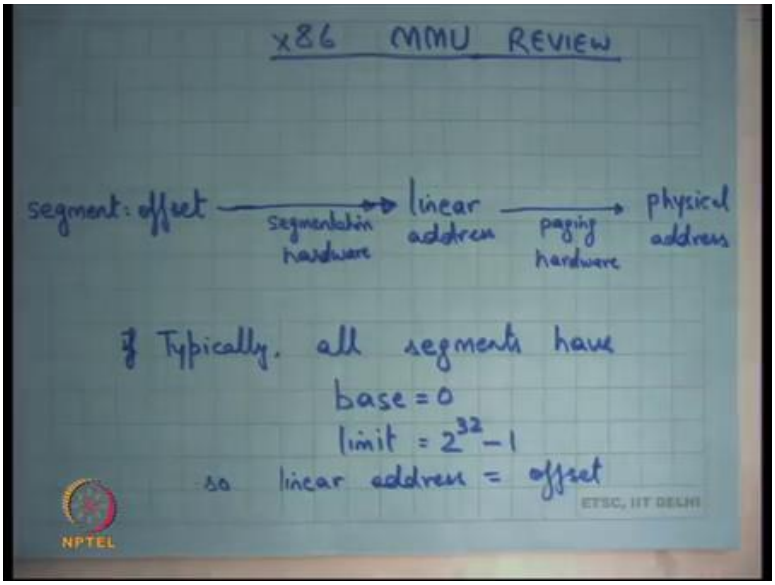
Operating Systems
Prof. Sorav Bansal
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture - 11
Paging

Welcome to Operating Systems Lecture 11. So, we have been looking at the Memory Management Unit in a typical hardware system and we have been looking at the x86 MMU in particular.

(Refer Slide Time: 00:35)



And we said the x86 MMU implements both segmentation and paging right. So, a virtual address is always of the form segment called colon offset. Now CS: offset, DS: offset etcetera. The segment will point to a segment descriptor. We have seen this in detail and then, the offset is going to be added to the base which is stored in that segment descriptor to get a linear address right.

So, far we had been calling this is a physical address, but it really is a linear address if paging hardware is also enabled. So, further this linear address goes through another level of translation which we call paging, to get you a physical address ok. Most operating systems or most systems in general, actually today do not use segmentation hardware for this kind of translation; I mean in the way that it in the sense that they all

they typically said base is equal to 0 and limit is equal to you know whatever the maximum value is in a on a 32-bit machine for all the segments.

So, you get a flat address space, irrespective of what segment you are going through, whether you are going through CS, SS, DS does not matter; it is the offset that counts right. So, basically effectively that means, that your linear address becomes your offset ok. You could imagine even operating system which is actually using seg only segmentation and not paging.

We have discussed this previously right; in the discussion that I had been having so far when we did not discuss paging, we said you know it is possible to implement virtual address spaces using segmentation and in which case you know linear address becomes equal to physical address. And, it's possible to do this, you just keep swapping the users descriptor and keep changing its base and limit depending on which process is getting loaded right.

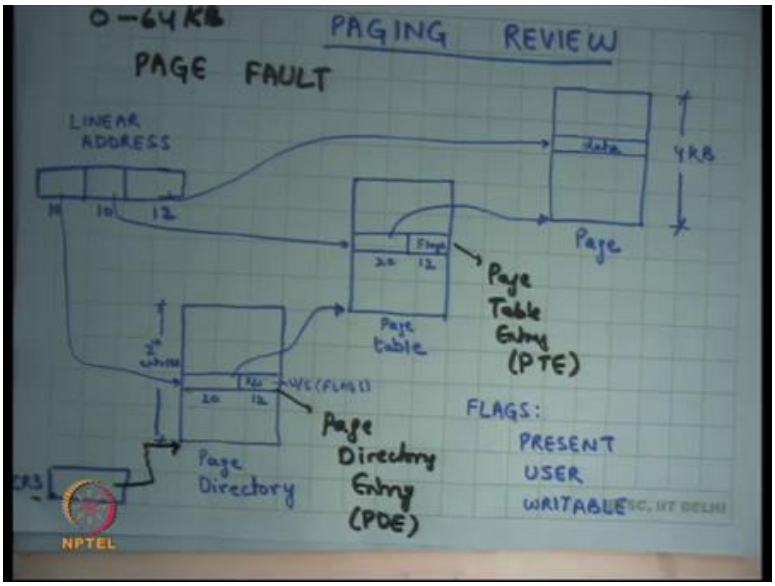
So, it is possible to implement a complete MMU using only segmentation. The only problem is it has some drawbacks and the drawbacks are for example, fragmentation becomes a problem, growth of a process is very tricky. And, plus you know if you are using multiple segments, then the programming model has to worry about which segment I am currently accessing and all that right.

On the other hand, if you have a flat segmented model which mean which all the segments are have exactly equal base and limits, in that case you are you know you the fragmentation problem becomes more severe alright.

Yet another operating system can just use paging hardware in which case he just sets you know all these things to thing. It still uses a segmentation hardware for things like you know for understanding what a privileged level I am executing on, yes recall we call that the last 2 bits of the CS register, basically indicate whether I am executing in privilege level 3 or privilege level 0.

And so, that still holds, but apart from that the offset pretty much has a one to one mapping to the linear address alright. So, base is equal to 0. So, in which case he is only using the paging hardware and yet another operating system could try to use both right, that is also a possibility alright. So now, let us look at paging in more detail.

(Refer Slide Time: 03:37)



So, we say look paging gets rid of some of the problems of segmentations. The main problem was with segmentation was that it required the entire process to be in one contiguous chunk in the physical memory. It is an entire process memory should be in one contiguous chunk in the physical memory and so; it was creating problems with fragmentation and growth.

So, as opposed to that if it was possible to have a mapping between virtual addresses to physical addresses such that the flexibility is more in which case you know a process could be sprinkled across that physical address space. But, in the virtual address space, it maybe contiguous and you could basically have mappings, arrows going from the virtual address space to the physical address space right.

And, we said you know it is possible to do that, you know one has to design it carefully if; so, basically at what granularity are you going to do this mapping. So, we said let us call the granularity at which you are going to do this mapping a page right. So, for every page; so, within a page contiguous byte in the virtual address space will be also contiguous in the physical address space, but across pages that need not be true.

Now, question is how big should a page be? He said you know page, page cannot be so small that the data structure to store this translation is so big, that it is actually bigger than the actual address space itself that is you know that is ridiculous. On the other hand,

it should not be so big that you know you are wasting a lot of space. So, we said you know let us say that 4 kilobytes are a reasonable value for a page and then we said ok.

Then, if 4 kilobytes are a reasonable value for a page, what is the maximum size of virtual address space can be? It can be 2 to the power 32 right. So, 2 to the power 32 divided by 4 KB comes to 2 to the power 20. So, you can have utmost 2 to the power 20 pages in your virtual address space. Similarly, your physical address space let us assume can be utmost 32-bit flight. Let us assume that the bus can only take a 32-bit address on the physical bus which goes to memory. So, your physical memory can also be up to 2 to the power 32 bytes let us assume.

Although, you know it may be smaller depending on actually how much ram you have in your system. In any case the hardware needs to be needs to provision for the maximum possible. So, let us 2 to the power 32 on both sides. So, what is the number of maximum number of pages on both sides? 2 to the power 20 and so, you need a mapping from a set of 2 to the power 20 numbers here to another set of 2 to the power 20 numbers here right.

And so, this mapping we said is you know if I was to do it in one contiguous table, it will require 2 to the power 20 entries. So, I can just have an array which has 2 to the power 20 entries and each entry basically stores where this page is mapped.

This array will actually be pretty big it will be 4 megabytes if I were to do this and so, assuming that each process has a separate address space, each process will need a separate mapping. And, so for each process I will need to have a separate data structure of this type; but this sounds very large you know 4 megabytes for every processes a is very large because typically processes are smaller.

So, let us divide this mapping into a 2-level hierarchy right and the first level we are going to call a Page Directory which is going to create a mapping from the first 10 bits to a page table. And then, the second page and the page table that you obtain you are going to use the next 10 bits to get the actual physical address. We have divided the 2 to the power 20 entries into sets of 2 to the power you know into is divided 20 bits into 2 into 10 bits 10 bit look up.

So, the advantage of this is let us say a process has only is only mapping you know 64 kilobytes of space right. So, 64 kilobytes of space can be captured in you know 64

divided by 4 that is 16 pages right. So, 16 pages can probably just you know 1-page table can have up to 1024 entries. So, 1-page table can actually have 1024 pages. So, in just you just need to allocate 1-page table and fill in those 16 entries here and just have one entry in the page directory that is pointing to that table at the corresponding offset right.

So, let us say my process was having an address space from 0 to 64 KB alright. So, what will happen is the 0th entry in the page directory will point to a page table and the 0th to 16th entries in the page table will point to pages, all the other entries will be invalid ok. So, that way a process which was only 64 kilobytes large needed a page table structure of 8 kilobytes right, 2 pages each of these is a 4-kilobyte structure. So, you needed 1-page directory and 1-page table.

So, you know you needed 8 kilobytes to store the mapping for a 64 kilobyte process which has a contiguous mapping from 0 to 64 byte kilobytes; so, significant improvement over the 4 megabytes that we had earlier right. It is a simple thing you just divide 1 linear array into a 2-level tree and you basically only have pointers where there exists a mapping and for others you just say its invalid ok.

So, of course, the downside of dividing 1 single table into 2 is what? Number of lookups right. So, number of lookups that you need to do in memory or number of dereferences that you need to do in memory has doubled. Earlier you just had to do 1 dereference to get the physical address, now you have to do 2 dereferences.

First you have to do 1 dereference here, then you have to do a dereference here before you actually get a get to the data right. So, here is how it works on x86, there is a special register called the controlled register 3; we said there are some special controlled register then x86. So, there is a special register called controlled register 3 which points to the base of the page directory.

And then, you know any linear address that is computed through the segmentation hardware, the first 10 bits are used to index into this page directory to get a page table. The next 10 entries; bits are used to index into the page table to get the page and the last 12 bits are indexed into the page to get the real data right. The physical address is actually this value plus whatever this offset is right that is that will be the physical address and that is where the data is going to live in physical memory.

Student: Sir, why?

Ok. Why does the pointer to the page table need to be 20 bits? Great question. Because in.

Student: Each page table is also a page.

Each page table is also a page you have basically divided your entire physical address space into pages right and you have mandated that page directories and page tables will also are also constrained to start only at page boundaries right. A page table cannot be across 2 pages right. So, you basically just statically created a partition of you know you are basically said every 4 kilobytes you have drawn a line and you have said that a page table has to live in one of those slots it cannot actually you know straddle 2 slots ok.

So, because and so, the number of bits you need to actually understand what which slot you have is only 2 to the is only 20 bits. So, similarly you know the page directory needs to be really only 20 bits. The page table can be addressed using 20 bits and the page itself can be addressed using 20 bits ok.

We also said you know even although these pointers are 20 bits, it makes sense to actually allocate 32 bits for the entire entry. So, this entry is also called the page, this is called the page directory entry, and this called the page table entry alright.

So, let us use the words PDE and PTE for them alright. So, both PTE and PDEs are 32 bits long. So, the last 12 bits are can be used for other purposes and in particular, these last 12 bits are used to store flags right. For example, just like in segmentation you can where you could store you know whether this segment is allowed to be accessed in unprivileged mode or not. You can say whether this page is allowed to be accessed in unprivileged mode or not alright. So, that is the user flag.

Firstly, there is the present flag whether it says whether the page is actually present right. So, so in this example let us say a process has only 64 kilobytes of mapped space from 0 to 64 kilobytes and it is a linear address. Then, the only the first entry will have the present bit set; all the other 4 you know 1023 entries in the page directory will have the present bit zeroed out alright. That is what it means that the entry is actually invalid.

Similarly, in the page table the first 16 entries will have the present bit set, all the other entries will have the present bit set to 0; so, that is a bit which says whether there is, whether this page table entry actually has means anything or not; whether the hardware should actually consider walkthrough dereference it at all or not alright.

Then, there is another flag which says user which basically says whether I am able to whether I should dereference this page table entry, only in supervisor mode or am I allowed to dereferences page table entry either in supervisor or in user mode right.

So, you can do it either in user or privileged mode and the third thing which it has is whether this page is writeable or not right. So, it gives you another sort of added capability. You can actually the operating system can actually map certain pages as read only right and we will see you know why this is a very useful thing to have.

So, if a page has a writeable with zeroed out, then if a program tries to write to an address that dereferences through that entry, it should generate an exception alright and this kind of an exception is called a page fault. So, instead of a segmentation fault, now you are talking about paging. So, the new name is called a page fault. The meanings are similar. Basically, you are trying to access an invalid address, or you are trying to address access a valid address in an invalid way right.

So, a page fault can occur if you are trying to access a non-user page in user mode or if you are trying to access a non-present page or if you are trying to write to a non-writeable page, you know either of these basically mean that you know you generate a the hardware will generate a page fault.

And you know the page fault handler will do the similar thing. For example, it may kill the process, or it may you know convert the page fault into a signal that it gives to the process all these just like before right.

Student: So, is this page table are completely managed by the hardware or does OS also has role to play in it?

Ok. So, yeah, great question; are these page tables managed by hardware or does the OS have a role to play in it. The OS sets up these page tables. So, the OS sets up the CR 3

pointer for example right. The OS sets up the contents of the page directory. The OS sets up the contents of the page tables.

Before it transfers control to the process, but for each memory access the OS is not coming into picture. It is a hardware that walks this page table or the hardware that reads these page tables right. So, this process of actually converting a linear address to a physical address, it is called page table walking or walking the page table.

So, the walking off the page table is actually done by the hardware. The operating system sets up the page table ok. So, the hardware walks the page table and if there was an error, for example you know you are trying to read an entry which is not present; then it is an error page fault right. So, yeah so, basically each of these flags. So, when I say there is a flag in the page directory entry, let us say the present flag in the page directory entry is not set, what does that mean?

Student: It means (Refer Time: 15:38).

It means there is actually no page.

Student: No mapping.

In this, there is no mapping here right. So, if the program actually tried to access a linear address that has a first 10 bits map pointing to this entry which has a present bit not set, then you will generate a page fault right there. It does not need to do a second level page look up at alright; similarly, user and writeable.

So, I mean you know so it is possible that this entry was present, but it says this entry is not writeable. On the other hand, the page table entry corresponding page entry table says writeable; you know one of the entries says not writeable, the second entry says writeable. So, the net effect is basically the and of these two which means not writeable.

Once you know conversely this could be writeable and this could not be this could be non-writable in which case overall it is non-writable. So, you know the OS can do it do this in a very fine grain level at the page granularity. You can do all these writeable, you can mark all these writeable and then, you know only pages which are not writeable in this particular entry that those are non-writable otherwise others are writeable and so on; same thing for user and supervisor right. So, you could have a you know the page

directory entry itself could say I am I am a user mode entry which means anybody can access me, but some of these entries are user and some of these entries are supervisors.

So, once again you take the and of these two. So, if both of them are supervisor, only or both of the only both of them only a both the user bits are set can the user access that page right. If only if for any of these entries of user bit is not set, then the hardware should not be should generate a page fault, if it is executing in unprivileged mode alright. So, let us see how paging can be used. So, we saw how segmentation can be used to implement processes question.

Student: What is the difference between segmentation fault and page fault like if we are not using segmentation then?

What is the difference between a segmentation fault and a page fault, actually at the Unix abstraction level a page fault is converted into a segmentation signal sigsev right? So, both of them actually mapped to the same thing ok, but at the hardware level you know on x86 different exceptions are used to indicate different conditions. So, there is a separate exception number for a page fault and there is a separate exception number for segmentation violation.

Student: If we are not using segmentation (Refer Time: 18:10) you should have (Refer Time: 18:13).

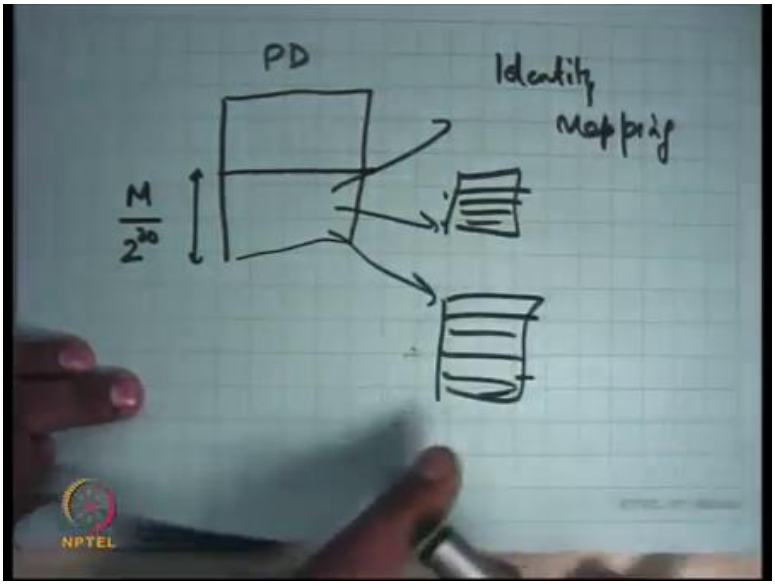
If we are not using segmentation, a segmentation violation should never occur. Well, that is not true because you know you are not using segmentation in the sense that you are not using it to segment memory, but you are still using the privilege bits or the you know the permission bits in the segment descriptor. So, you know if you try to execute for example, the user tries to load into the segment selector a privileged descriptor.

So, that will still cause a segmentation violation ok. So, it is still possible to have a segmentation violation even in this flag segmentation model, where you know you basically when the entire space into all the segments alright.

So, let us see how paging can be used. Well, one very simple thing is you set up the page table to basically have an identity mapping right. So, linear address x always maps to physical address x right and so, that is one way and how will you do that? You will just

say oh how big is my physical memory, let us say 0 through m. So, you know you will basically say 0 through M.

(Refer Slide Time: 19:19)



Let us say this is my page directory. I am going to set up you know the first M by 2 to the power 20 entries here right and then, each of these are going to point somewhere and each of them is going to set up all its entries, appropriately right to exactly the same address. So, you could set up a linear completely identity map between your linear address and your physical address right.

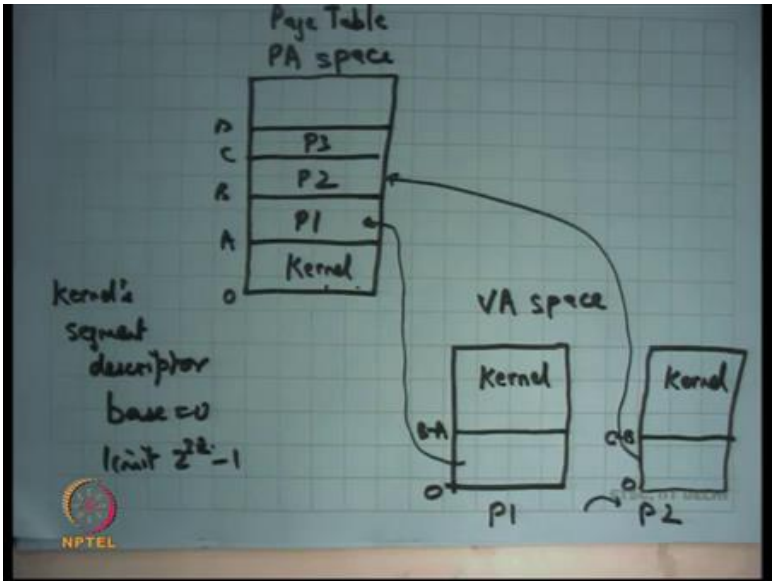
That is probably you know that is probably what you will do if you are not implementing multiprocessing and you are not having multiple processes and all that. For example, if you are you know using your system, your processor to implement some kind of an embedded system like for example, something which does not expect multiple processes to be running.

It is just doing when run the OS in one way and you have this paging hardware, you would probably just set up an identity mapping and or alternatively you could even disabled paging and so, it is the same effect right.

So, just to get a just to understand that you know you can how you are going to do this, it sorts of increases your understanding of how paging works. So, identity mapping is easy.

The other thing you can do with paging is you know set it up like segments right. So, you can say ok. So, here is my page table.

(Refer Slide Time: 20:53)



And let me let me just assume that you know it is a uniform sort of address space and so, what I am going to do is I am going to say let us say this part of the. So, let us say this is my physical address space; this is my PA space alright.

So, let us say this part of the PA is a or let us say this part of the PA is reserved for the kernel and this part of the PA is reserved for process P1, this part of the space is reserved for P2, P3 and so on and let us say this is free ok. Just like what we had in segmentation and we were implementing and using base and limit. Let us what I am going to show you is basically that paging is more general than segmentation because you know you can easily do this and paging and so on right very clear.

What you are going to do is basically say that in your page table. So, let us say this is physical address 0 and a this is you know something let us say A B C and D right. So, in your page table and your page directory, you are going to say let us say in your in your let us say this is VA space.

In your VA space, so this was a PA space, and this is the virtual address space. In the virtual address space, each process should see a see the same set of addresses. So, what you are going to do is in the VA space, you are going if P1 is loaded; then you are going

to map P1 to this space right. You are going to set up the page table such that 0 to B minus A of P1 maps to A to B in physical address space right. Easy to set that a you just set up the page directory entries and page table entries appropriately.

Then, let us say you switched to P2, you will load a new page table alright and how will you load the new page table? You will change the CR 3 register right; you will change the CR 3 register. So, a new page table gets loaded and the new page table will have a similar thing, where it will say 0 to let us say C minus B and it is going to map this to this alright. So, you can just so in the segmentation model, each time there was a context switch we were overwriting the segment resist descriptor with a new base.

In this case, each time there is a context switch; I am overwriting CR 3. So, that I have I have a new page table loaded and I have set up the page table to basically emulate exactly what the segmentation hardware would have done.

Student: Sir, the changing CR 3 where changing page directory not page table right?

Yes, we are changing the page directory and also the page tables. So, we are changing the entire 2 level hierarchy right; so, assuming that there is no sharing between different page directories right, each page directory. So, you change the page directory and consequently you change all the pointers of the page directory also right ok. So, your when you context switch in the paging world, we are going to replace the CR 3. So, you are going to have a new page directory and you can you know set up your page directory or page table to do this mapping just like you were doing in segmentation.

The only difference is that in this case you know the page tables will be different right. In 1 case the page table mappings that entry is visually present in the page table, you know the or the pointers in the page tables will be different. There will be pointing to different regions in the physical memory basically question.

Student: Sir, sir if they are using two separate like for each process separate page directory and page table, there are 2 level hierarchy, we have to like in case we update one of them we have to make sure that those physical address or not mapped in (Refer Time: 24:46).

Yes. So, if you know each process has a page directory and a page and a and page table structure and let us say I update one of them with a new pointer. I have the OS must ensure that this pointer wherever this page is pointing in physical memory that is not it does not that that page is not being pointed to by any other page table structure right.

So, that is that is possible for OS to do. No problem. The OS just keep some there is some bookkeeping, at these pages are belonging to this process these pages are belonging to this process and there they are all you know disjoint sets of physical pages and so and you then create mappings appropriately alright.

So, let us assume that all processes are completely disjoint set of pages in physical memory and you are going to just set up your page table in such a way that the page table of process P1 is going to point to which pages. And, process page table of process P 2 is going to point to which pages and each time you context switch you are going to overwrite the CR 3 register, yes.

Student: Sir.

So, for in the GDT, there is a u base and u limit for each process, yes ok.

Student: The segment register will point to that location; they will not be going to change?

Yes, all the let us say all the segment registers will always point to u base and u limit when the processes execute ok.

Student: So, that u base and u limit you should point to page directory?

That u base and u limit will not point anywhere ok. ubase and ulimit will only be used to compute a linear address right. So, from the offset, it will compute a linear address and now, the hardware what will point to the page directory is CR 3, is a completely you know separate register. You are going to use the CR 3 to actually look into the page directory, use the use the linear address computed through your segmentation hardware to index into CR 3, the first 10 bits and so on alright.

So, except that there is one thing, so this is this sound very similar to how it was done for segmentation right; the only difference really is in this case I am changing the page tables themselves. In that case, I was just changing the segment descriptor right.

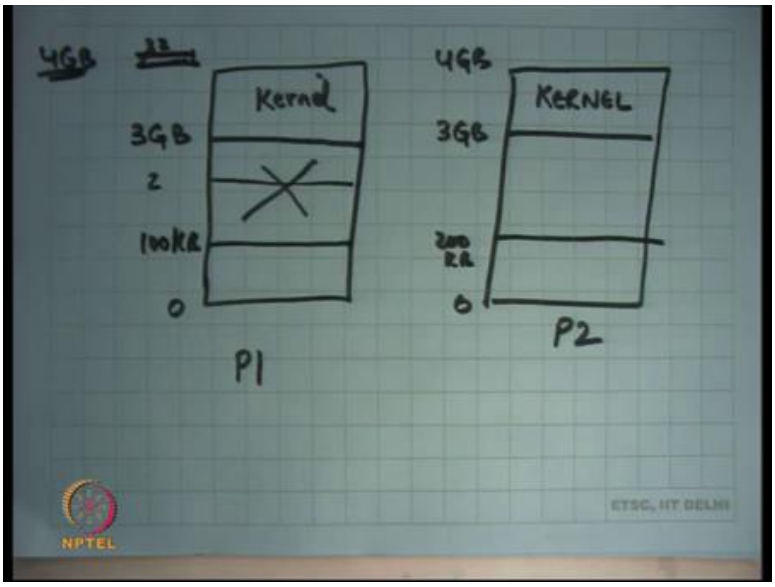
The difference here is that the kernel is also you know I said that in the in this model let us assume that the segmentation is completely flat. So, even the kernel's segment descriptor also has base equal to 0 and limit is equal to 2 to the power 32 minus 1 right.

So, the kernel is also you know the kernel is also not using the segmentation hardware at all let us say right. In which case you know somebody else, so the virtual address to physical address translation is just going to take the offset and make that the linear address and now, the linear address has to be converted to physical address by a paging hardware right.

So, the kernel's linear address is likely it is possible that the, so the kernels linear address needs to be converted to physical address by the paging hardware right. And so, what you do is you inside the page table of every or inside the VA space of every P1 , you also mapped the kernel somewhere right.

So, the kernel shares VA space with the process ok. Let us see you know this, so basically let us take some concrete examples; how does it work in Linux alright. So, how does it work in Linux?

(Refer Slide Time: 28:25)



Every process has an address space which can go from you know by definition on a 32 bit machine, you can utmost have 0 to 2 power 32 minus 1 except that you say that a process is not allowed to access the entire 0 to 2 to the power 32 minus 1; if you are compiling a process for Linux or if a process should be able to run inside Linux legally, then it should never access an address beyond, so, you know this let me call it 4 GB just to make it more readable.

So, 2 to the power 32 minus 1 is let us say a 4 GB and so, you should not ever access an address above 3 GB right. So, the process is constrained to access addresses only within 0 and 3 GB. The addresses from 3 GB to 4 GB, the virtual addresses are reserved for the kernel. So, that is where the kernel is mapped ok. So, let us say this is P1.

So, typically what will happen is P1 is mapped 0 to let us say some address you know 100 KB and all this is not used right and then, there is P 2 which also has 0 to you know let us say 200 KB this time and, but also has them kernel mapped from 3 GB to 4 GB right.

So, the kernel also ensures basically while setting up the page table that its own pages are always mapped in a certain address range of the VA space. So, if for example, there is a transition from user to the kernel and the kernel wants to access its own data structures, it can just use one of these addresses from 3 GB to 4 GB to start to try and to start and accessing its own code and data for example, because we did in this scenario we are not using segmentation at all.

What is happening in segmentation? Each time there was a switch from a user to kernel CS was overwritten and CS had a different base right and so, that way the kernel you know you are now actually accessing kernels memory and not-users memory immediately.

And you know the first thing the kernel will do is load other segment registers so that you know apart from the code other data, other segments are also pointing to the kernel. But in this model when everything is flat and the kernels also using a flat model, the user is also using a flat model, when you switch from when you switch CS, you are still in the same linear address space right and but you need you need to say that now I want to access kernel’s data or and now, I want to access kernel score.

So, that the way it is done is cs the offset of cs is still 0, but the EIP there is certain EIP's that are reserved for the kernel. So, in particular the EIP is above 3 GB are reserved for the kernel on Linux right. So, in the interrupt descriptor table for example, the CS colon EIP, the cs base of CS if you are if I am not using segmentation the base of CS will be 0, but the EIP will be some address above 3 GB typically, that is where the handler will live right. And so, so because the offset is above 3 GB, the linear address will also be above 3 GB and now it will go through the paging hardware and you will basically go reach the kernels space.

It is the job of the kernel to ensure that the paging hardware translates the address 3 GB and above to the right place in the physical memory and that it can do by setting up the page table in such a way. So, each process is a page table, not only maps that processes address space, but also maps the kernel's the address space at the top in Linux alright.

On Windows you know this is typically the model followed in many operating systems, similarly in windows. Windows seem to you know sometimes need more space. So, it can actually take 2 GB, you know it actually takes the top 2 GB for its own kernel on the 32 bit machine, where it actually allows you to actually change it if you if you want right.

So, depending on so and this is part of the specification of the operating system. So, if you are compiling something for Linux, then you should ensure that your application will never access an address above 3 GB ok. So, it is part of the interface. The compiler should be aware of that or you know if the programmer is assembly programmer you should be aware of that. And how do you do protection? how do you ensure that the user is never able to access kernels data or execute kernels code?

Student: Sir, its flag.

The flag right; the user flag; so, the user flag is off for these mappings of kernel. So, for the all the kernel mappings the user flag in the page directory entry and the page table entry is 0; for all the other entries the user flag is 1. So, the user can never access that kernel. But if there is a there is a trap, then CS gets reloaded and the last 2 bits of CS will become 0 and now, it can access the upper regions of memory virtual address space question.

Student: Sir. So, these pages are common across those processes like (Refer time: 33:42)?

Right. So, just like we saw that in segmentation the kernel remains constant right, the kernel does not move the kernel in the physical address space remain sort of there and the mapping for the kernel remains constant. Similarly, the mapping this remains constant; so, the same entries which get copied in every page table or every page directory and page table of every process.

Student: Is it a kind of an exception for overrule that every page should like point to a different physical entry?

It is a yeah. So, it is a so, here is an example we have we are breaking the rule that every page table should be pointing to a distinct set of pages right. Here is an example where 2 different processes page tables can be pointing to the same page in physical memory. Yeah, great point and actually there are more examples which we are going to discuss later. Question?

Student: Yes sir. These kernel address space; does this also have the kernel stack for that process?

Does this have also the kernel stack for the process, the kernel address space?

Student: Yes, sir.

Yes.

Student: So, when the kernel stacks for two different processes, different from each other?

Right. So, let us look at how the kernel stack gets switched. So, we said that if there was a trap while I am executing the user mode, both the CS will get overwritten, and SS will get overwritten, EIP will get overwritten and ESP will get overwritten and so, what does the OS needs to ensure that the EIP is an address above 3 GB and the ESP is also an address above 3 GB and within this kernel space, you could be having multiple stacks; 1 per process.

So, actually you know in the kernel space you are actually having all the stacks of all the processes, all the kernel stacks assuming the process model of the kernel. So, in the kernel space you will have all the stacks of all the processes, but it is all inaccessible to the user. The user cannot access it right.

So, it says that the kernel has mapped its entire data structure and code into the user address space. So, if there is a trap and a flat segmentations model, then you straight away go to that to the kernel address space in both for EIP the program counter and the ESP the stack pointer ok. And, then you also change your other sort of segments and all that ok.

Question is why do I need the entire kernel to be mapped inside the process address space; could not I have just mapped the kernel stack in the process address space? Just one you know let us say whatever size I want to have for the stack kernel stack that is what I am going to put it in the, but what about the code? You know so the program counter, you will also need to put that. So, the handler needs to be there right.

And so, and the handler is going to make more calls. So, you could potentially say that look I do not you know this 1 GB of space taken away from the process seems too costly and so, what I want to do is actually have a very small sort of handler inside which is not 1 GB, but let us say you know few 100 KBs and that handler is actually immediately going to switch the page table and then, he is going to execute the kernel logic that is a possibility right. There are pros and cons to doing that. This seems this actually works this is 1 of the best performance kinds of designs.

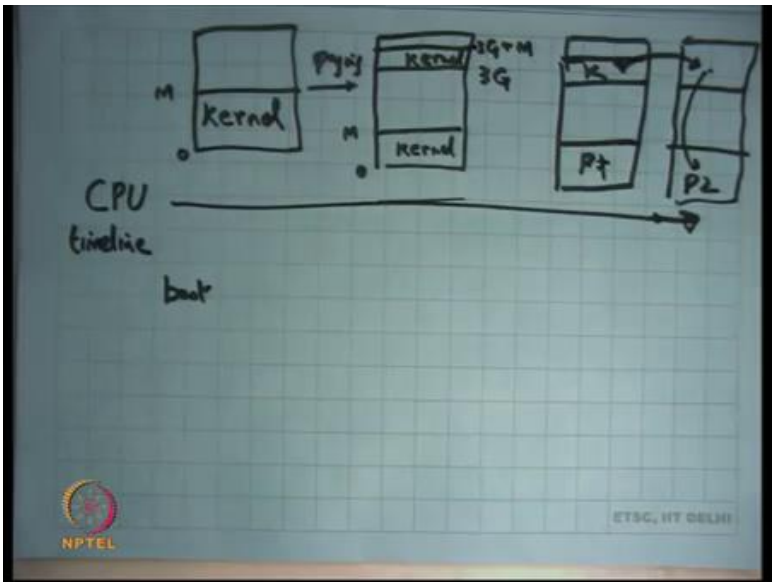
So, you know there have been multiple designs of operating systems that are being proposed, but this is this is sort of the most popular design and one reason this is the most popular design is it is the most performing design or I should qualify the statement. This is in a controversial statement may not be the most performing design, but it is the easiest to get performance out of this design alright and this is a design that you know your mainstream kernel use, Linux, Windows, x86 is you know what are you going to we were going to be starting next starting next time, on which base is going to be your programming assignments alright.

So, basically this is how. So, the kernel basically maps itself in every process address space and you know if there is a trap, it can immediately transition without having to do

anything and now kernel can now execute happily inside the process as possible. This is this is basically there is one difference that the kernel needs to be aware that it should only be accessing you know; it should be living in a certain address range in segmentation case.

The kernel could have mapped itself from 0 to something right because you know CS base has been changed. So, you know you are pointing somewhere else. But in this case the offset needs to be within a certain range 3 to 4 GB let us say on Linux ok. So, let us see what happens on a timeline just like we had seen for a let us say this is boot time, just like we had seen for the process.

(Refer Slide Time: 38:23)



So, what happens is you know at boot time it will first set up its segmentation table and now in this space, it will just set up the segmentation tables to completely identity mappings 0 and 2 to the power 32 minus 1. Then, it will set up its own page table which will be let us say an identity mapping and it will enable paging in the hardware right.

And so, it is the kernel page table that is getting loaded here; then, the kernel will load itself in a certain address space. So, the kernel has been compiled to assume that it will be living in a certain address range right. So, the kernels EIP and all these the program counters will always be in that address range and so, when it loads the kernel and the core kernel starts executing you know. So, let us say. So, initially you start executing in

physical address space. So, kernel is living from 0 to whatever maximum value you have.

Then, what the kernel does is it load itself from a you know while its executing, it loads itself in the top address space also. So, it first set up sets up page table. So, enabled paging it sets up the address space and now it says let us map myself in the new page table at the top.

So, it is going to map itself on the top also. So, it is going to say 3 GB to let us say 3 G plus M right. So, it is going. So, this is where it gets loaded and now, now it starts running from here and now it is you know context switch is to the first process, it starts the first process and so, the kernel lives here I want to say K and the process let us say lives here P1 right.

And then, at again context which is when it context switch is it actually you know it executes. So, when a context switch is a trap occurs while it was executing here; when the trap occurs its starts executing here, here it does the appropriate changes in CR 3 in the page table.

So, when it does the page table change it actually reaches here. It has changed the address space its change CR 3 and now it calls return to get to presume execution of process P2 alright. So, process P1 is executing and user mode, a trap occurs, you switched to kernels address space in the same page table.

The kernel in that page table while its executing that page table, creates a new page tables for process P2 if needed; maps itself in that new page table, switches the page table because it had mapped itself it exactly the same addresses when it switches itself you know it is not like the carpet had been dragged out under its feet because the same addresses are still valid right.

So, the new page table has the kernel mapped in exactly the same place where it was mapped in the old page table. So, when you switch it does not matter right. The kernels can still execute it just the user space address space that has changed and now, you can return back using the added instruction to the process and now the process P2 can start executing.

Student: Sir, but kernel they got already a kernel space in the in the P1 page table. So, why did you tend to do a context switch?

So, let us say you know the kernel has decided that it wants its context switch from P1 to P2 alright. So, it needs to load the new P2 page table, but the nice thing is that P2 page table also has a kernel mapped at exactly the same position. So, when it does, when it overwrites CR 3 its addresses are still valid. So, the next instruction pointer will exactly go to the same physical location still right.

Student: Sir.

And now the next thing it will do is basically a go to user mode and this time it will see P 2 there.

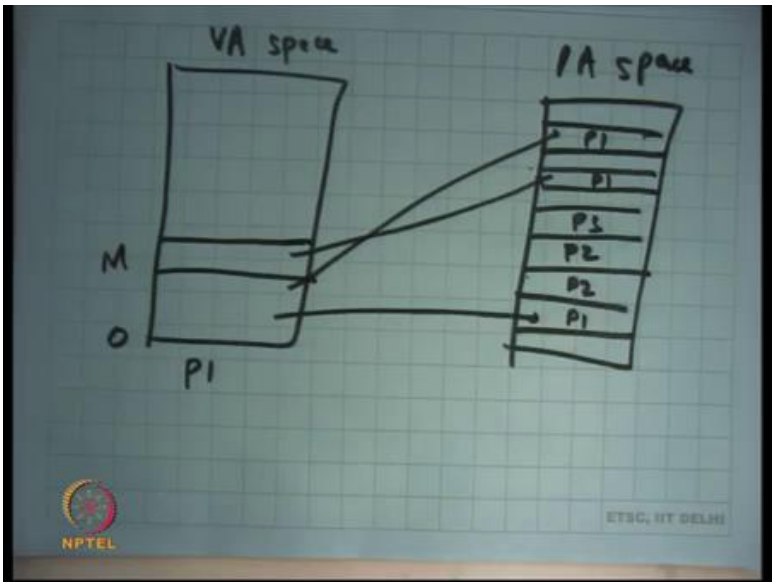
Student: Sir. So, the all the kernels spaces are each of the processes are exact replicas of each other?

All the kernel spaces in each of the processes are either replicas; well, they are actually pointing to the same place. So, they are not replicas, they are shared. They are not copies. They just shared with each other.

Student: Sir, but page tables are replicas?

Page tables are replicas. So, the kernels space itself is not replicas. The page tables are replicas for those particular entries, not all the entries; just for those particular entries which are mapping the kernel ok alright. So, let us see some nice things that you can do with paging you can do something called. So, firstly, page paging solves the problem of fragmentation and growth in a large way right.

(Refer Slide Time: 43:23)



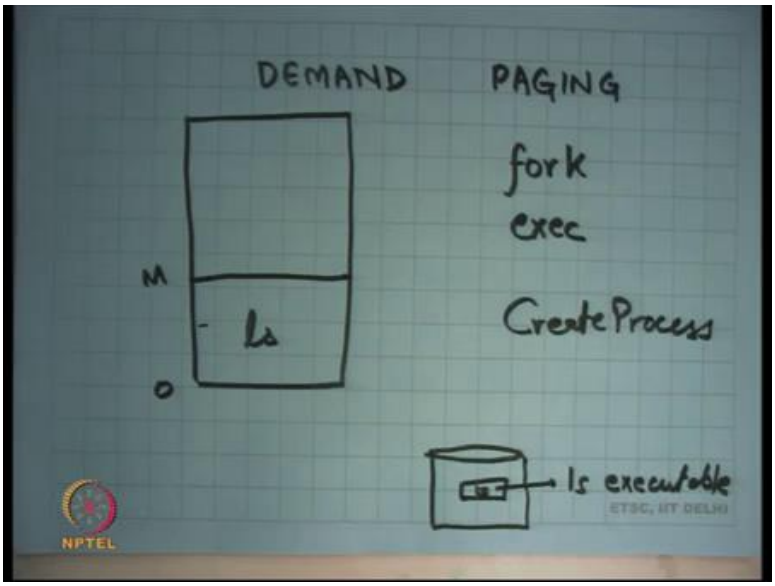
Because now if a process needs to grow all it needs to do is let us say this is the VA space and this is the PA space. So, let us say the process is mapped in 0 to capital M, but you know these are pages which are mapped here. Let us say it has 2 pages; one is mapped here and the other is mapped here.

So, let us say this is P1. So, these are P1's pages and these are P1 pages and now there are other pages like P2's pages P3's pages. So, there is two; let us say two P2 pages, one P3 page and now there is all this empty space. Now, if I want to grow M all I need to do is you know create another mapping for another page, allocate another page here P1 and create this mapping here right.

So, growth is very easy. No problems of fragmentation uh, you just sort of allocate a new page in the in the physical address space and you create a mapping in the page table. So, that the virtual address space grows ok.

So, growth is no problem; fragmentation is no problem because you know the mappings do not need to be contiguous, if there was enough space in the if there all they are enough pages for the new process to fit in, it will get fit in. It is not like they need to be contiguous alright ok. So, it is all fragmentation. The other nice thing it can do is what is called Demand paging right.

(Refer Slide Time: 44:51)



What is demand paging? Let us say I created a new process alright and let us say this processes ls right. So, let us say this is ls and ls maps its code in let us say 0 to capital M hypothetically and let us say there is this disk right which has this program file called ls executable right. ls executable.

Now, when you actually say ls, one way to implement the exit system call in the OS is to actually copy the entire contents of the ls executable into physical memory and create this mapping right that is you know up front you allocate that much memory, you copy the entire discontents to memory, create the mapping and start the process running.

And optimisation over that is that you actually do not read the disk into memory up front, you just create an address space from 0 to M by reading the executable and then, you basically just store some meta information in this address space saying that look right now these pages are not present, but if the user tries to access this page; then, here is where you should get it from on disk. So, here is the disk block from where you should get this particular page from right.

So, for example, if ls was a large program you know let us say the program was let us say 1 megabytes long large, but when typically when you type ls, you are only going to be executing let us say 100 kilobytes of code somewhere and you know almost you know let us it 4 kilobytes of data. So, you only needed those 100 and 4 kilobytes in memory, you did not need to read the entire 1 megabyte into memory.

So, what you do is you know demand paging helps, but the OS does not know you know which what may get executed in future. So, but demand paging helps it creates the address space for the entire 1 megabyte. It maps the first page of the page containing the instruction which is the first instruction into memory and transfers control to that instruction. If that instruction happens to execute some other address which is not currently mapped, what will happen?

Student: Page fault.

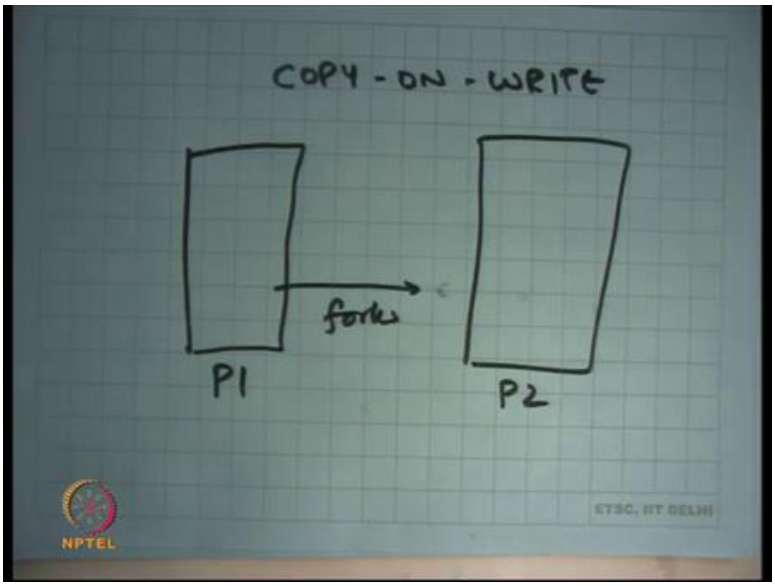
Page fault, the operating system will come into action, it will figure out oh you know actually the process is not doing anything illegal, its I who is playing tricks under the rug right and so, I should not I should actually you know stick to my contract and so, that is when you will basically pick up the page pasted into physical memory create the mapping and restart the instruction alright. So, this is this is demand paging, very useful optimization.

In fact, we discussed this we discussed fork and exec right in Unix and we said you know and then, we said oh windows has this create process right and we said look fork seems very base full because you are going to actually copy the entire process in from disk to memory and all that, but the reason you know Unix designers at that time thought that it is not a big problem. You can actually fork is not inefficient is because of demand paging.

At that time memories were really small. So, most of your program code and data use to live one disk right and so, fork was basically just fork forking a process just involved creating a new page table and you know updating and having the same pointer shared pointers through the disk and the fork was really cheap in that sense. And if the next thing the process going to do is exec you know no problem; you actually did not waste much work right. So, at that time it seems very natural and so, fork and exec was the nice interface given that demand paging was a very possible optimization.

So, I mean optimisations also dictate your programming models right, nice examples and another thing you can do is copy on write right.

(Refer Slide Time: 48:53)



So, what is Copy-On-Write? Let us say here is a process P1 and it forks another process P 2 alright and let us say you know we are it is not the old world, in the new world then processes memories are large; so, the process actually living in memory right. So, once again fork is actually one way to implement fork is that you create a page table for the new process and you basically point the page table entries of both P1 and P2 to the same physical pages except that you mark all those pages read only right. Because if one process writes you do not want that to be visible to the other process, you mark all those pages read only. So, you so, the entire space is shared except that its now become read only.

Now, if one of those process types to write one of those pages, then immediately there will be a page fault and you will just copy it at that time. So, that is called copy on write right. So, you actually share the pages and assuming that you the child process was going to call exec immediately, its actually not going to write to any page right and the page table was going to get overwritten. So, fork is again very cheap even in the memory world alright.

Ok good. So, let us stop here.