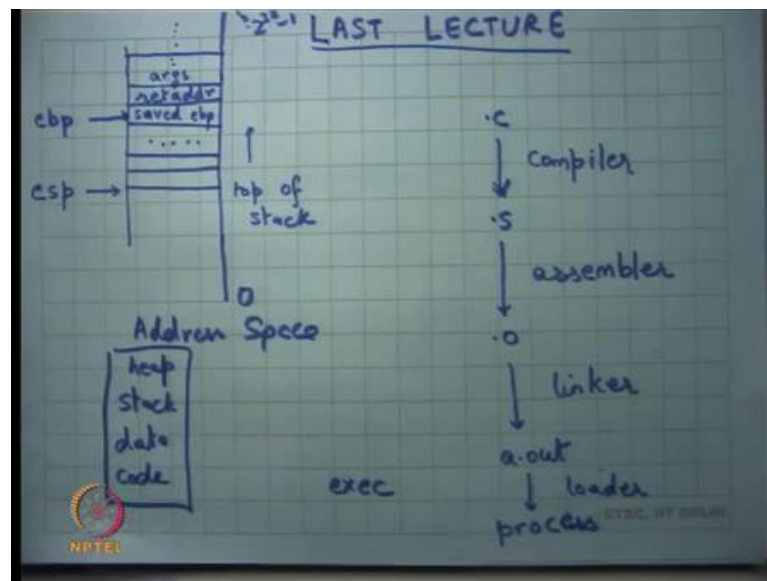


**Operating Systems**  
**Prof. Sorav Bansal**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Delhi**

**Lecture – 06**  
**Physical Memory Map, I/O, Segmentation**

So, welcome to Operating Systems, lecture 6.

(Refer Slide Time: 00:29)



In the last lecture, we were looking at how function calls are implemented and in particular we were looking at the stack, right. And we said look, so this rectangle that I have drawn here is really the address space of a process let us say and the stack is defined by this register called ESP, right.

So, this register ESP which means the stack pointer points to the top of the stack, right. This address space can be thought of as growing as you know going top to bottom which means let us say this is the lowest address 0 and this is let us say the highest address. And the highest address you know 32-bit machine will be.

Student: (Refer Time: 01:10).

You know  $2^{32} - 1$ , right. So, this is the address space and somewhere in the address space there is this pointer called ESP which points to the stack pointer. By

convention, the stack grows downwards. So, this stack the point where the stack pointer point is also called the top of the stack, right.

It is a stack in the sense that anything which is below the stack is junk. So, it can be overwritten. So, safely the program can start pushing data to the stack and does not have to worry about what is below the stack, right that is what a stack really means. But everything above the stack is useful and should be handled carefully.

It is not, it is not just your stack in the strictest sense because you can actually do random access to the middle of the stack, right as we saw last time. So, example if I just want to access this argument all I need to do is put an offset to ESP and I will be able to reach this argument, right.

Similarly, I can put an offset to EBP, and I can still reach this argument, right. So, you know it is a choice of the compiler. Typically, he will also maintain another pointer called the EBP which is also called the frame pointer and which points to which is equal to the value of the stack pointer at the entry of the function, right.

And at that point we as a convention compilers like gcc also save the previous EBP that is the saved EBP and so EBP points to the frame pointer and the first value there will be the saved EBP, the plus 4 offset value will be return address and so on, and then there will be the arguments. So, you can you know access these things using offsetting EBP.

And now all this can be my functions local bodies data for example, the local variables that I allocate on stack, all right. Above these arguments there will be more local variables of my caller, right.

And we also said look I mean this EBP pointing to saved EBP allows us to do back trace because we all I need to do is look at the EBP, from there I will get the saved EBP, so that will give me the frame pointer of my caller and so on and from each of these I can also get the return address, so I can get the whole call chain till for example, I reach the topmost function let us say main or something, all right, ok.

Then we said you know let us look at the workflow or the tool chain that allows us to build a program. So, we let us say we write our programs in high level language like c.

There is a compiler which compiler set into a dot s file dot. What is the dot s file? It is an assembly file which has all the instructions in assembly format or in assembly syntax.

Assembly syntax is just a human readable representation of machine instructions, right. So, this is a human readable representation of machine instructions. And assembler converts human readable representation to a machine-readable representation, so it converts it into actual binary format which a machine can read when it executes.

Then the linker can link multiple dot o files to make one a.out file. For example, you can call printf in your function, in your file, in your dot c file, but printf does not need necessarily need to be defined by you, right. So, printf could have been written by somebody else, but if you link all these things automatically things get patched so that the right printf gets called.

The a.out file then gets loaded. So, there is a loader which loads the a.out file. So, a.out file has to be in a certain format for it to get loaded and then it becomes a process, it becomes the running process, right. So, we said at the load is typically implemented in the operating system.

For example, when you make the exact system call, you know the operating systems loader comes into action it loads it looks at it reads the parts the a.out file, paste it into the processes address space, right. So, let us say this is the address space of the process, address space. It looks at the a.out file and the loader is going to paste the contents of the a.out file into the address space.

For example, that will paste the code into the address space it will post all the initialized values of the global variables into the address space, right and it will initialize a stack, right. So, it will allocate some space for the stack and add it will initialize the value of ESP, all right and so a process gets created.

And now initializes the value of EIP with the first instruction that needs to get executed which is also, this information is also encoded in the executable a.out and so now, the process starts running, right. So, from a.out it got the code and it got let us say the global, global data, the global variables of your program and all the local variables will now get allocated on stack let us say.

And then it also initializes the heap, right. So, you have heard of the stack and the heap before? In our courses, ok. So, let us, so the address space is let us say divided into you know code, data, stack, and heap, all right. I mean the layout need not be necessarily in this order, but roughly speaking this is the these are 4 different parts of the address space.

The code is basically the code that gets to run and as we said even code lives in memory or code lives in the address space of a process and code is obtained from the executable a dot out. Similarly, data refers to the global data of the executable and once again data is again obtained from initialized from a dot out, right.

The stack is initialized to some empty space, right and so some empty space is allocated in the address space and the stack pointer is initialized to point into the stack, so that now when the program runs it can actually start pushing and popping from the stack, right.

And then there is this thing called heap. Heap is basically all the other space that needs that the program may need to allocate. So, for example, if you want to allocate a data structure like let us say a linked list or a binary search tree or something then you would use you know neither stack nor global data are use or write the, right places to do it because stack grows only in a certain way data is fixed size you want variable sized dynamic on demand creation of memory that is what is called heap, right.

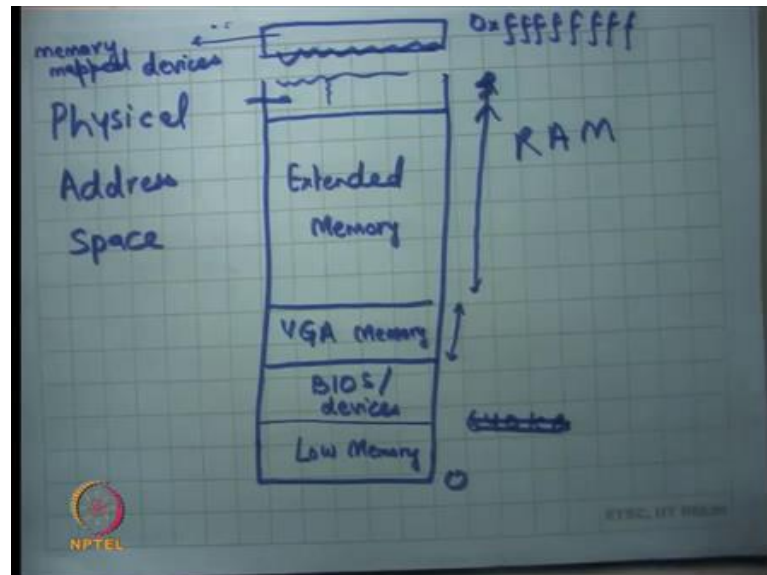
So, heap is basically an area in the address space from where you can with a program can request memory using calls like malloc or free memory using calls like free, right. So, that is what operates on the heap, ok.

So, the code and the data are initialized by a dot out. Stack and the heap are just initialized to you know empty address spaces with appropriate pointer pointing to the right things. And we are going to look at how malloc etcetera get implemented later on but let us just look at the stack for now and understand that, ok.

So, that is the address space of a process and you know I have not yet talked about how the address how a process, how the address space actually gets implemented. We have so far been assuming that every process has a private address space and we have talked about how the address space gets initialized by the loader and how it gets used etcetera and, but how does the address space get created that is what we want to talk about in the next couple of lectures.

But before we do that let us look at the physical address space that is available to the OS, all right. So, what is available to the OS?

(Refer Slide Time: 09:03)



So, let us draw the address space which is which I will call the physical address space. It starts at 0, all right and potentially goes all the way up to  $2^{32} - 1$ , all right. And let us look at you know, so let us look at a real world setting and let us see you know how does the physical address space look to the OS when the OS boots.

So, when the OS boots, he sees some memory and he can use addresses to access that memory and that you know that is what we are looking at. So, what are what address is referred to what parts of the memory. So, you know there is something called low memory right, till 640 KB then there are bios slash devices, right. Not necessarily in this order, but let us forget about exact numbers, but let us just say you know what all exists. So, there are bios and devices.

Then there is you know VGA memory. So, VGA memory is special memory. If the OS writes to that memory it actually appears as you know it actually goes to your display, right. So, let us say there is (Refer Time: 10:52), you know there is some area of memory which is basically called VGA memory and whatever the OS writes to this area of memory there actually goes to this place.

So, it is actually you know mapped into the display it is not real memory. And then there is what is called extended memory and the size of this is basically dictated by the size of your RAM, right. So, when you say I bought, I buy a you know I bought a 1 GB machine, so the size of the extended memory is basically 1 GB, right.

And then at that, so let us say the extended memory is to this point then everything above it the address space is unused which means that if the OS says I want this address it is going to get an error, right. The hardware is going to generate some kind of an error signal.

It is going to say error interrupt and that is what you commonly see as a bus error, you know if you have ever seen the bus error in your programs; the bus error basically means that an address was generated which does not have a mapping in the physical address space, right, ok.

And then you know somewhere on the top there what is called memory map devices. What these are basically, these are addresses that are backed not by physical RAM, but by devices, you know even think of some device and the idea is that when you write to this device to this memory address it actually goes to a device, right, it goes as a let us say command to the device and you read from that memory address it actually becomes a read from the device, right.

So, the device will for example, have certain registers. So, let us take for example, the printer, right. So, let us say the printer has certain registers which say you know start a command and stop command etcetera and these registers are actually mapped in the physical address space.

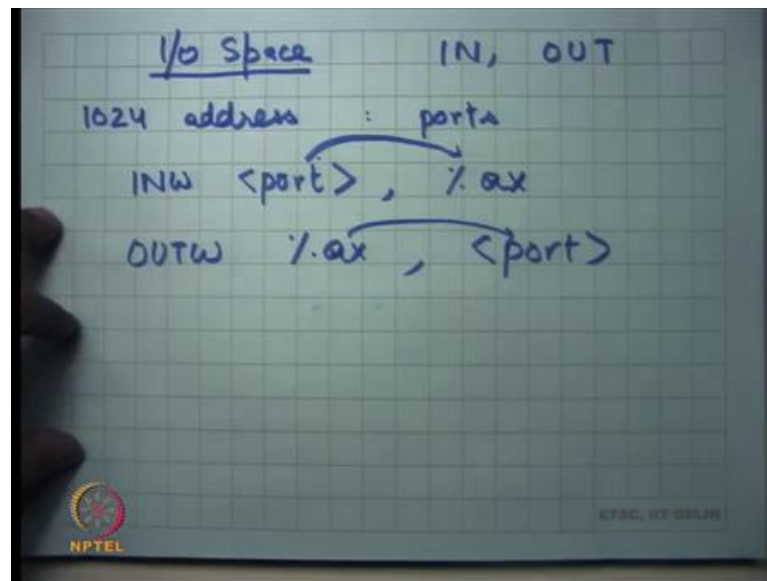
So, you know I am just taking examples, but the exact mappings will be determined by some kind of a manual which the printer manufacturer gives you, but basically the way it will work is that the registers are mapped in the physical address space and the OS is can read or write to this a physical address space to send or receive commands and data from to and from the device, right, ok.

So, this is part of the physical address space, but it does not necessarily behave like memory, right. What are the semantics of memory? That if I write something to address

x and then I later read something from address x with no intermediate write to the same address then I am going to get the same value.

But this address space will not necessarily follow those semantics, right. So, in other words writes and reads to these addresses may have side effects, ok. So, they do not necessarily behave like memory, but they share the physical address space, ok. So, that is one way of accessing devices.

(Refer Slide Time: 14:07)



Another way of accessing devices are what is called using IO space. So, x86 also has special addresses which is, special a separate space for IO which is called the IO space, which is accessed using special instructions like IN and OUT. So, last time I said that there are special instructions for IO called IN and OUT. So, let us see how they work in action, right.

So, the way it works is the IO space is actually made of 1024 addresses. So, it is a pretty small space actually compared to the 32-bit space that we had for memory, right. So, it is only 10-bit space for the IO. And so, and these addresses are also called ports, IO ports, right.

And I can say I can use commands like in from a port, right to let us say register, so let us say the register is ax. So, let us say it is I want to read a 16-bit value from this port to register ax this is the instruction to do that, ok. So, once again as we saw in the memory

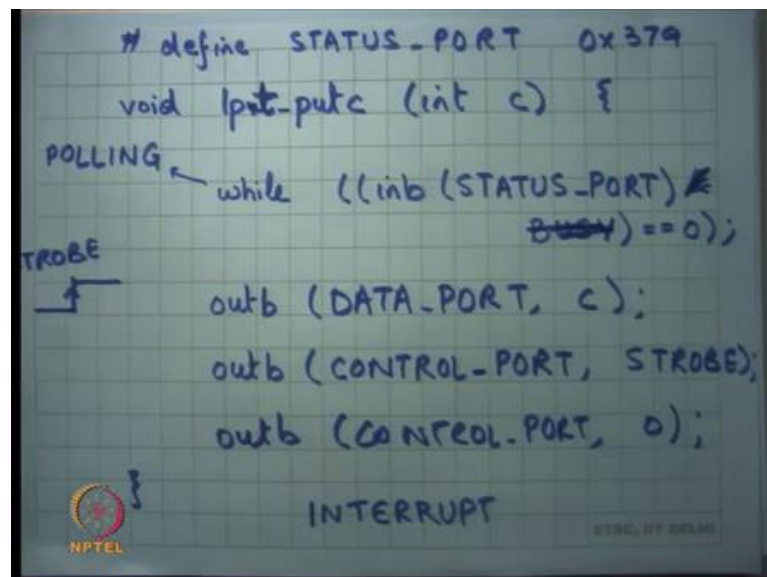
mapped case there were certain registers of the devices that are mapped to certain addresses of the physical memory.

Here certain registers of the devices are mapped to certain addresses or the IO space, right. The difference being that in that case I could just do read and write to that location or in fact, I could execute any instruction on that memory address like increment, decrement, subtract, add, move etcetera. In this space I can only use these special instructions called IN and OUT, right.

So, IN basically says read a value, read a byte or read in this case read let us say a suffix with W, in this case it says read two bytes from this port to this register ax, right or I could say OUTW percentage ax some port which basically says write these two bytes from ax to this port, ok. So, this is there is a way to access devices.

And let us look at a real example of how something like this works. So, we have seen two ways of accessing devices one is using the physical address space of memory and the second is using the IO space which we call also called ports, all right. So, let us look at let us say the printer.

(Refer Slide Time: 16:37)



```
#define STATUS_PORT 0x379
void lpt-putc (int c) {
    POLLING ← while ((inb (STATUS_PORT) & 0x04) != 0);
    STROBE ↑ outb (DATA_PORT, c);
    outb (CONTROL_PORT, STROBE);
    outb (CONTROL_PORT, 0);
    INTERRUPT
}
```

The image shows a handwritten C code snippet on a grid background. The code defines a macro for STATUS\_PORT as 0x379 and a function lpt-putc that takes an integer c. Inside the function, there is a while loop labeled 'POLLING' with an arrow pointing to it, which checks if the 4th bit of the status port is set. After the loop, there is a line 'STROBE ↑' with an arrow pointing to the first argument of the first outb call. The code then writes c to the data port, sets the strobe control bit, and finally resets it to 0. The word 'INTERRUPT' is written at the bottom of the function block. An NPTEL logo is visible in the bottom left corner.

So, you know I will give you some example code or some pseudo code of a printer driver which basically wants to put a character putc to a printer lpt, right line printer let us say. So, I want to I character to the line printer and let us say I want to implement a function



in C which puts this character `c` to the printer, all right. And let us see how something like this could be implemented.

So, I could say `while (inb (STATUS_PORT))`, all right. What am I doing here? `inb` is another function which internally is just executing this instruction called `inb`, right on this port called status port. So, let us say you know status port has been defined here some value let us say, right. It is some number less than 1024 and the printer manuals say that this is the port on which I will basically be accessible to you, right.

And so, now the software can say make can execute the instruction `inb` on this particular port and get the data on that byte in a particular register and then you know it can execute its regular code to check the value. So, in this case I am actually you know I am writing C code, but you can you know imagine converting this to assembly, where you just read you execute the `in` instruction on that port, you get the; you get the byte and then you compare the byte with this particular value busy, right.

Let us say busy is also some number and bit and you come and you AND it, and till its busy you keep spinning, right that is what it is saying. So, it is saying till the status says I am busy keep spinning. So, it keeps spinning till the status is busy, right.

As soon as the status becomes not busy it is going to break out of the loop, all right, ok. Then at this point I know that the printer is not busy, and I can let us say send character `c` to the `DATA_PORT`, right. I am writing these as functions, but these functions are basically just single instruction functions that just execute the `outb` instruction, ok, all right.

And then I put a data value on the data port and then I may need to tell the printer that look now sample the data, right. So, I put some value on the data port and now I need to let us say tell the printer that you know please pick up this data I put the data in your register, now pick up the data and let us say the printer was edge triggered which means you know there is some strobe signal in the printer which picks up the data on every edge or every transition of the strobe signal.

So, let us say I do that by saying by putting, so there is a `CONTROL_PORT` let us say. So, let us say the value of `STROBE` was 1. So, I put 1 to the control port and then I put 0

to the control port. What this does is it simulates an edge on the control port and that is when the printer is supposed to pick up the data from the data port, right.

So, an example of how let us say a driver could be implemented, right. So, first you check the status port waited it for it to become free, then you wrote some data to the data port register and then you strobe the control port for the printer to pick up that data.

And now the printer may say I am going to be busy for the next few milliseconds or whatever, let us say its printing that data, in which case if there is an extra character that needs to get printed it is going to be now the CPU is supposed to wait till the printer is actually ready to take data, take the next byte of data, ok.

Now, clearly let us say there is just one printer and you know, so the operating system should be the only one that is that is actually executing code like this, right. It cannot be like a process cannot be executing code like this, right, because you can imagine if multiple processes try to do the same thing then there will be chaos, right.

Because both of them are going to see the printer is free and both of them are, I am going to try to write to it and so the printer state machine is going to get confused. So, there is going to be there has to be, there has to be some organization such that the printer state machine gets followed properly and one way to do that is to use the OS as an intermediary.

So, the process is not going to access the printer directly, process is going to use system calls like read and write and the OS is going to convert the system calls into code like this, right. Also notice that this kind of code involves this while loop. While the CPU is executing this while loop the CPU is essentially busy, right. I am basically just checking the status port to see if the status if the printers got free.

A more efficient use of my resource could have been that I use the cp; if you know the status port is its busy I figure that out and I start executing something else and there is some way for the printer to let me know that the status port is now free and, so that is when I start executing the rest of the logic, right. That would have been the more efficient way of doing things, right, in some situations.

So, this is called polling. Polling means I keep checking, perhaps at periodic intervals whether the condition I am looking for has been, is because has become true or not, right. As opposed to the other approach which I just described which is called interrupts way approach in which case I tell the printer look interrupt me when you become free, right, and so and now I start going doing something else. And now when the interrupt becomes when the printer actually becomes free it interrupts me and I check again whether it is actually free and then I execute the same logic.

Student: Sir, in this called busy variable like, busy is the variable or a constant?

Busy is a constant.

Student: Sir, so if we have just (Refer Time: 24:31) and (Refer Time: 24:32) so, like why are we doing bitwise AND (Refer Time: 24:37).

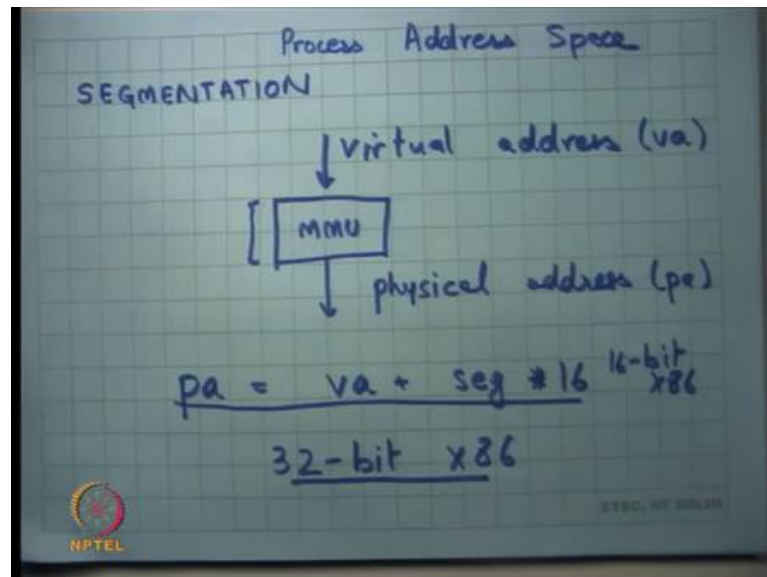
We are just checking if a certain bit in this value status port, in the value returned by status port has been set or not, right. You know for simplicity you could even ignore this and say you know let us say the specification says that the status port is going to be either fully 0 or fully 1 depending on whether it is busy or not, right. I was actually just; so, busy is just a way of saying that I am checking 1 bit as opposed to the entire byte, ok, all right.

So, polling versus interrupt, these are always two options in system design, right. So, one entity wants to access another entity, whether I should keep checking whether that entity is ready to receive my message or whether I should use an interrupt based mechanism and it is not necessarily that one is better than the other. Interrupt comes with its own cost, right.

Because if I set up an interrupt on the printer, the printer is going to in future invoke an interrupt handler and there will be some cost associated with executing that interrupt handler. For example, if I expected that the status port will become free very soon, you know let us say in the next 20 to 100 iterations I am very likely to find in the status port to become free and actually cheaper to do polling, all right. On the other hand, if I know that the status port is going to be free after a long time let us say 100,000 cycles or million cycles then it makes sense to do interrupts, right.

So, it is really an choice that an OS has to make depending on the device characteristics and the workload characteristics whether to use one or the other. And we are going to see more examples of this design choice later when we talk about real examples, ok all right. So, now, let us talk about process address spaces.

(Refer Slide Time: 26:45)



So, far we have looked at physical address space and IO address space, right. Now, let us look at how an OS implements process address space. So, once again what are our requirements for a process address space? Firstly, it should be private which means nobody else should be ever able to access it, right. Second, it should be protected which means a process should never be able to step out of its address space.

So, it should not be able to access anything outside the address space, right. So, let us just work with those two sorts of requirements. And, one way to implement address spaces is what is called segmentation, all right. So, how does; what a segmentation? So, we saw when we were discussing the x86 architecture 16-bit, we said there are these special registers called segment registers CS, DS, SS, ES, right.

So, they were basically being used to add. So, there is an MMU. Here is an address that the program gives you and here is the address that actually goes out on the wire to the memory, right. So, let us get say that the address that the program gives you is a virtual address and the probe and the value that actually goes out on the wire physically is called the physical address.

In the 8086 machine that we saw this MMU was doing a simple addition. So, you know let us say this is  $va$  and this is  $pa$  then on the 8086  $pa$  was just equal to  $va + segment * 16$ , right and that is really not enough for you to be able to provide any kind of protection or anything of that sort.

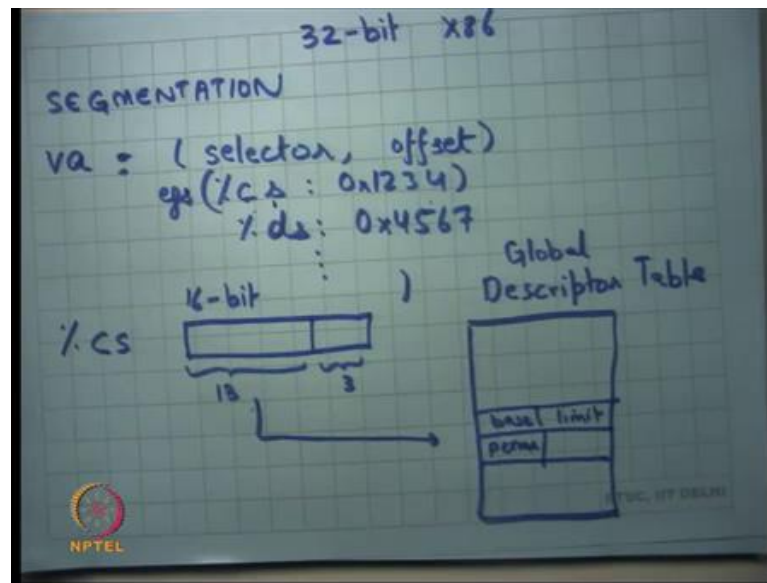
It was just a matter; it was just a matter of convenience that you know you can access more memory with less number of virtual addresses. But now we want the MMU to be smarter in the sense that the virtual address of a process should get translated to a physical address such that it only accesses a memory that it is allowed to access, it never steps over other person's memory and so on, right.

So, let us look at how such an MMU is implemented. And now and what I am going to look at is segmentation on 32-bit x86. So, this was 16-bit x86. This is 8086 which we saw. And now let us look at how processes are implemented or how segmentation works on 32-bit x86, right.

When 16-bits machine processors like 8086 were designed at that time multi-processing or the ability to have multiple processes which do not trust each other was not that important because you know workloads had not evolved to that stage at that time.

So, that time they just use the simple segmentation hardware to be able to do MMU, but by the time 32-bit and very soon you know multi-processing was needed and so they needed more kind more hardware in the MMU to be able to do that, all right, ok.

(Refer Slide Time: 30:27)



So, let us look at 32-bit x86 segmentation. So, once again a virtual address is a pair of segment selector and an offset, right for example, CS: 1234, right or DS 4567 and so on, right. So, that is what of, that is how a virtual address is specified. It is specified using what is called a segment selector or the name of a segment register like cs, ds etcetera and an offset which is the 32-bit offset now, in the 32-bit world.

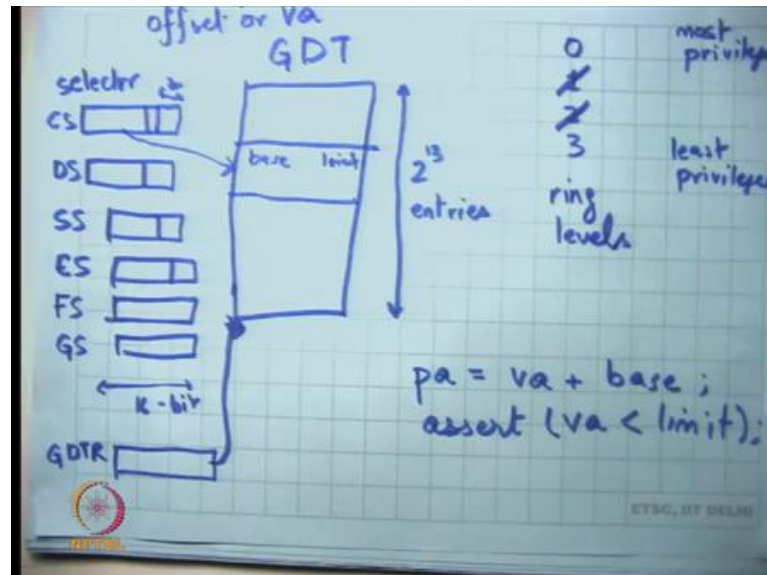
And once again in the 16-bit world it was very simple I just used to multiply this by 14 and add to it, but now we were going to do something, something more complicated because we want to provide protection, all right. So, the selector let's say CS is a 16-bit register of which the top 13 bits are used to index into a table which we call the descriptor table, let us say it is a global descriptor table into an entry which contains things like base, limit, permissions and so on, all right.

So, what is happening? If the application specifies CS: 0x1234, I am going to look at the CS register I am going to take the top 13 bits of the CS register use it to index this table which we call the global descriptor table to get an entry. This entry will have values which say base, limit, permissions among others, right.

And now what the hardware is going to do is going to add the offset to the base before comparing it with limit. So, first it will compare the offset will limit to see whether you know this offset is allowed within this segment or not and then add it with the base and

that is what the physical address is going to become. So, the physical address will be offset coming from here plus base assuming that it was less than limit, right, ok.

(Refer Slide Time: 33:55)



So, let us see here is a global descriptor table let us say I call it GDT. Here our segment registers CS, DS, SS, ES and actually 32-bit has 6 segment registers FS and GS, these are all 16-bit wide, all right. The GDT itself has 2 to the power 13 entries, ok. The top 13 entries, the top 13 bits of each of these points into the GDT and tells you which entry to use to do the translation, right. So, let us say there is an offset that is coming from the user and there is a segment selector, so this is a selector.

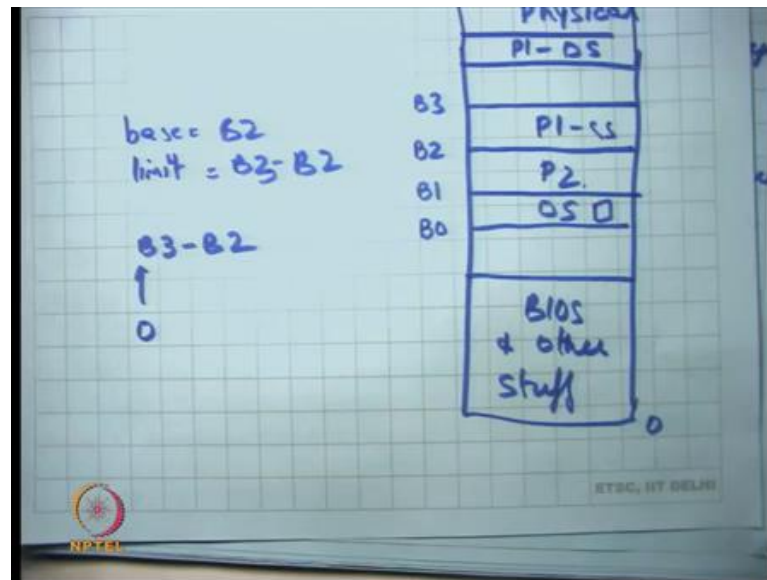
The selector the top 13 bits tells you which entry to look here look at here and from here I am going to get numbers like base, limit, and permissions. But let us just say base and limit for now. What I am going to do is I am going to say  $pa = va + base$ , where  $va$  is this offset. But also assert that  $va < limit$ , all right. So, it is giving you more than what the 16-bit does and the thing that is giving more is that it is also asserting, it is also checking that it is not overflowing, it is permitted limit, ok.

Student: Sir, it asserts some  $va$  or a  $pa$ ?

It asserts some  $va$ , ok. Just a matter of convention, all right, ok. So, what am I going to do? You know if I wanted to implement address spaces, one way I could implement address spaces is I would set up the segment registers in such a way that they point to

memory regions which are allocated for that process and I will set up limits in such a way such that the process is never able to access anything beyond what I have allocated for him, right.

(Refer Slide Time: 36:57)



So, for example, what I could do is let us say this is the physical address space and now let us say this is 0, you know this is BIOS, and other stuff, but here is where the actual memory starts which is usable memory, right. So, from here I said ok, this region I want to give to process P1, right and this region I want to give to process P2 and this region I want to keep for myself, right.

So, what I am going to do is I am going to look at let us say this was base B0, this was base, this was address B1, this was address B2 and this was address B3. When I run when I scheduled process P2, I am going to set up the segment registers in such a way that let us say I want to run process P1, so segment register will P set up in such a way that base will be equal to B2 and limit will be equal to B3 - B2.

And so, when I run this process P1 when it uses a segment selector it can only access this area of the memory, right. So, the process itself will use addresses like 0 1 2 3 or you know any other address.

The legal addresses for a process in this case would be 0 to B3 - B2, right. It can use any of these addresses. 0 gets translated to B2, B3 - B2 gets translated to B3, anything in the



middle just gets translated to  $B2 + pa$ , right. So, the process will just use 0 till some limit and the OS is going, the hardware is going to be converted to the appropriate physical address.

Assuming certain things a process will never be able to access anything outside this region, right. So, that is how I am going to implement address space. Notice that in doing so, a process has does not have to worry about where it was placed in physical memory, right. The executable can assume that it is starting from address 0 or the address space is starting from the address 0 and it has some limit to it, right.

So, when so in the compiler, linker, loader its or the compiler, you know assembler and linker do not have to be worried about exactly where the process will be placed at runtime, right. The canonical there is there is some convention that the operating system is following, in this case the operating system is following that the address starts at 0 and goes up to some maximum limit and so the program safely can be put at address 0 let us say, right, irrespective of where it gets put in the physical memory later on.

Student: Sir, how can we know process needs this much address before the processes started from (Refer Time: 39:56) address space?

So, how do we know at load time how much memory to allocate for that process? In some sense we are limiting the space that the process may have, right that is a great question. Firstly, we know that when you load the, when the OS loads an executable it knows the sense of how much how many bytes the code takes and how many might data takes. It also has some default values of how much space to allocate for the stack, right.

And then it will have no, let us say in this scenario I could also say here is my default value of, here is the maximum value that you can allocate for the heap, right and so the operating system is forced to make this pre-allocation upfront at load time.

The process actually may be a very short process and may not allocate any memory at all in which case I just over allocated or the process may actually not need a lot more memory than what I allocated in which case I under allocated. And in both cases I will have to tell the process using some kind of signal like segmentation fault or something that you have trying.

So, in the first case there is no problem, in the second case you will basically tell the process that look this you are doing something illegal not allowed, all right. But of course, you know I am giving you a first flavor of how address spaces can be implemented, there are more advanced ways in which address spaces are implemented and that allow better more dynamic allocation of space to processes, all right. So, let us just; let us just look at this for the moment, all right, ok, all right.

So, plus actually here I have drawn different processes as different address spaces. By the way though if I wanted to some space to be allocated only for the OS, the OS could say you know this space will never be marked as base and limit for any process, but will be used as base and limit for its own functioning etcetera, right.

Here I am saying that each process must; so, one limitation of doing this base and limit way of doing physical MMU is that a process needs to be contiguous, right. One could relax that requirement by saying that look there are 6 segments. So, I should ideally allow up to 6 different contiguous segments, right. So, an OS could be more complex and say I am going to have you know P1, CS here and P1 DS here or something, right.

So, that way a process is allowed up to 6 different contiguous regions and so in that way it can have more dynamic allocation depending on what it does. So, example something like the code segment needs to be allocated statically in most cases. So, it can just get allocated statically and other things like stack can be allocated on demand or heap can be allocated on demand, ok.

But more interestingly let us understand how an OS can actually set up these segment selectors, but not allow a process to be able to manipulate these segment selectors. So, the hardware has to make some give some hardware support to the process. So, that the process is not able to overwrite the segment selector, right or the process is not able to override the global descriptor table and manipulate the base and limit, right, ok.

So, firstly, how is the GDT computed? So, there is another register on the hardware on the chip, which is called the GDTR global descriptor table register, which points to the base of the GDT in memory. So, this GDT itself is saved in memory, right and so GDTR points to the GDT in memory. And what the hardware is going to do is going to take the selector take the 13 bits index, the pointer at GDTR add these 13 bits to GDTR and

multiply them by whatever the size of an entry is and then use that to get base and limit, ok.

Student: Sir, what have been we have more than 2 to the power 13 processes like because totally child process also. So, what (Refer Time: 44:34) more than 2 to the power 13 processes?

What if there are more than 2 to the power 13 processes? It is ok, right. So, we are I am going to discuss how exactly it is implemented. Just.

Student: Sir, what about the remaining 3 bits (Refer Time: 44:47).

What about the remaining 3 bits? Let us just hold on and we are going to discuss how they are used, all right.

(Refer Slide Time: 44:59)



So, firstly, the GDTR itself should not be manipulatable by the process. So, a process should not be able to set a value of GDTR because of the process is able to set the value of GDTR then he can pretty much do anything he wants, right.

So, to be able to do that firstly, the processor must support two modes, privileged and unprivileged, right. Privilege mode is the god mode, can do anything, right. If you are running in the privileged mode, you can do pretty much anything. Unprivileged mode is

the peasant mode where you know the god tells you exactly what you are allowed to do, right.

Now, the process runs in the peasant mode and the operating system itself runs in the god mode, right. The hardware has certain restrictions on what instructions can be executed in god mode and cannot be executed in, there are, there are certain instructions that can be executed in god mode but cannot be executed in peasant mode.

And the instruction that sets the GDTR, load GDT is an example of one such instruction, all right. So, the lgdt instruction which actually loads the GDTR can only be run in god mode not in the present mode, right.

So, you are going to as you can imagine that you are going to run the process in the peasant mode and because the if the process is running in the peasant mode, the process even if it executes the lgdt instruction what is going to happen? It is going to create an exception, all right. Just like a segmentation fault is going to create an exception, the operating system is going to get notified and it can for example kill the process, saying that you are trying to do something illegal.

On the other hand, the operating system is free to execute this instruction that is number one, all right. So, the GDT can only be executed set by the OS and not by the process, ok.

Second, the segment registers should they be allowed to be overwritten by the process?

Student: No.

Well, one answer is no. But, actually on x86 a process is allowed to overwrite its own segment registers, right. For example, it may be using you know the compiler may be using things like it I want to shift my code segment somewhere, right. So, the architecture designers did not want that user should not be able to set up its own segment registers because segmentation, the segment registers are used for multiple for more purposes apart from this protection.

So, user is allowed to override the segment registers, but then what protects it? What disallows it from doing something wrong? It can only set up these registers to one of the allowed values in the GDT, all right.

So, what it can; so, what an OS can do is that it can engineer the GDT in such a way that only certain values, certain base and limit values are present in the GDT and so even if the program changes its selectors it can only change it to one of the allowed values, right. So, if there is no base and limit for the OS address space in the GDT then it will never be able to change to that. It will never be able to access the OS's address space, all right.

Student: Sir, we can still change the segment to another process (Refer Time: 48:42)?

We can still change the segment to than other processes base and limit. No, even that is easy, right. Before a context switch to this process, I am going to switch the GDT such that it only contains entries of that particular process, right.

So, at the context switch time before I start this process running, I will remove the GDT of the previous process or I will modify the GDT such that the previous processes entries are not there and the new processes entries are there and only the new processes entries are there. So, the new process is allowed to freely change his segment registers between these new entries, but he is not able to access any other processors memory or its own memory or the OS's own memory, all right.

Convincing that this is a this is a way of ensuring that applications have private address spaces and do not are able to write on another application that is space and are not able to write on the OS's address space.

Some more facts. How does the processor know what is the current privilege level? The way an x86 processor decides what is the current privilege level is by the last two bits of the CS register. So, the last two bits of the CS register tell him, tell the processor in whether I am working as god or whether I am working as peasant, all right. So, the convention is that 0, so they are two bits. So, the you have 4 possible values 0 1 2 3, all right and 3 is the least privileged and 0 is the most privileged.

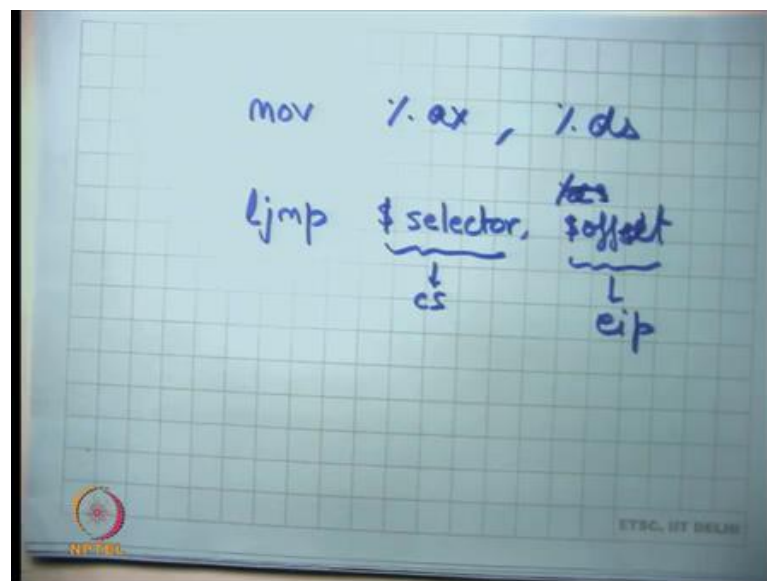
Most operating systems just use 0 and 3, do not use 1 and 2 at all, right. So, the operating system set, when the operating system is running the last two bits of CS are set to 0 and when the application is running the last two bits of CS are set to 3, right. So, 0 and 3 is basically, decides whether I am running in this mode or that mode, all right.

Student: Sir, when the process can modify CSO, if it modifies these bits than it will if the?

Interesting. So, a great question. But a process can modify its CS. So, if it modifies its CS then it can you know change its privilege level at will. So, the hardware also makes sure that only certain types of modifications are allowed, ok. So, when you modify the CS if you cannot go from a less privileged mode to a more privileged mode we can go in the other direction for example, right, ok.

But in some cases, for which we need for implementing system calls which you are going to discuss later but let us just assume for now that a process cannot in general just upgrade its privilege level by modifying CS, right. If it tries to do that, so you can change the CS and the way to change the CS. By the way how can you change that how can you change DS? So, there are instructions how you can change segment registers?

(Refer Slide Time: 52:09)



The instructions like move, a register like ax to the segment register. So, this is a valid instruction. A process is free to execute this instruction to change a segment register. But CS is an exception you cannot just execute this instruction on CS, just like you cannot change EIP directly you cannot change CS directly.

So, the way to change CS is basically what is called a ljmp instruction, all right. And ljmp basically say takes arguments selector and offset, right. So, we have seen the jump

instruction before, where we just say jump to an address it just changes EIP to that address. ljump takes two arguments select and offset and the semantics are that it will set the selector in to CS and set the offset into EIP, right that the semantics of ljump.

And when it executes the l jump instruction only certain values of offset are permissible. So, for example, if I am executing in at privileged level 3 and I execute l jump to a selector which has privilege level 0 in it, that is not allowed, it will cause in the hardware to generate an exception and the OS will come into action, right. So, only certain values are allowed. So, basically you are not allowed to go from a less privileged mode to a more privilege mode, ok.

Finally, what prevents an application from just modifying the GDT? So, GDT lives on memory, right. Why cannot an application just say fine figure out the address of GDT and just write something there?

Student: Privileged instruction.

So, one answer it is you require a privileged instruction to do that. Is that right? Only need a memory move instruction, right just some move instruction is going to suffice to be able to write to a memory address.

Student: Because only base is privileged more (Refer Time: 54:14).

Right. So, the GDT is stored in the region of memory which is only accessible to the OS and not to anybody else. So, when I have this figure and I say this is the physical memory. So, GDT is probably going to live somewhere here, right. And because this area is not mapped into the address space of any other process no process can actually modify GDT, right.

So, we are looked at protection. We saw that a process has different privilege levels, let us say there are two privilege levels at least, unprivileged and privilege. Certain instructions can only be executed in privileged mode the loading of GDT is one such privilege instruction.

Then we said that the segment registers are, can are free to be changed by the application, but they can only be changed to certain values and those values are dictated

by the values in the GDT, all right. And you cannot just lower your; lower your privilege level or upgrade your privilege level or lower your you know, so low your ring level.

So, this 0 1 2 3 are also called ring levels, right. Lowering a ring level means upgrading your privilege level. So, an application cannot just lower its ring level and so that is another thing. And third, finally, because GDT is a protected structure, a protected structure must live in a section of physical memory that is not mapped in the address space of any other processes memory and only in that is space or the OS memory, ok.

Let us stop here. And, we are going to continue this discussion next time.