So, these are you know let us call them physical addresses for now. So, and base GDT saying here is where the GDT starts, and limit GDT saying this is where the GDT ends, that is all.

Student: We can just have limit GDT we do not need base GDT.

How do you specify where it starts?

Student: Because we have GDT here which points to the base (Refer Time: 08:31).

No. So, that I am saying GDT I will point to this structure and this will contain the base and limit.

Student: We can have limit in the table itself right, GDT?

You are saying you know why I need this kind of thing; I could have you know GDT I could GDT could itself have some notion here which says limit. It is a matter of you know, it is a matter of fact that this is how it is done on x 86, but you could design something else ok.

Student: So exactly what is the (Refer Time: 08:59).

So, it basically says that you know everything above this is not possible. So, for example, if I did not have a limit GDT and let us say the I have assumed that the limit GDT will be 2 to the power 13 always. Then what will happen is the kernel will have to reserve that extra space and you know invalidate all these entries. So, you know there is a certain bait in these descriptors which says this descript is invalid. So, you will have to say an invalid and all these things.

If he does not do that then that space can be used for some other data structure and if data structure you know so a process can now manipulate those things right. Because, let us say there was some other data structure living in this space and now the process actually pointed to point here then you know he can now access something he is not supposed to access for example.

So, if there was no limit then the kernel would have had to allocate the entire 2 to power 13 and just two sorts of not waste that space, you the hardware allows you to specify limit. So, so how are UBASE and ULIMIT accessed? They are basically accessed using a virtual address right. The virtual address in this case is basically of the form of a segment register colon some offset, and offset could be specified in one of the mode immediate you know indirect or displaced whatever.

In either case CS actually goes here. So, what happens is offset gets added to base and offset gets compared to limit right. And, that is how the actually dereferencing, accessing the GDT entries on each a memory access ok. The user is not allowed to read these this GDT directly, its only allowed it only gets dereferenced through this mechanism of virtual memory translation or MMU ok. The user cannot access these entries directly no, in case of context switch does the limit GDT. You mean does UBASE and ULIMIT change?

Student: No limit GDT change.

Oh. You mean this? No this does not change. So, these remain constant I mean. So, it depends different, different operating systems can implement it in a different way, let us say the way we are were talking about it is that the GDT remains constant. It is just this entry that is changing, everything else remains constant right it is just just particular descriptor that is changing on a context switch question.

Student: Sir if a GDT stays constant then why did we need to store it anywhere? Like if we know the base GDT and we know the size of GDT that is required for the entire operation, why do we need to store it under a base limit?

Why do we need to store limit GDT to because we need to tell the hardware what the size of a GDT is right? because the hardware that is going to actually dereference. So, how is you, how do you go a do this? Is going to say, let us look at the selecting CS and added to base right and he is going to compare it against limit. So, if there was no limit then you know CS would point here. And so now, it is it is the responsibility the kernel to ensure that you know all these entries are invalid.

Student: No, but sir, like kernel will always have just 6, 6 segments selectives, user will UBASE and ULIMIT will always have 6 segments. So, basically the size of the GDT will always stay constant right?

Let us say in our model the size of the GDT always stays constant. So, yes absolutely.

Student: So, in that case like we do not need to really store limit GDT anywhere we can just like build it built it into a model itself (Refer Time: 12:18)?

No. So, this limit GDT is a way for the of the kernel to tell the hardware, that is all; I mean the kernel is not storing it for its own purpose. The kernel already know that it is 6, its telling the hardware that its six no it has to tell the hardware in some way and that this is a way of telling the hardware look you know there is only six entries that you should be accessing ok.

Student: Sir, in this model (Refer Time: 12:43) when your CS remains always the same? CS.

CS DS yes everything remains the same. In fact, you know in this model when I context switch between two processes the segments selectors remain the same. They always pointing to let us say the third entry in the stable right. It is just the entry that is changing every time.

But you know yeah it is possible that you have you know it is possible that the kernel allows you multiple UBASE and ULIMITS in which case the program is allowed to change between multiple things; great. The next question the next thing I wanted to point out was that on every memory access of this type right, let us say CS or DS offset whatever what is required is CS is stored on chip. So, you know there will be some logic that dereferences CS to get a selector, but the GDT itself now that that CS the selector is going to get added to the base of you know of GDT.

So, but to get to the base you need to dereference GDTR and the base is actually, stored on memory. So, you actually go to memory to get the base right, when you add it to CS then you get the descriptor, then you get the base from the descriptor once again you make memory access to get the base from the descriptor because the descriptors stored in memory.

And now you add the offset to the base and compare the offset to the limit and now you do a final memory access which is base plus offset. So, you know one memory access had how many more memory accesses? Two extra memory access right one for dereferencing the base and the other for dereferencing the base of user one for dereferencing the base for the GDT and one for base of the descriptor, right. So, that that sounds very wasteful right. So, what is the typical thing that is done?

Student: Caching.

Caching right: so, you have already run your computer architecture course you have seen caching. So, what is done is when you when you execute the load GDT instruction these values get cached on chip. So, there is there is some space on the chip which basically allows you to cache these values on chip ok. And plus you know the semantics of this are that if you change if the kernel you know let us say changes the base limit to something else, then it does not immediately get you know updated in the chip because there will be a lot of logic that is required to do that.

Instead what happens is you know GDT the cache can remain out of sync from the memory. So, if you want to actually you know recache it then you should execute that LGDT instruction again at the new base and limit right. So, its it is caching, but it is not you know they do not remain consistent. So, the cached value and the real value in memory do not necessarily remain consistent you can reload the cash by re executing the low GDT instruction ok.

So, what does this kind of caching? Is it write back, write through?

Student: We have not done this (Refer Time: 15:38).

You have not done write back or write through questions?

Student: (Refer Time: 15:43) write back.

This is right back is this write through.

Student: (Refer Time: 15:52).

No, it is not written through right because I can actually go and write to the memory without the cache knowing anything right, write through basically means I have all my writes have to go through the. So, I go through the cache and I first update the cache and then all the things that I update in the cash go down. So, actually I can I have direct access to the main memory. And so, cache is actually can be out of sinks. So, it is not write through.

Is it write back? It is not right back either because, once again I actually changed the value underneath and then update by using the LGDT instruction right. So, it is not, it is neither write back nor write through its you know it is another kind of caching you know

basically there you explicitly invalidate, or explicitly load new values will be cash right. The cache is actually just acting as a read only cache for you know.

So, that the cache is accessed in only read only mode, the only way you write it through explicit invalidation of the old value by re executing the LGDT instruction or you know invalidate instruction. Its neither right back nor right true, but you know it is a different type of cache alright.

Similarly, when you load values into the registers CS DS etcetera these the descriptor values base and limit get cashed on chip right just like this the descriptor values get cached on chip. So, you do not actually make memory access to each time you make a you do not actually dereference memory to get these values each time you make a memory access you have it on the processor cache.

Once again, the semantics are that when I load it gets loaded into the cache on locally on the processor, but if you change it later on that cache still remains retains the old value alright. If you want to override that old value, you need to re execute you know you need to reload the segment register using some instruction like move let us say ax to cs right. So, when you reload the segment register the cash gets freshly loaded, but if you change the memory underneath the cash remains keeps the old value ok.

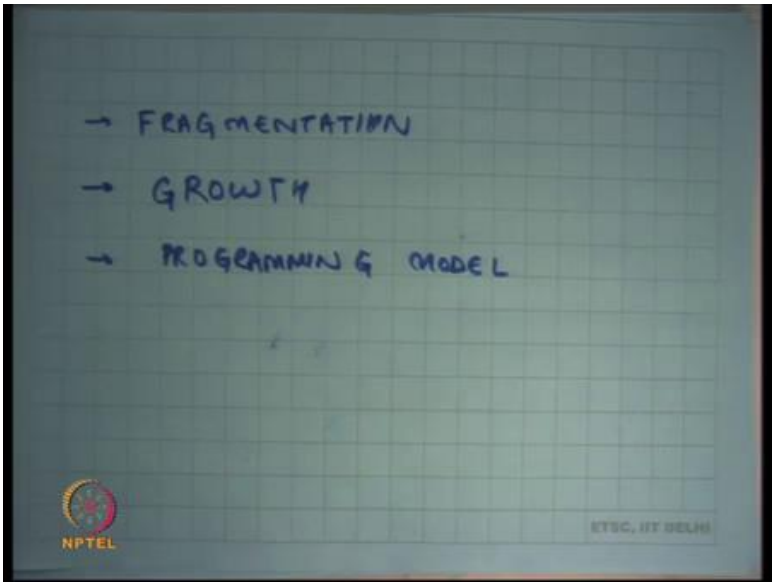
So, this is the caching model. So, with this caching model you know every memory access will actually make only at most one memory access usually typically right. Because you know each time you execute this address the GDT has base stored unlimited stored locally you can add it to the selector to get the descriptor value the descriptor value itself actually is stored in the locally. And so you can just get the UBASE from there or KBASE depending on whether it is a kernel segment or a user segment and you can and you actually just go to the memory for the real physical address and nothing else right.

So, this way you know you also have to worry about performance and so you care about performance, but this caching model also means that the OS needs to be careful that the cashed value and the memory value do not go out of sync right. So, GDT remain in the in this kind of model that we are talking about GDT remains constant. So, it only gets initialized once it gets cached once was does not need to bother really.

But and in fact, even KBASE and other descriptors are actually only remain pretty much constant it is only this user base and user limit that are actually changing and so the on every context switch what the OS will need to do is reload these segment registers again alright with that particular descriptor value. So, that they actually now get the new cash values before it actually context switch is back to the process ok.

So, that was segmentation and segmentation allow you. So, nice thing about segmentation or virtual memory in general is that it allows separation between physical address space and virtual address space. And our virtual address space can be this uniform address space starting from 0 to some value and the physical and you can place the processes anywhere you like at runtime and the same program is going to run with the assuming that it is actually running on a uniform address. But it has problems, the first problems we saw was fragmentation right.

(Refer Slide Time: 20:05)



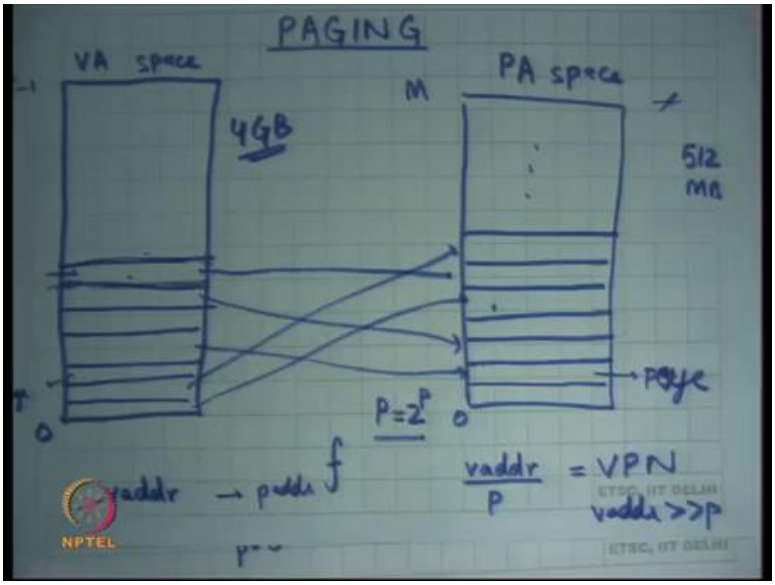
So, because a processes address space needs to be contiguous if different processes of different sizes are created, then very soon my physical memory can get fragmented and I could be wasting space. So, new processes may not be able to get admitted because you know the spaces that I have the holes that I have are smaller than the size of the process, even though the total space I have could have accommodated that process.

So, that is problems of fragmentation, it has problems with you know things like process growth. If I want to grow a process it is very difficult because I have already placed the

process somewhere and now the other processes which are you know around this process. So, now, I want to grow the process the only option that I have is either move other processes up or down or move this process somewhere else right and in both cases that requires physical copying of the entire memory of the process from one region to another in physical memory that is a lot of work.

And thirdly you know some of these problems can be alleviated by having multiple segments, but multiple segments have a problem that the programming model becomes complicate right. The programming model if you have multiple segments then the compiler has to worry about you know which segment I have to access as opposed to just saying here is an offset and that is it and I am doing that ok.

(Refer Slide Time: 21:41)



So, paging is another way of implementing virtual memory alright. Here what is done is let us say this is the physical address space, I am going to call it PA Physical Address space pa space alright. What it does is, it divides physical address space into fixed size chunks right, called pages. So, each of these sorts of frame is called a page right and then it divides the virtual address space also into pages, are the same size ok. So, this also page and then it has some function in the middle f which is implemented through a table, a lookup table which maps a page in virtual physical address to a page in virtual address a page in virtual address to a page in physical address and so on.

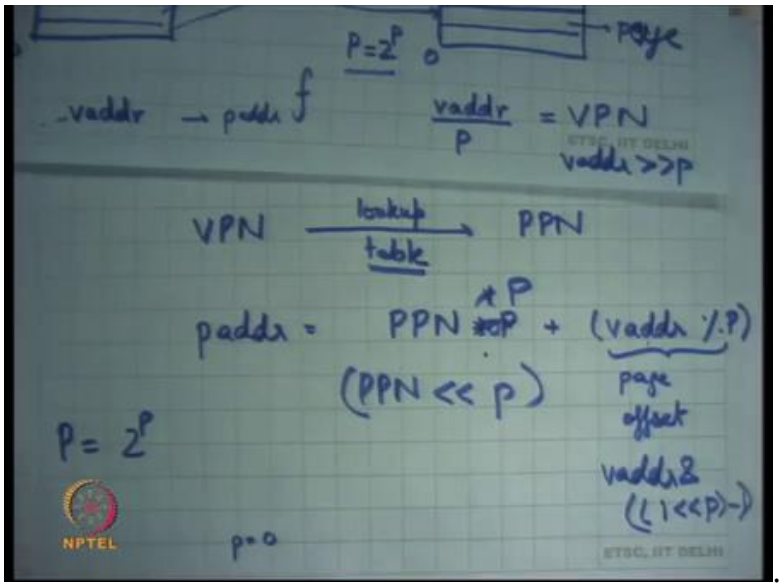
So, completely arbitrary mapping alright; so, the virtual address space also you know just like before start from 0 to some value, let us say you know let us say it goes all the way 2 to the power 32 minus 1 because that is the maximum addressing that you can do anyways right because it is a 32 bit machine. So, the maximum size of the operant that you can have for a memory address is 32 bits because the registers are 32 bits, the immediate operant is 32 bits even you execute in displace mode it gets wrapped around to 32 bits alright.

So, so the virtual address space goes from 0 to 2 to the power of 32 minus 1 and the physical address space goes from 0 to whatever maximum you have alright depending on how much memory your machine has whether has 500 and 12 MB 1 GB. So, that is the value of M right. And this function is going to map these pages in this address space to pages in the physical address space ok.

Now, the program just works as before program just assumes that it has this virtual address space, it just know once again it just says I want to access address vaddr I want to access a address called vaddr, what happens is you go you say oh vaddr is here let us say. So, you say it belongs to page number this right, you basically do some calculation to figure out is this is the page number that it belongs to and you figure out on this page number is living in this page number of physical memory. So, you do that translation and then you can convert that vaddr into a paddr ok.

So, what is the nature of this translation? Well let us say the page size was p right and let us say the page size was p and p is you know equal to 2 to the power p. So, let us say p was some multiple of or some power of 2 just to make things simple, because you know are, we are working on a binary machine where bits are binary. And so, so if I want to convert vaddr to paddr what I am going to do is I am going to say vaddr divided by capital P that is going to give me my page number in virtual space. So, I am going to call this the virtual page number VPN right.

(Refer Slide Time: 25:41)



And I am going to convert VPN; I am going to look up a table to convert VPN to PPN ok. That is a physical page number and now I am going to say now paddr is equal to PPN into P plus vaddr mod P right. That gives you a one to one mapping between assuming there is a one to one mapping between in VPN and PPN this gives you a one to one mapping between vaddr and paddr right.

So, basically you divide vaddr into a virtual page number and an offset inside that virtual page right. You convert VPN to PPN and then you add that offset. So, this is the offset this called the page offset right. So, you divided vaddr into a virtual page number and a page offset, then you converted VPN to PPN using a lookup table and then you convert computed paddr as this right you just dereference memory on that page and added the page offset to that ok.

If P is the power of 2, then these operations are very simple, vaddr by P is nothing, but vaddr shift left by p bits small p bits right and we had a %p is just vaddr and you know let us say one lesser than P minus 1 right. So, this gives you, its it is a mask which has the last small p bits set to one and everything else is set to 0 alright. And so, and PPN into P is nothing, but PPN shifted left by p bits alright.

So, now, the question is how big should this P be, or you know consequently p be? So, how big should it be if its 0, p is 0 which means P is 1, then you know I have extreme flexibility in where I can place each and every byte right, I can place each byte

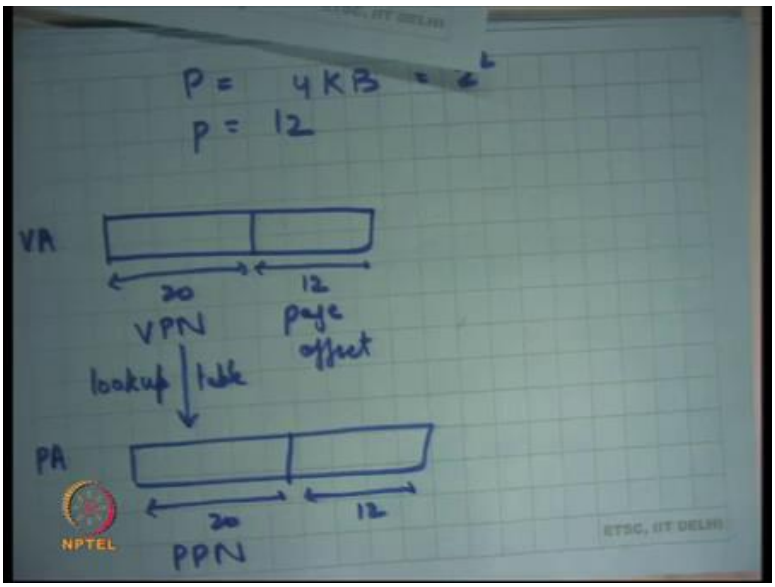
completely anywhere in the thing. But the look up lookup table how big is the lookup table going to be its pretty big right because for each byte I need an entry which maps it.

So, on the other hand if I have p as a very large number let us say P was P was equal to let us say let us say P was equal to 10 megabytes or. Power of let us say 32 megabytes ok. So, let us say P was 32 megabytes then you know firstly, I can now my lookup table becomes really small right because what is the size of this address space its 0 to 2 to the power 32 minus one which is four gigabytes right and the size of memory is also limited by whatever physical memory I have.

So, let us say if I had only 512 MB of physical memory then the number of pages I will have if I have a 32-megabyte page is 512 by 32 which is let us say 16 right. So, 16 pages in the entire thing; so, my lookup table is going to be just mapping 16 integers here to 16 integers there right. So, my lookup table is only going to need 16 entries if my pages that big.

But, the problem with that is if most of my processes are really small, then each time I allocate a pay process I need to actually allocate at least 32 megabyte of state space number 1. So, I am wasting a lot of space in the middle right and ok. So, you know here is an engineering decision to be made, what is the size of a page and the typical size of a page that is used in modern hardware is 4 kilobytes alright. So, capital P is equal to 4 kilobytes and small p is equal 12.

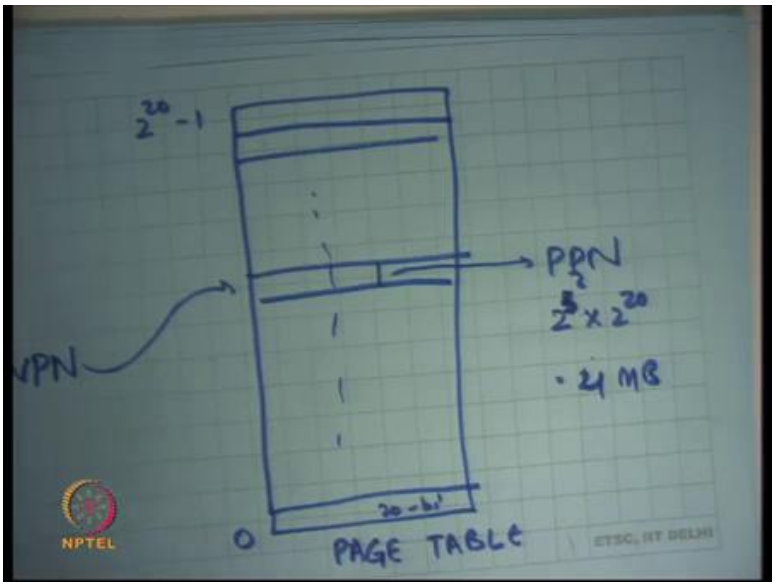
(Refer Slide Time: 30:25)



So, just 2 to power 12; so, we are going to you know as we go along the course were going to see why you know a number like this makes sense you know it actually depends on the physical characteristics of the hardware etcetera, but let us just first understand assuming there are pages 4 kilobytes, how does this mechanism work? So, pages 4 kilobytes what happens is a virtual address is 32 bits of which the last 12 bits are called the page offset and the top 20 bits it is called the VPN right.

And, similarly in the physical address this is the PPN. So, now, I need to design a lookup table which maps VPN to PPN. So, in other words I need a mapping from 2 to the power 20 numbers to another 2 to the power 20 numbers in the most general case. So, one way to do that is to have a table of 2 to the power of 20 entries right. So, I could have one big table.

(Refer Slide Time: 32:01)



Now, let us call this the page table, which has 0 to 2 to the power 20 minus 1 entries ok. And I am going to take the VPN and use it to index this table right and what about value whatever value is stored there that is going to be called my PPN ok. That is one way to do this. How big is this table?

Student: (Refer Time: 32:42).

So, each entry will need to be 32 bits right. So, that is two to the power 5 into the number of entries, yeah. So, into 2 to the power 20 that is equal to 2 to the power 23 or 8 MB.

Student: What each entry of the power 40 bits, 20 bits for the, for the source and 20 bits for the rest?

The question is should not each entry have 40 bits instead of 20 bits.

Student: (Refer Time: 33:13).

No, because the VPN is actually being used to index the thing right. And so, you save space in that sense by doing this kind of thing. The another question could have been you know why do I use this thing where I have a huge page table, where I am actually assuming the VPN can be used to index set why not have something more efficient like a link list which has all these VPN to PPN mappings right. That is likely to be smaller in the common space, but you know it is a time to actually walk through that linked list that is going to become the bottleneck.

So, as you can imagine you are going to actually do this translation on each and every memory access and. So, the most important engineering decision you to take is basically to minimize this translation overhead right. You want to minimize the translation overhead; it does not matter space can be optimized in other ways if possible ok.

Student: Why is the PPN 32 bits?

Why is the PPN 32 bits, it could have been 20 bits right, but you know 20 bits the memory itself is byte addressable right? So, if I say its 20 bits here, then the address of the first entry becomes 20th bit and does not makes I paddr access a bit in memory I can only access memory at granularity of a byte number 1 ok.

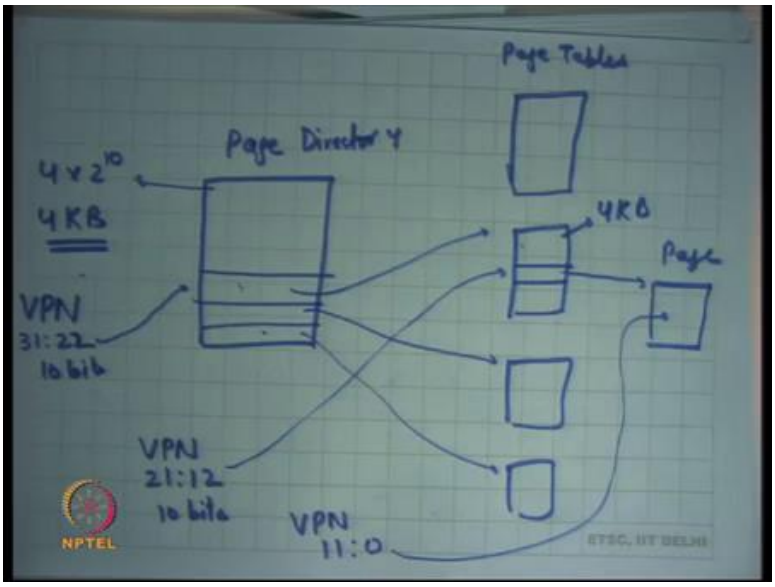
You could say why does it need to be 32 bits why cannot it be 24 bits for example, right which is 3 bytes, well that is possible, but actually that that also complicates hardware you know it is better to actually have power of 2's. And the other thing is you actually use that extra 12 bytes for other purposes extra 12 bits for other purposes which were going to discuss later ok.

. So, that is a lot space. So, that like let us say this is truth of 32, 32 MB right, I am getting it right ok. So, 32 hold on hold on. So, there is 2 to the power 2 right; so, 4 MB right that is 32 bits which is 4 bytes so 4 MB off space. So, clearly 4 MB off space cannot be allocated on chip right. We said chips can only have some registers which are

few 10's to 10's of bytes. So, 4 MB of space paddr be allocated in chip. So, this data structure has to live on memory right.

And, but if the state data structure has to live in memory you are saying that every process will have its own page table because every process will have its own virtual address physical address mapping. So, every process will have need at least 4 megabytes of space, to at least store the page table typical process sizes will be few 10 of KB to 1 MB to you know things like, processes like grap and ls all these are very small processes and you know if for every process I am actually creating a page table that is 4 megabytes of size that is a lot of wastage.

(Refer Slide Time: 36:13)



So, what is done is basically you divide the page table into a 2-level page table. The first level is called the page directory and the second level are called page tables, where each of these entries in the page directories points to a point to a page table and so on.

So, once again what I do is I look at the VPN and I take the top 10 bits of VPN. So, let us say VPN bits 31:21 or let us say 22 to 31st bit and we use that to index the page directory. I get some address this contains the physical address of the page table and then I look and then I use VPN, 21:12 to access the corresponding page. And so, I get a page and then I use VPN 11:0 to offset into the page right.

So, just I mean what I have done is I have divided the page table into a 2 level hierarchy, I said why not why not just have a list or a tree to store this and you said you know having a list or a tree is actually going to increase complexity of hardware. But at the same time, you need to strike a tradeoff between how complex the hardware needs to be and how much space you can waste.

And so you know the designers thought let us just let us just divide the table this large table of 4 MB into a tree of tables, but allow the tree to be the tree must be of size 2, or depth 2. So, use it; so in this case I was using 20 bits to index into this page table in this case I am using the first ten bits to index into the first page directory and I am using in the next 10 bits to index into the second page directory or page table.

So, this way you know if I have a process which is actually taking only you know let us 100 KB of space. So, it only needs 0 to 100 KB mapped in the address space. So, will happen is you know 1 or 2 of entry the entries here are going to be filled and all the other entries are going to say they are not mapped, right and that way you save space.

So, only one or two entries are filled in the page directory and you only have one or two page tables in the in that case and the total amount of space that you using for this page table structure is let us say 3. 3 of this right, 1 here and one of each of these let us say two of these one or one or two of these depending on the size of the process. Let us see what the size of the page directory itself I am using 10 bits to index into the page directory. So, how many entries does it have?

Student: 2 to the power 10.

2 to the power 10 right, and how big is each entry.

Student: (Refer Time: 39:36).

Each entry contains a physical address of the page table. And how bigger physical address be? 32 bits. So, it has, but a physical address can be 32 bits, but you have also segmented the physical address space in to pages right. And so, the number of pages you can have in the physical address space is 2 by 20. So, really each entry needs to be 20 bits right each entry needs to be 20 bits. And so, but once again 20 bits is you know I is a is not a very nice number. So, we extended to 32 bits and so it becomes 32 bits 4 bites.

Student: 4 kb.

Right, so it becomes 4 into 2 to the power 10 which is 4 kb.

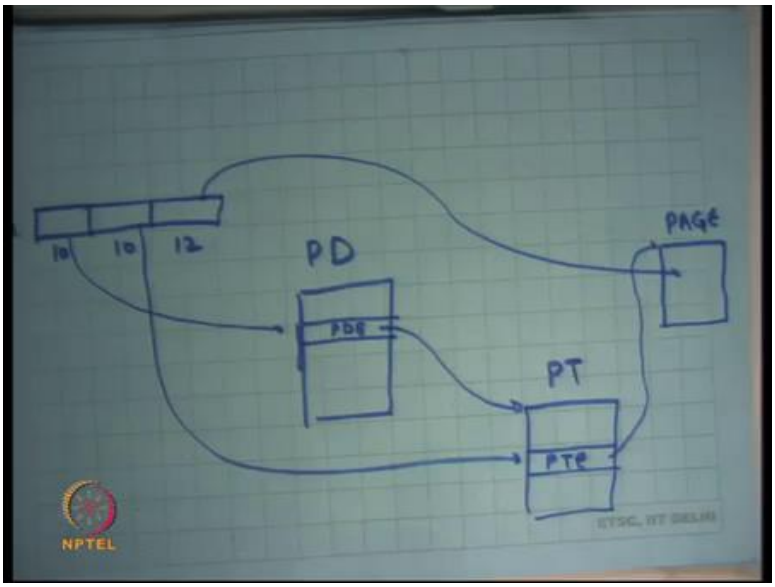
Student: (Refer Time: 40:23).

So, that is the size of page directory and interestingly that is the same size of the page right. So, the page directory itself lives on a single page. Similarly, what is the size of the page table?

Student: (Refer Time: 40:37).

Same thing right; you are using 10 bits to index in the 4-page table and each entry is 4 bytes. So, its 4 KB again, once again the page table itself lives on a single page right. And then you use that to index the page and you get the page alright. So, let me just review this, this is the virtual address VA.

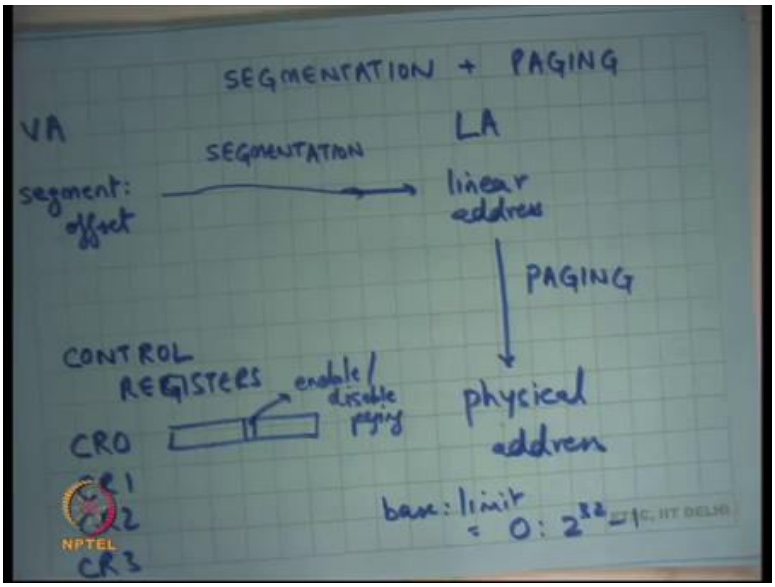
(Refer Slide Time: 40:59)



We divided into 10, 10 and 12 was 32 bit address, you use the first 10 bits to index into a page directory that I am going to call it PD Page Directory and you are going to get up PDE Page Directory Entry using this. You are going to get the contents of the PDE to dereference a page table. Let us call it a PT, and you are going to get the next ten bits from here to index into the table I want to get a page table entry right.

And now you are going to get a page from here finally, you got a page and you are going to use in last 12 bits to index into this page to get your final physical address. Let us see how this works on x86.

(Refer Slide Time: 42:13)



So, x86 actually uses both segmentation and paging alright. So, virtual address of the form segment call colon offset, this is called a virtual address VA. Go through the segmentation hardware as we discussed earlier to give you another address which we have been calling the physical address so far were actually gives you what is called a linear address alright.

So, let us call it the linear address. The reason it is calling the linear addresses because now this address has no segment colon offset, it is just 1 number with 132-bit number right. So, it is a linear address space and now this goes through paging hardware to give you a physical address ok.

. So, it gives you both an operating system is free to use both; an operating system is free to use only segmentation in which case all it needs to do is disable paging alright. So, how do you disable paging? It has certain registers which are called controller registers CR0, control registers just a 0 CR1, CR3 let us say CR2 maybe more. And, you know there is a bit in CR0 which says enable or disable paging. If you disable paging only using the segmentation hardware and that is likely to be quite fast right, because you are

not going through the paging thing which is which is involves looking through the page tables.

But actually, as you can discuss later paging is also not all that slow and you know there are ways to make that fast and this is basically what is typical us it has advantages. So, an operating system is free to use only segmentation or an operating system by disabling this bit. All in operating system is free to use only paging right, in which case what does it need to do? You know all it needs to do is basically set all its you know set its segment selector that we discussed in our first slide ok.

So, all it needs to do is set its set all its base and limit to 0 to 2 to the power 32 minus 1 right, if all its segment descriptors it sets its base and limits to 0 to 2 the power 32 minus 1 its effectively disabled segmentation.

.So, in the in the next few lectures I am going to assume that the base and limit have been set to 0 and 2 to the power 32 minus 1 which means the virtual address is equal to the linear address. And the most operating systems actually do this right. So, most operating systems like Linux, Windows and the operating system that you are going to do in this class which are x86 (Refer Time: 45:33) and pintos will all set base and limit to these values. And so only use paging for implementing virtual memory ok.

Student: (Refer Time: 45:6).

What is the use of having segmentation? Well the hardware designer has given you a choice basically right, and the hardware designer designed the hardware before the operating systems were written on it. So, there was basically tried to say that I do not know what the operating system may want in future.

And so, he designed it and because of backwards compatibility it remained in the hardware. In fact, on the 64-bit architectures that have you know come up recently segmentation is not present. So, you do not have segmentation in this form, it is a very thin form of segmentation that you need because most operating systems are not using segmentation.

Alright; so, I will stop here and we going to discuss more about paging in the next lecture.