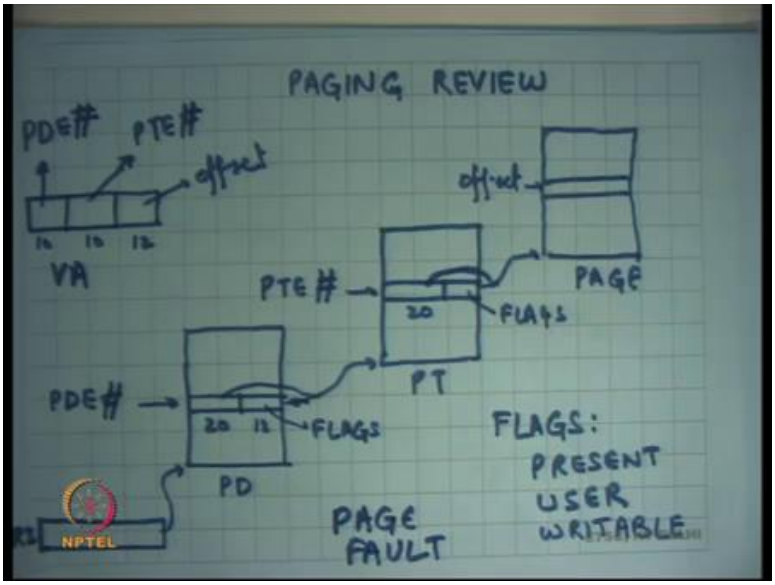
Operating Systems
Prof. Sorav Bansal
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture - 12
Process Address Spaces Using Paging

So, welcome to Operating Systems, lecture 12. So, far we had been discussing paging and we said that the hardware provides the capability of setting up a structure called a page table.

(Refer Slide Time: 00:44)



So, the operating system can set up the structure called the page table which we said that if we are using pages of size 4 kilobytes and an address space of size 2 to the power 32, then you would probably want to have a 2-level page table, so that it has a nice tradeoff between the time it takes to translate a virtual address to physical address and the space that such a structure requires, right.

So, we said a 2-level page table seems like a nice tradeoff between these two things. If you have just one continuous page table then translation time is very small, but at the same time the space over it is very large. If you use a multilevel page table more than two, then the space overheads can become even less, but then the size of then the translation times become unnecessarily large, right.

So, a 2-level page table seems like a reasonable tradeoff. And, so the way it works says there is a register on the chip, on the CPU processor chip called CR 3 which points to a page, a page sized structure, this is this entire thing is one page, but it contains its it is a special page because it contains page directory entry. So, this page is called the page directory.

The first 10 bits of the virtual address, so this is the virtual address; the first 10 bits of the virtual address are used to are called the page directory entry number and based on the PDE number you would see which entry you are going to dereference. From that you are going to get a 20-bit address which is going to point to another physical, which is be another physical address that will point to the page table, right. And then there will be the other 12 bits which are flags.

The page table is going to get dereferenced again in physical memory, so this is a physical address that goes, and you dereference a page table. You get the next 10 bits and that is basically forms a page table entry number. You use a page table entry number to look up the page table to get that entry similarly, and you look at the 10, first 20 bits to get the page number, once again this is a physical address, right.

And once again you have the last 12 bits which have been used as flags. And finally, you use the last 12 bits of the virtual address to index into this page to actually get the data that you were looking for, ok. I said that the PDE number is just a number between 0 and 2 to the power 10, right. So, they are 2 to the power 10 entries in one page directory because you are using a 10 bit number to do that and each entry is 4 bytes, so you know a 4 kilobytes, so the whole page directory fits in one page. Similarly, the whole, the one-page table fits in one page and the page itself is one page, right; 4 kilobytes.

Also, we said that this 20-bit number here is a physical address. Why does it need to be a physical address? Can it be a linear address?

Student: It is finally, (Refer Time: 03:38).

Right. So, it cannot be a linear address because linear address will need to dereference the page table to get converted to a physical address. So, if you use, if this were a linear address then I am basically in an infinite loop, right. So, that will, I will never be able to

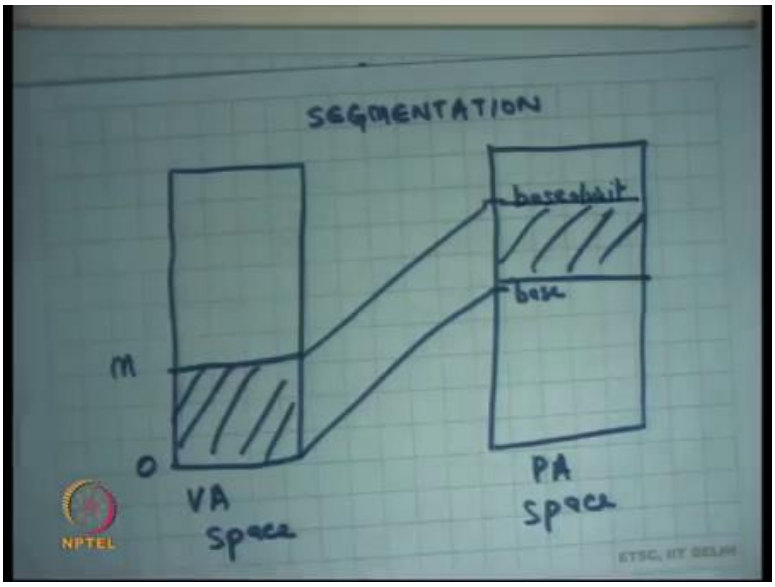
translate. So, this has to be a physical address, this has to be a physical address and so on.

What about CR 3? Should it be a physical address or a virtual or can it be a linear address? It should be a physical address also, right. It cannot be a virtual address. So, all these structures should be physical addresses, ok.

So, what this allows you to do is a much more flexible mapping from virtual address space to physical address space, right. Unlike segmentation where we had a very simple base plus VA computation, paging allows you to have a relatively much more arbitrary mapping except that it the mapping is done at page granularity, it is not done at byte granularity it is done at page granularity, right. So, you can say that this page is here, and that page is there.

But within a page two consecutive bytes should be consecutive in the physical memory also. So, within a page byte should be contiguous, both in contiguous bytes within a page on the virtual memory or in virtual address space are also contiguous in the physical address space within a page, ok.

(Refer Slide Time: 04:57)



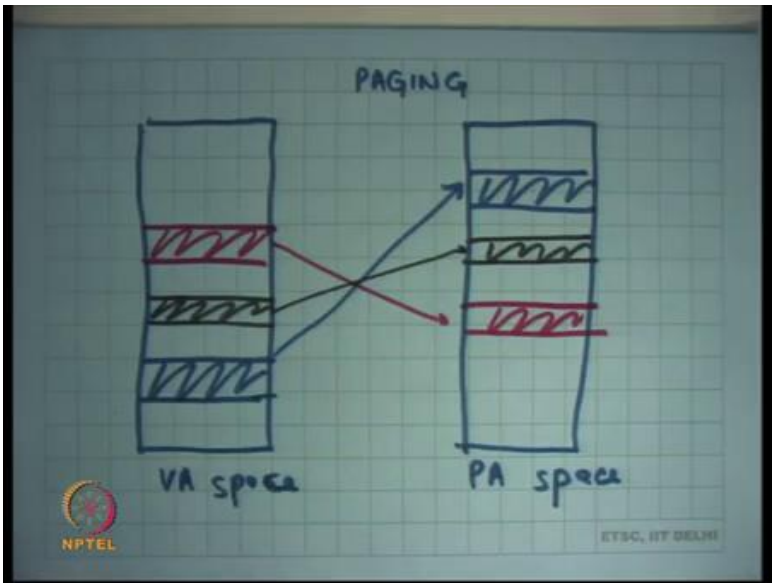
So, just to get a comparison here is you know how segmentation worked. If this was the virtual address space and its starts from 0 to let us say some value M, then segmentation would only be able to map it to some base plus 0 and base plus M, right or base plus

limit. And so, you know this is how the process will basically, that this is the only possible translation and we saw that this has some problems of fragmentation, process growth and things like that.

So, you know it is not very flexible way of doing it. The advantage however of doing things in this way is that the time it takes to actually translate a virtual address to physical address is very small, ok, it is very extremely fast. Just have to do an on-chip edition, right you do not have to look up into any structure nothing, right.

And we also said that the structure, the GDT itself although it lives in memory, it gets cashed on the CPU, right. And, when does it get cashed on the CPU? When you load the segment selector or a register. And so, at that time it gets cashed and later on you know it just used from within the chip, so the translation logic is really fast.

(Refer Slide Time: 06:04)



Student: Sir, in the case of paging sir is the address, this the base address of page directory also cached?

In the case of paging is the base is CR 3 cashed. We are going to talk about what gets cashed or not. But, by the way base directory of the CR 3, I mean CR 3 register is already on the chip, right.

Student: No like the address which CR 3 stores, the base address of the page directory.

That is on chip. CR 3 is a register which stores the base address of the page directory, right. So, it is already on chip. So, it is no problem. So, on paging, on the other hand you can have much more flexible mapping. So, here is an example of a VA space to PA space mapping on in paging.

Here I can you know do say that, these pages are mapped and by the and these pages are mapping to this area and these pages are mapping to this area and these pages are mapping to this area and all the other empty spaces are not mapped. So, all this is possible, right. I do not necessarily have to map from 0 to something, I can just say, or these pages are mapped, and these pages are not mapped.

And you know you can think about how you are going to architect your page directory and page table to be able to do this mapping, right. For example, if you have to do this mapping from here you are going to say, ok, what is this address? Let us say this address is M. So, I am going to say what does M divided by, you know M; what are the top 10 bits of M, right and those are, and based on that you are going to set up the page directory entry of that to point to a page table, right.

And, so now we are going to look at what are the next 10 bits of M and that particular entry in the page table should have a pointer to that page. Where? In this space. All other entries in the page directory and page table can be invalid, right. And how do you say whether a page table entry is invalid, or a page directory entry is invalid? Using these flags. So, if a present bit is not set in an entry, it basically means that this entry does not exist, it is invalid, ok.

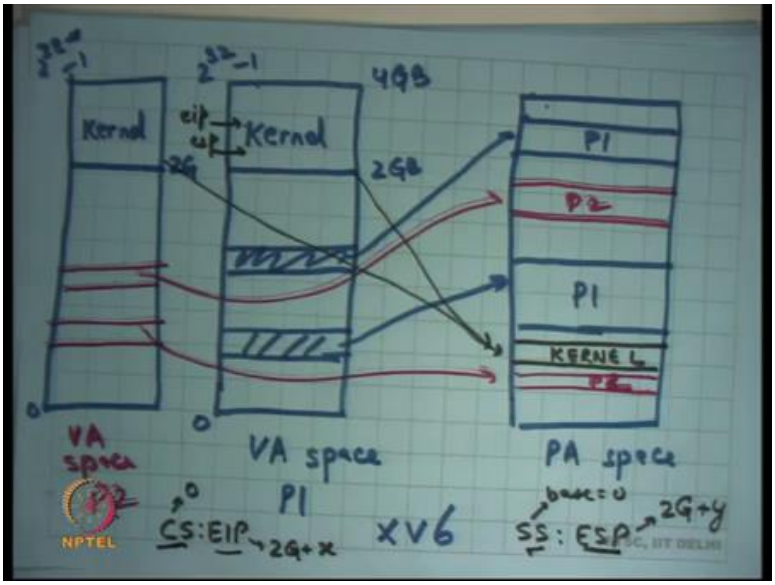
Similarly, there are other flags like user, basically says that this page should be, should is accessible and user mode or not, right. So, only if the bit is set to, this user bit is set in the flags is this page accessible while the processor is executing in a user mode, ring level 3, right. Other pages which do not have this bit set will not be accessible then user mode, they will only be accessible in ring level 0, kernel mode.

Similarly, you can have a bit which says whether this particular page is the page that is getting dereference from this PDE and PTE, page directory entry and page table entry is it writable or not, ok. So, if it is not writable then if some instruction tries to write to that address then you should get a page fault, ok. So, there is something called a page fault which is just an exception in the hardware.

So, an exception gets triggered and the exception handler got gets called, right and we have discussed how an exception handler gets called through the interruptive dissectible, ok. So, that is what we discussed and then we said, let us look at how an operating system uses this page table hardware page and page directory hardware to actually implement processes, right.

So, what does an operating system really need? It needs a few things. It needs to say that my address space should be protected from everybody else is address space number one. Number two, one processes address space should be protected from another processes address space, right. Those are the basic two things that the operating system wants, and we have already seen how it is done in segmentation.

(Refer Slide Time: 09:53)



So, in paging one way to organize things and that is how we are going to focus on this way for the in the beginning, is that let us say this is the VA space, and let us say this is the PA space. So, one way to organize this is that you have per process VA space, so this is VA space for process P 1 and let us say this is let me use a different color. So, let us say this is VA space for process P 2, ok.

In the PA space you could say, here is, let us say this is a process P 1 page and let us say here is another process P 1 page and you can and you could have multiple of such regions, non-contiguous regions. And here is let us say P 2 page, see here is another P 2 page, ok.

And so what will happen is addresses in the P 1 page region it will map to areas in the PA space like this and similarly you know an entire this P 2, they are mapping here and so on, right. Also, we discussed last time that the kernel always stays mapped in the virtual address space and why it so, let us discuss it in a minute, but let us say how does it work.

So, how big is the VA space it starts from 0 and it goes all the way to 2 to the power 32 minus 1, right on a 32 bit machine, but the VA space that is given to a process on something like Linux or let us say windows or XV 6, it is basically that you say a process will not get the entire 2 to the power 32 space let us cut it somewhere, right and let us say that a process has allowed this size of the space. So, this 2 to the power 32 is actually 4 GB, right.

So, one way to do it and let us say let us look at one particular operating system XV 6, right. So, what does XV 6 do? It says you know the process can use 0 to 2 GB and that is it, right. All the addresses above 2 GB in the virtual address space belong to the kernel, ok. So, the kernel maps itself in the address space of the process, except that this particular mapping has the user bit set to 0, which means these pages will only be accessible if the processor is executing at ring level 0, right in privileged mode.

These pages will not be accessible by the user if the processor is executing in unprivileged mode. And it does for every process. So, even in this process is a 2 GB, so exactly at the same place the kernel is mapped, ok.

So, a process has a page, has a virtual address space. Here it can have whatever it likes and those are getting mapped to different regions, potentially different region in the physical address space, but the kernel is mapped at the exact same place in every process. So, we will basically be saying the process and no process is allowed to use more than you know let us say 2 GB, one XV 6.

And of course, this is also blacked, so let us say let me use a different color for the kernel. So, black color and let us say let us say this is these are the kernel pages, ok. So, let us say these. Here is the mapping from the kernel and this stays constant. Irrespective of the process that is running, you will always have this mapping.

And so, what that means is, so how do you implement a different virtual address space for different processes? You have separate page tables for every process, right. So, every process has a separate page table and let us say if you are running process P 1 you will load process P 1's page table into CR 3, if you are executing process P 2 you will load process P 2's page table into CR 3, right.

But what will be common in these page tables is that there will be some entries which are for the kernel and which will always be there in all these page tables, ok. So, that is how you do it. You do not make copies of the kernel, the kernel and the physical memory is just one copy, but there are multiple processes which have this copy mapped in their address space, ok, all right.

So, why do you do this and let us see how it works basically. So, let us say I was executing in the process P 1 and some exception occurs, like a page fault occurs or let us say some external interrupt occurs, like a you know a timer interrupt occurs or a disk interrupt occurs or a network card interrupt occurs. Then what happens is the processor goes through the IDT and replaces the current value of code segment and EIP with the value of the CS and EIP in the IDT, right and so typically these values will be the kernel values.

So, let us have with executing in P 1 and an interrupt occurs, then CS colon EIP that will be loaded from the interrupt descriptor table we will have CS is base as 0, right. So, we are assuming that the segmentation model is completely flat, so all segments are 0 to 2 to the power 32 minus 1. But EIP will be an address which is above 2 GB, right. So, EIP will be some you know 2 G plus x whatever, right.

So, it is going to point somewhere here. And so if an interrupt occurs or if an exception occurs the process straight away, the processor straight away jumps to the kernel address space, but it also reload the CS, so that the last two bits of the CS register are set to 0. So, that you know in the processor is now executing in privilege mode, right. That is the only way it EIP can actually point to something in the kernel space, right.

A user if it just calls jump to this EIP, it will trigger a page fault, right because if the user is executing in ring level 3 which is the last two bits of CS, then it says jump to some address which is above 2 G immediately you know there will be a page fault because the

hardware will try to walk the page table and it will say you know you are running in user mode, but this particular page that you are trying to access is actually not a user page.

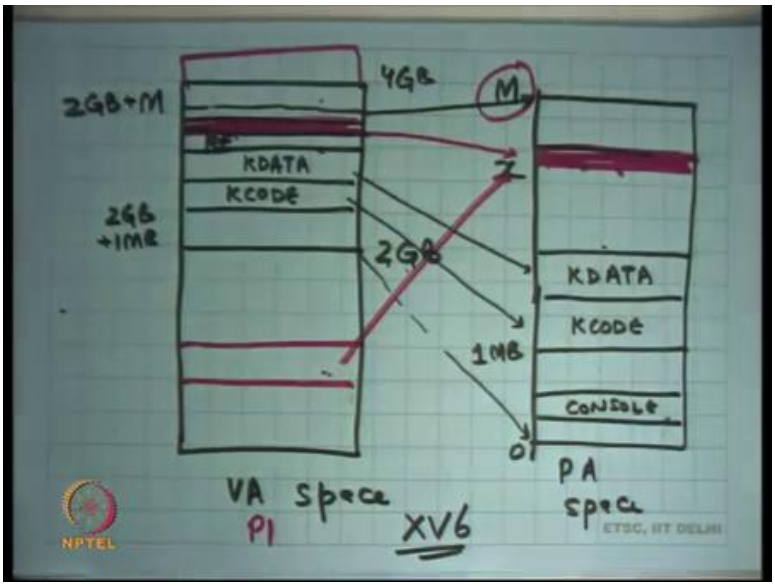
But if you get an interrupt like this then you also reload the CS, so now you are running in privileged mode, so now you can actually dereference a page above 2 G. So, the handler is now going to live in the kernel space. Also, the stack is now going to the kernel stack of the process is now going to live in the kernel space.

So, even in the SS and ESP, the SS's base is still 0 because we are using a splat segmentation model, but ESP values is some 2 G plus you know let us say y, so ESP is also pointing somewhere here, right. So, that is the kernel stack of the process that was currently running. Where do you get the ESP from? The task state segment, right. So, the hardware just reads a task state segment. So, with the responsibility of the operating system to set up the task state segment with the right ESP for that particular process.

If the kernel was a process mode kernel, right, it was the process model then you will have a separate ESP per process in which case you know before you contacts which to the new process you should load a new ESP in the TSS, task state segment. If it is an interrupt model kernel, then the ESP remains constant. This is the same stack that always gets executed, right.

So, in segmentation what was happening was the segment itself, the base itself was getting changed. In paging, the base remains same, the privilege levels changes, yes, but what is actually changing is the offset, which is EIP and ESP, right. And these offsets should be set up by the kernels at that they point to the kernel address space, which is 2 G and above, 2 GB and above, ok.

(Refer Slide Time: 18:53)



So, let us look at this again. So, let us say this is the, this is the virtual address space and let us say this is the this is 2 GB, all right and this is the kernel, right. So, the entire address space is 4 GB here. So, what typically happens is that you know the kernel will have some code and so you know let us say this is kernel code, KCODE.

These, this is this is the memory region which contains all the instructions that the kernel contains, right. And then you have certain region which contains KDATA which let us say contains all the global variables that you are that your kernel defined contains a KDATA and everything else above this is actually free space, right.

So, let us say this is your physical memory, so this is PA space and its going from 0 to let us say some value M capital M, right. And we said you know, the last time we discussed we said you know the addresses from 0 to 1 MB are actually this address space in the PA space is actually cluttered because you have some address range which is meant for the console, right for the VGA console, so if you write there it is actually not memory it is actually going to the VGA output, and the other things like BIOS ROM and all these things.

So, 0 to 1 MB on XV 6 it is a little cluttered. So, let us just leave it as it is, all right, but from 1 MB to M is space that is actually real memory that you can use with memory semantics. And so, what happens is you know let us say in this you would first load KCODE and KDATA. So, this is, and so this is physical memory.

And so, all the other space is basically what is available for other things, right. So, KCODE is mapping here and KDATA is getting mapped here, but all the other space in the physical memory is basically just extra space that you are going to use for other things. What are the other things? You are going to use it as a heap for let us say maintaining your PIB, process, PCBs process control blocks. You are going to use it as a space from where you are going to allocate the kernel stacks of the process, C's, right.

The kernel stacks of the processes are also going to get allocated in this space. Moreover, the address space of the process itself is going to get allocated from this space, right. So that, clearly this is the entire space which is going to get used for allocating address spaces for the process. It is going to be used for allocating the kernel stack for the process and it is going to be used for allocating the other kernel data structures like process control blocks or whatever else it needs, right. So, all that has to get allocated from this space, ok.

So, how, let us look at how XV 6 does this. It basically says, it says 2 GB is basically what the process will get in the VA space, above 2 GB is the kernel. Till 2 GB plus 1 MB, it is not going to touch anything, so it just maps this region from 2 GB to 2 GB plus M, where M is the size of the physical memory to 0 to M in physical address space, right.

So, 2 GB plus let us say this is 2 GB plus M gets mapped to 0 to M. So, the entire physical address space is also mapped in the kernels virtual address space, on XV 6. So, and so the kernel can now access the physical memory directly by just saying, if I if he wants to access physical memory byte number 10 or let us say any x then all I need to do is 2 GB plus x is then the virtual address space is basically physical byte number x, right.

So, let us say the kernel wanted to allocate an address space for the process, so what we will do is it will say let us allocate an address space for the process from here, right which will mean it will allocate an address space for the process from here. So, let us say let us use a different color. So, let us say it allocated an address space for a process from here which means it allocated an address space for the process from here, right and then it sets up a mapping in this area to this area, ok.

So, what is happened? See the, so this is the processes address space, right. Let us say this is P 1. So, P 1 wants to map some area of the physical memory and its virtual

address space. So, it is going to have a mapping in the physical address space like this, but this area is also mapped in the kernel address space like this.

So, here is an example where the same page in the physical memory is actually getting mapped both in the user side of things such that a user can access it, but it is also a mapped in the kernel side of things so that when the kernel actually allocates it or deallocates it has access to all this area, ok.

So, basically just to recap, the kernel maps the entire physical memory in its address space. It allocates areas from this address space and maps it into the user's part of the address space to implement the processes address space or user side address space, ok. Mapping the entire physical memory into the kernels address space has advantages that you can just you know if you want to access a byte number x all you need to do is look at you know your base plus x and you are get the particular address.

Student: Sir.

Yes.

Student: Sir, that address generally 4 GB, so how can you map 4 GB physical space into 2 GB?

So, question is how big is M? So, in this organization that I have discussed so far how big can MB.

Student: (Refer Time: 25:28).

At more, actually it is going to be 2 GB minus 1 MB, right because, actually yes 2 GB because you know 1 MB is also getting mapped here, so it can utmost be 2 GB. So, that basically XV 6 has this limitation that you cannot have more than let us say 2 GB of memory in your system, ok. Actually, I should also mention that the top few address spaces are also reserve for devices.

So, it is actually even less than 2 GB in XV 6. So, what does it; so, question is this? So, for if for something like XV 6 it is good enough number one, because it is just an academic operating system. But for something like a real operating system like Linux this may not be good enough. It may have been good enough in the early days of this

operating system like 90s when you know the memories were not, they were nowhere near 2 GB mark.

So, this was you know this kind of organization actually worked for hardware of that time, but how Linux deals with memories which are greater than you know let us say 2 GB and if you started at 2 GB is basically to recycle these addresses. So, depending on which address you want to access it just changes its address space on the fly, right.

Here I am talking about a static mapping. I am saying the entire memory is mapped in the kernel address space all the time, so it is completely static for the entire duration of the system that is running. But if you are actually running out of the address space because the address space of the kernel is only let us say 2 GB, but your memory is let's say 3 GB then you will say the first few MBs which contain all my handlers and everything those remain always mapped, but other things like the area from where I am allocate process memory for processes that I sort of keep recycling, right.

So, some part of this 2 GB slice will get recycled. So, sometimes its mapping to this area of the physical memory, sometimes it is mapping to that area of the physical memory and the kernel is basically ensuring that its always its doing what it wants to do, but some area of the kernel will always stay mapped because things like handlers, right or kernel stacks, they will always stay mapped.

The memory, the area from where a memory for the handlers is taken which is KCODE should always remain, right you cannot just remove it at any time. Similarly, the memory from at which the kernel stack is living for that for the current process that should always remain and some other things which are basic functionality of the operating system they always remain in the kernel.

Student: Sir, if our physical address space is growing then why cannot we just grow our virtual address space?

If our physical address space is growing why cannot we grow our virtual address space? All right. So, question is if my physical hardware memory is growing then why I cannot just switch to a 64-bit system, that is what; that is what has happened, right. So, one of the big motivations is actually moving to a 64-bit system is so you get rid of these kinds

of limitations, ok. But, let us just focus our intention on 32-bit system because you know that simpler and will help us in understanding many concepts, ok.

So, yes, if you had a 64 bit address space then you can imagine that these are absolutely no problem, non-problems, right because there is no way a physical address space today at least or actually anytime in foreseeable future that you can have actually a memory which is 2 to the power 64 bits long, right that is I think you know I do not know how many items are in the universe, so probably its comparable, ok.

So, this is the organization which let us say XV 6 users and in fact, other operating systems also use. So, let us understand what the advantages are of doing this kind of organization. So, the entire kernel space is mapped in every process and we said that the entire physical memory is mapped in the kernel space.

So, essentially the entire physical memory is mapped in every process in the kernel side of things. And then there is a user side of things which is controlled by the kernel, and so anytime what this gives you the kernel is that if there is an interrupt while the user is executing you do not have to change any page tables or anything you just switch the privilege level and you are executing in kernel mode.

If you want to do some operation on behalf of the user for example, you let us say the user said I want more memory, so all I needs to do is allocate something from its own address space, convert it into its corresponding physical address and create that particular page table mapping, right. So, it can only use, it can use its own malloc function inside the kernel to manage that space the that it has which is actually the entire physical memory, right.

So, it can just use malloc to create a page, let us say it wants to allocate a page for the physical process for the process then it just create mallocs a page, converts, it gets it address, converts that address to the physical counterpart, creates the corresponding mapping in the page table at the, right position and that is it, returns back to the user, right.

So, any interrupt or exception or system call does not need any change in the page table. So, it is very fast in that sense, because the kernel is mapped entirely in the address space of the process although in privilege mode it sorts of improve things in that sense, ok. The

other thing is let us say you have a system call and this system call will have some arguments, right.

So, things like exec and exec is a system called which takes the first argument as string. The question is, how does a process give an argument to a system call. How does a process; how does the argument of a function work?

Student: (Refer Time: 31:16).

Right. So, if exec what the function call, the pointer to the string which holds the name of the executable will be pushed on the stack, right and the stack address will be visible to the function, right. So, the function can just look at the stack address and it can dereference at address and that the dereference will also be visible to the function because that address to which it points will also be mapped in the user address space, right.

So, it can dereference the argument and it will still get a value which is in the user address space which you can access, right. Similarly, in the case of a system call the user can use a similar organization, it can store the string in its own address space, for example, its slash bin slash ls and then make a system call called exec. The control moves to the kernel, but if the kernel wants to read the arguments you are still executing in the address space of the process.

So, it can just dereference of argument and because the page which contains that strings slash bin slash ls is still mapped dereferencing should still work, right. The kernel would be executing in privileged mode, but privileged mode execution can access unprivileged pages. So, it can just, so accessing the arguments of the user is very easy. You just have to dereference the pointer and you can just get the arguments, ok.

Once again, there is an advantage of mapping the kernel into the user address space because if the kernel is doing something on behalf of the user and the user wants to give some pass some information to the kernel it is very easy to pass information from the user space to the kernel space because they are both living in the same address space. The user can just set up something in its own address space and give a pointer to the kernel, right. The kernel can just dereference a pointer and it will be able to read the value because you are in the same address space.

Compare this with segmentation, right. In segmentation when there was a system call then you were actually changing the base. So, now, if the user wanted to give an argument to the kernel, I cannot, the kernel cannot just dereference a pointer because you are in a new address space, the base is different, right. So, the pointer that the user gives you, if you just dereference it now you are actually going to be looking at something else, maybe something wrong, right.

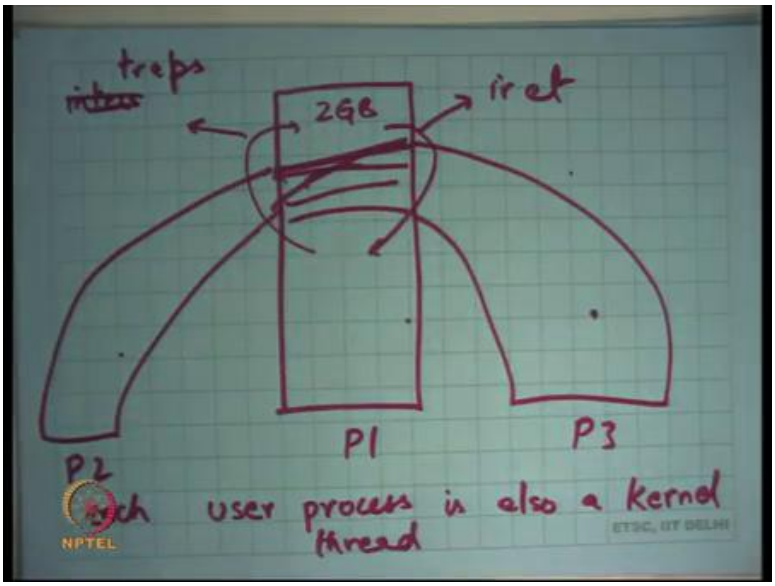
So, in the case of segmentation what is actually needed is that this the user actually, user needs to you know package all its arguments and copy it into the kernel space somehow, right because the kernels address space and the user address space are separate and so you need to copy it into the kernel using some kind of a mechanism. And in this case, you can just simply dereference the argument, right. So, that makes things faster.

Student: Sir, in segmentation could not the user end pass its own CS for inter as well for giving arguments, if it gives both the?

That is an interesting question. So, could not in segmentation could not the user have you know could not the kernel have changed through. So, the kernel already knows what the users' segments are, right. So, the question is cannot you, in the in segmentation cannot the kernel just switch to the user segment before dereferencing the pointer, right that is your question.

So, but that requires you know switching back and forth, and it will basically the, so that is this that is how it will typically be done, but that is requires switching segment, segments and then copying. So, typically what you will do is you will switch to that segment, copy it to your own segment and then switch back and then start operating on it, ok. So, there is more overhead in that sense. Question, ok.

(Refer Slide Time: 35:09)



Let us review this once again. Here is another way of drawing this, right. Let us say, so this is process P 1, this is process P 2, and this is process P 3. Each of them has a separate address space in the lower 2 GB, but they have the other same address space in the upper 2 GB, right. And what that means is if process P 1 makes any change in the kernel side then process P 2 will see it immediately because they are sharing the address space at the kernel side, right.

So, another way to say this is that the user processes, each user process is also a kernel thread, right. Recall, what is a thread? We said a thread is basically you know, threads are basically control flows which share an address space, right. So, these are controlled flows that are sharing an address space, in this case the address space of the kernel, right.

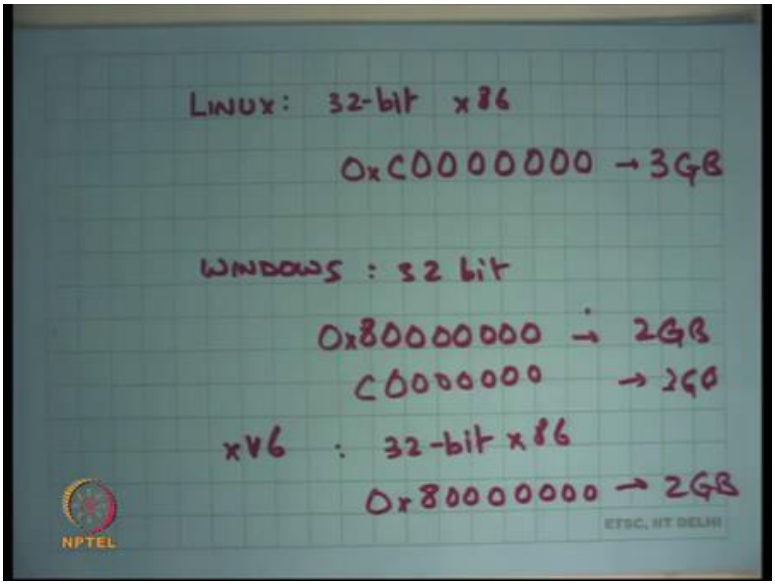
So, even though there are processes at the user level because they have different address spaces at the kernel level in this organization, they are threads, right. So, each process has a kernel half and a user half, and it switches between the kernel half. So, how does it switch from the user half to the kernel half?

Student: Through interrupts.

Through interrupts or exceptions or let us call them traps, ok, that includes system calls. And how does it switch from kernel half to user half? Let us say using the iret instruction, right. And so, but whatever change one has made the other one will see

immediately, ok. So, in that sense it is a thread. They do not have separate address spaces, ok.

(Refer Slide Time: 37:24)

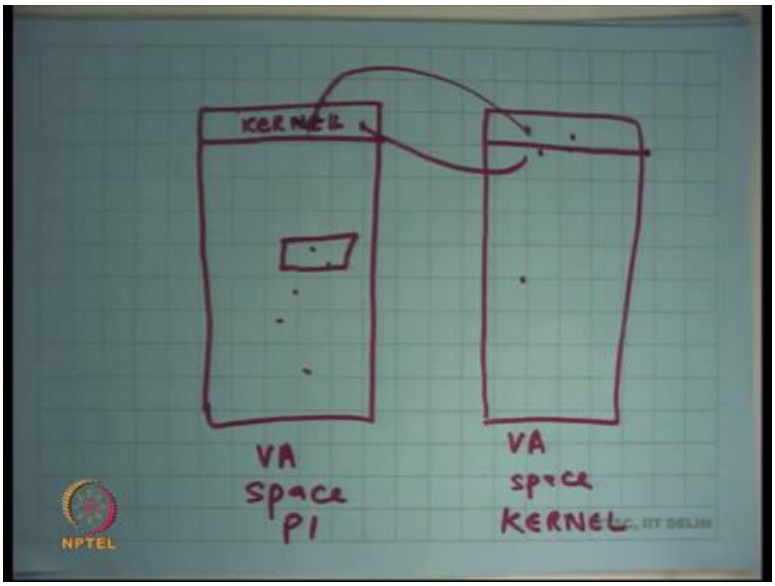


Just some facts. Linux on 32-bit x 86 starts the kernel address space at 1 2 3 4 5 6, at this address, which is 3 GB, all right, ok. Windows on 32 bit, I think by default starts at this address that is 2 GB, but you can configure it to say that you know please start at 3 GB, all right XV 6 on 32 bit x 86 starts at 2 GB, ok.

So, all these operating systems are actually mapping the entire kernel into the address space of the process and basically because it improves, it allows very fast switching between user and kernel number one you do not have to change any address spaces. Number two, it allows very fast communication from user to kernel, for example, arguments and maybe even return values, right.

So, very fast communication bit back and forth between user and kernel because at the same address space. All you need to do with dereference a pointer and you are there. Let us for completeness also understand what could have been the other organization, right. Let us say if the entire kernel was not mapped into the process address space what is the other possibility, right.

(Refer Slide Time: 39:15)



The other possibility is let us say this is the virtual address space of P 1. It just maps a small area for the kernel, right. And let us say there is another virtual address space which is just the kernel's address space not associated with any process.

And so what will happen is if you are executing in process P 1, the handlers and all that are still in this space, so if there is an interrupt you will got jump here, but the first thing the handle I will do is switch page table and you are here, right. And if you want to go back to the user, you are going to switch it back and then you are going to go back to the user using iret, right. But notice that this involves number one, switching page tables on a kernel execution, number one.

Number two, if I want to communicate from here to here it is not easy because what I need to do is copy my string from here. Let us say I wanted to communicate a string. So, I need to copy the string from here to here, then he needs to copy it from here to here, right, so the way he will copy it from here to here is let us say this same area is also mapped here, right. And so, if he copies it from here to here it is immediately visible here and so when it switches it, it now you can see it, right. It involves extra copying basically from user to kernel. So, it is a relatively slower.

Student: Can we say it is same as segmentation?

Can we say it is same as segmentation? Yes, roughly same except that in segmentation you did not even, I mean segmentation is giving you a mechanism, it does not even need you where you do not even need to map this area into the user address space because it is a completely different segment, all right. Question?

Student: Sir, sir in the kernel region all the addresses are linearly mapped to PA space, physical address space.

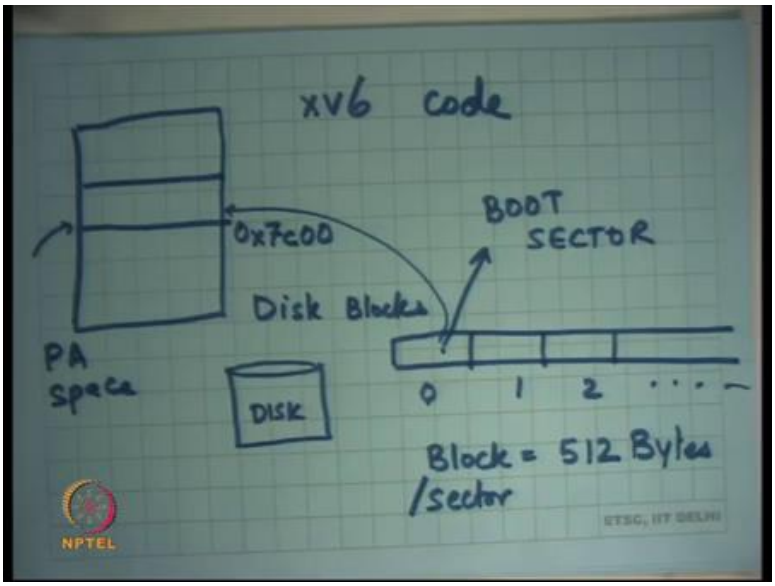
Yes.

Student: Sir, so we do not even need another this table it another table, we can directly just know that this is training in a kernel mode, so we can directly?

So, question is you know if the kernel is just going to have a one on, one to one mapping from virtual address space to a physical address space why do I need another VA space? Well, just that you know once you enable paging on hardware you cannot just access physical address directly you have to go through a page table.

So, once you enable the hardware you know every address will go through the page table. So, if you want to access the physical address you just set a one to one mapping from virtual address space to physical address space, all right. So, that is you know that is my, that is the introduction to paging. And from now onwards we are actually going to look at XV 6 code and look at XV 6 code to understand exactly how it is doing all these things that we have so far discussed, right.

(Refer Slide Time: 42:14)



So, let us just understand, so we are going to start with how a compute boot right. So, how does it appear to boot? Across power recycles you know if you switch on, switch off the power and you switch it on everything in your memory it is just everything gets wiped out, no processes, no kernel space, no user space nothing possessed. The only thing that possessed is a disk, right.

And so, when you boot on the computer there are certain conventions on how the disk should be laid out for the computer to actually start, right. So, let us just; so, a disk is actually a cylindrical device, but let us just consider it as a linear address space of blocks, right. So, this is block number 0. So, disk blocks. So, block number 0 and block number 1, 2, right and let us say it is very large, ok.

So, that is how the disk is. It is a collection of blocks where each block and once again this is the property of the hardware is 512 bytes, right or sector, also called sector, all right. So, each block each sector or a block is 512 bytes and what happen. So, once again the architecture defines exactly how it is going to interpret the contents of the disk.

So, the first block of the disk which is block number 0 is called the boot sector, right. So, and so when you start up your computer, the first thing let us say this is physical address space, right. So, the first thing the computer will do before even the first instruction of your operating system executes is that it is going to load the sector number 0 into some

predefined address, ok. It is going to load it here and it is going to transfer control to the first instruction in that sector, right.

On x 86 this addresses hexadecimal 7c000, ok. So, you load this particular sector of 512 bytes, at this address at this physical address 7c00 and you transfer control to the first instruction. Recall, from a previous discussion that when the computer boots on x86 it boots in 16-bit mode, right.

So, in 16-bit mode there is no paging, segmentation is you know of a very primitive form where you just multiply the base with the segment value with 4 with 16 and added to the virtual address. So, there is no GDT or anything; that is what. So, at this point the processer will be executing in 16-bit mode.

The boot sector should be what should the boot sector do? So, the hardware will only load the boot sector from here to here. The boot sector should know where the rest of the kernel is and now it should load into the right memory and transfer control to it. So, in those 512 bytes the kernel developer needs to write code which loads the kernel into some other space and then jumps to it, right.

Fortunately, 512 bytes are enough to do this small operation, right. Because what you are going to look at next time is basically the code to do this and then what does the kernel do next and next and next, right and so how does it (Refer Time: 45:54) paging and then how does its starts the first process, ok.

So, let us stop here.