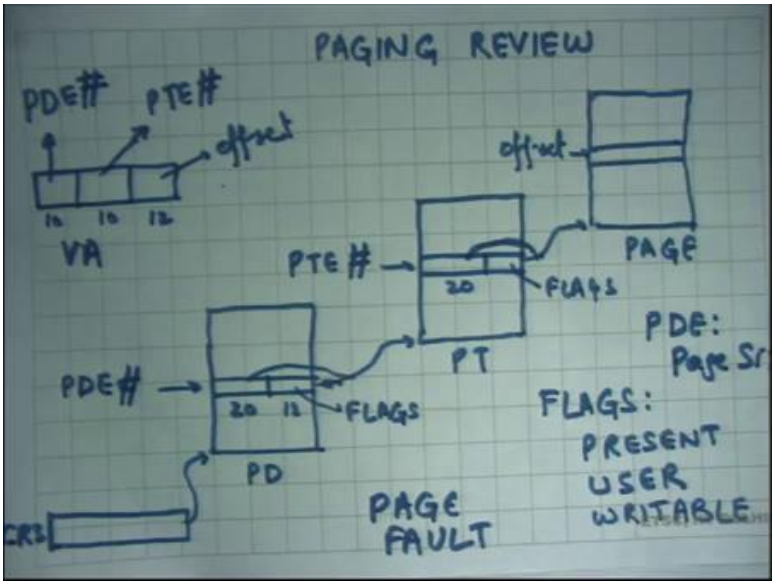
Operating Systems
Prof. Sorav Bansal
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture – 13
Translation lookaside Buffer, Large Pages, Boot Sector

So far, we had been looking at paging and let us review it once again.

(Refer Slide Time: 00:09)



We said that segmentation allows you mapping from virtual address to physical address, but it only allows contiguous mappings because it is a simple base plus VA gives PA mapping which is not very flexible. It does not it has problems of fragmentation and it has problems when if a process wants to grow etcetera.

So, if you add a more general mapping and that is what paging implements, then it would be much more flexible. And, what paging does is divide the physical address space and the virtual address space into page lined, fixed size, page sized units right which are called pages. So, aligned fix size units which are called pages and, now, there is a mapping hardware in the middle which will map a page in virtual address space to a page in physical address space.

This mapping hardware is basically requiring a table which can have at most you know 2 to the power 20 entries. So, such a large table cannot be stored on chip. So, the such a

table needs to be stored on physical memory right or you know the technology for physical memory is also called DRAM.

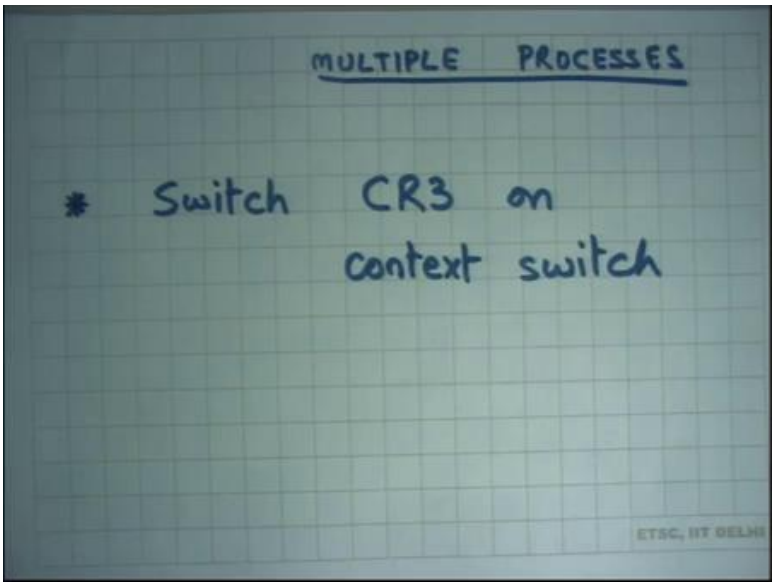
So, you store the table in physical memory also we said that rather than storing the table contiguously which will make it very large 4 megabytes; let us divide it into a two-level hierarchy, and so, that way small processes will only have a very small page tables and large processes will also you know will also benefit in space. Even if there are all the pages mapped, even then you know the two-level hierarchy is not too much space too much space hungry than hungrier than one-level page hierarchy alright.

So, on the chip there is a register called CR3 which holds a physical address, which points to the base of the page directory. The virtual address is used the 10 bits out of the 10 bits of the virtual address are used to index into the page directory from which you get the 20-bit physical address of the page table and 12 bits of flags. These flags are you know three use three relevant flags for us are present, user and writeable. So, whether the page is present, whether page table is present in this case, whether user mode execution can access it and whether it is writable right.

So, for example, if this is non-writable then the all the pages which are reference from here, they are all non-writable right. On the other hand, if this is writable then depending on what the flags are for the in the page table corresponding page table entry the page will have writable or read only permissions alright.

So, the top 10 bits are used to index the page directory, the next 10 bits are used to index the page table, and the last 12 bits are used as an offset into the page alright.

(Refer Slide Time: 03:02)



And, we said that an operating system can implement multiple processes or multiple address spaces by switching the page table on every context switch right. So, each time the process wants the operating system wants to change from P1 to P2 you change the CR3 value from P1 space directory to P2 space directly. You just load a new value into CR3, and you have a new page table.

And, we have also said that the kernel maps itself into the address space at the same point in every page table. So, a process does not see the entire 4 gigabytes of virtual address space, it sees something less than that. On xv6 it only sees a bottom 2 g gigabytes and the top 2 gigabytes of the virtual address space are reserved for the operating system or the kernel. And, so the kernel is really mapped at the same place pointing to the same physical addresses in every process right.

And, so we also said that every hence every user process is also a kernel thread right because a process now has two halves. One is the user half which the user can access and the other is the kernel half which only the kernel can access. A user cannot just switch into the kernel half the only way a user can switch into the kernel half is through the interrupt descriptor table through a trap right.

And, once it is switches into those kernel halves then it executes in kernel mode. It is possible that multiple core processes simultaneously are executing in the kernel half in which that is why we are calling them kernel threads ok. On the other hand, if the

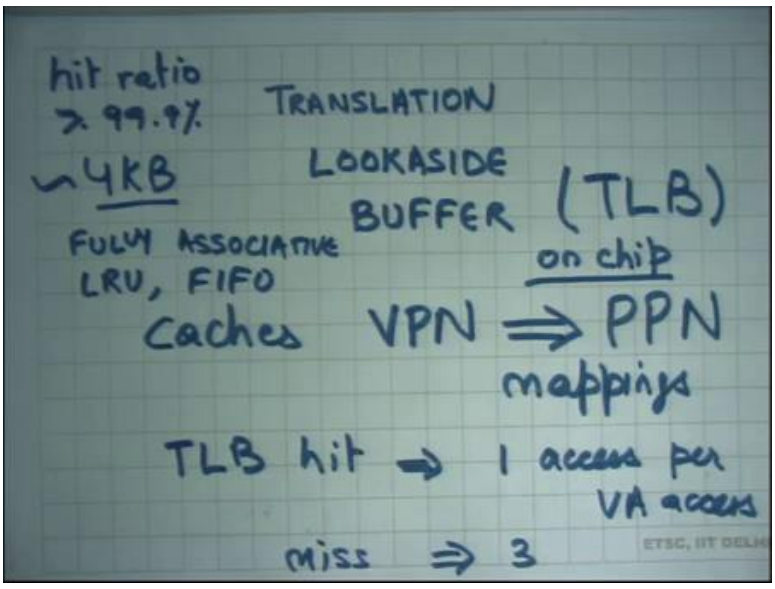
multiple processes are executing concurrently on the user half no problem because these are independent address spaces alright. So, how many accesses memory physical memory accesses does the processor need to make on a memory accessed by the program?

Student: Three.

Three. So, first you will dereference first you will add these 10 bits through CR3 and then dereference this value. So, that is memory access number 1. Then, you will get these bits and dereference this value – that is memory access number 2 then you will get this and then dereference this value, that is 3 right. So, for every memory access you are actually making three physical memory accesses. So, for every virtual memory access you are making three physical memory accesses and that is not acceptable right that is too costly.

In segmentation, we saw that such a problem was solved by basically ensuring that when a segment descriptor gets loaded the segment descriptor gets cached into the chip right. So, you do not need to access memory to get the base value each time; base value is just cached on chip.

(Refer Slide Time: 05:50)



Similarly, in paging, also there is caching involved and there is a special cache called the translation lookaside buffer translation lookaside buffer or TLB for short ok. This cashes

VPN virtual page number to PPN mappings. So, each time you basically access a page, a virtual address you walk the page tables the 2 two-level page table and you get a physical address right. But there was a mapping from the virtual page number to the physical page number that this page table structure was providing.

So, if you could cache this end-to-end mapping from VPN to PPN in your TLB then and you could first check your TLB and so, TLB is on chip right. This is on chip, on CPU. So, if the virtual address the VPN corresponding to the virtual address being accessed is already in the TLB, you do not need to walk the page table, you can just get the VP PPN from there add the offset and directly access the memory.

So, that way you just have to have if you get a TLB hit implies 1 access per VA access right 1 physical memory access per VA access. A miss implies 3 right. So, after the TLB miss you have to walk the page table just like before; if the TLB hit you do not need to go to the physical memory right.

So, it is important in real world that most of your memory accesses are TLB hits. You do not know most of your virtual address accesses are TLB hits for to have any acceptable performance right. Otherwise you know the whole idea of paging falls flat if the TLB hit ratio was not high enough right because segmentation was giving me a very fast translation. If paging is giving me a 2x overhead on every memory access that is not; that is not acceptable right.

So, fortunately most programs have a very high locality of access. Especially, because we are dividing the address space into page level granularity one entry one VPN to PPN entry is capturing locality of access within an entire page right. And, so it is possible to have a small TLB and yet have hit rates of 99.9 percent or above right and that is the kind of hit rate you will want in such a system to have any acceptable interrupt acceptability of this paging idea right.

So, you know typical TLB sizes on modern processors will be let us say 4 kilobytes also right roughly. So, you know a 4 kilobyte TLB can cache let us say; let us say each TLB entry requires 8 bytes, then a 4 kilobyte entry will 4 kilobyte TLB can cache 512 such pages page entries and assuming that your memory footprint. So, 5 so, you can basically cache 512 VPN to PPN mappings in your TLB and assuming that the programs memory footprint is less than 512 pages, then pretty much you should be able to access you know

this translation should be pretty much free. You are only accessing the on-chip TLB alright.

Also, let us understand whenever we talk about caching so, assume you have seen caching and you know what is hit ratio and all that right. So, I have been talking about hit ratio and let us say this is you know greater than 99.9. Let us say, now, just a ballpark figure 99 to 100 percent somewhere in the middle and that is the kind of hit ratios you are looking at ok.

So, when you talk about caching, we should also say you know what its cache replacement policy is? what is its associativity; what is the whether it is a write through or write back cache right. So, you have seen all these terms in computer architecture class right. So, firstly, what is its associativity which means you know you know about cache associativity.

So, typically the TLB is a fully associative cache right you want your TLB to have such a high hit rate, so, you better make it fully associative. If on the other hand you had a direct mapped cache you will just have conflict misses in your cache right ok. The other thing is what the caches replacement policy. Well, the hardware can is free to implement any cache replacement policies it likes. Given that most of the things are going to be hits.

You know cache replacement policy does not cache replacement policy usually matters if the size of your caches is small. If your cache size of the caches is bigger than the working set size, then cache replacement policy is actually not that crucial. It does not really matter what your cache replacement policy is.

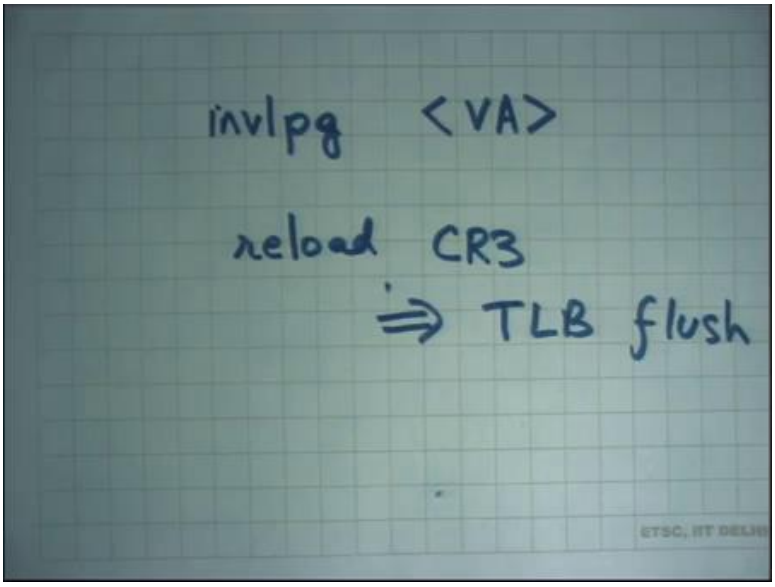
So, you know you can use something like LRU which will require some bookkeeping or simpler policy maybe FIFO and the hardware designer can make it make a choice you know. FIFO is simpler to implement, but it may have a slightly lower hit ratio than LRU. LRU is more complex to implement. It may or may not have better hit ratios than LRU you know that.

Now, let us look at whether it is a right back or a right through cache, let us see what happens. Each time a page table is walked by the hardware the entry gets cached into the TLB. So, this is a read operation, right? All these lookups are just reads of the page table.

The only right to the page table is then the kernel actually overwrites some entry by looking at that particular address right.

Recall that the page table itself is mapped in the kernel address space right. So, the kernel can change the page table by just writing to a memory location there. If the kernel changes a page table entry TLB does not even come to know about it right. So, a kernel needs to space the kernel needs to tell the hardware explicitly to invalidate a page directory entry or TLB entry right.

(Refer Slide Time: 12:59)



So, there is an instruction called let us say invalidate page and it takes a virtual address right; basically means invalidate any entry in the TLB corresponding to this virtual address right – so, explicit invalidation of the TLB entry, the cached entry. If the kernel forgets to do that, it is a bug right. Very bad things can happen ok.

You can imagine what can happen. What can happen is that the kernel thinks that it has mapped a certain page somewhere else and removed the entry here where the TLB still caches that entry and, now you know a user can probably access somebody else's page right. So, those kinds of bad things can happen.

So, it is important of the kernel executes each time it changes a page table entry it invalidates the virtual the page all TLB entries corresponding to a particular virtual

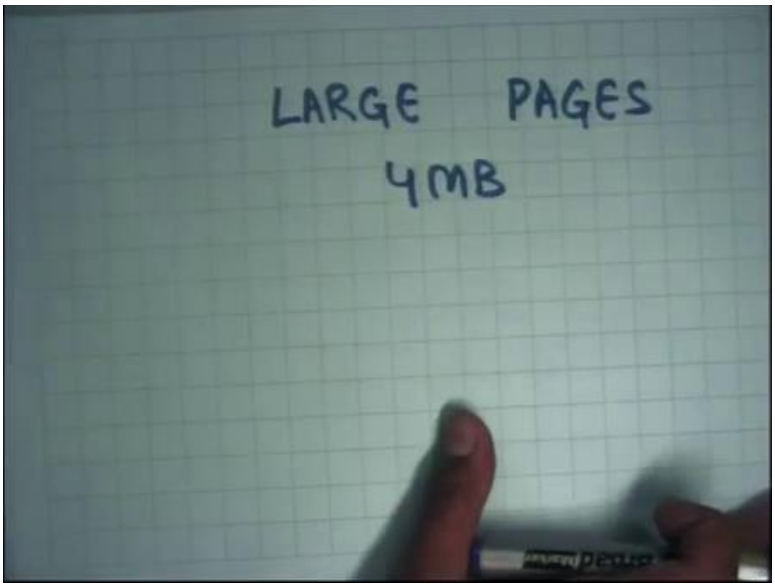
address or the single TLB entry corresponding to virtual address. Also, each time you do a context switch you change CR3 right. So, what should happen to the TLB?

Student: (Refer Time: 14:04).

All that you know the entire address space has changed right. So, you should completely flush the TLB right. So, reload of CR3 implies TLB flush right. So, in every context switch you flush the TLB completely because you know the entire address space is changed all the VPN to PPN mapping have only cache VPN to PPN mapping need to be invalidate alright.

So, it is important to for an operating system to ensure that the number. So, the number of entries in the TLB typically remains small you know and one way that one method that the hardware provides to do that is supporting what are called large pages on x86 right. So, let us see.

(Refer Slide Time: 15:09)



There is something called large pages ok. So, we said you know normal pages are 4 kilobytes, but large pages are 4 megabytes. So, you could have large pages and how would you how do the x86 architecture implement a large page? In the flags of the page directory entry there is yet another bit. So, for PDE flags there is another bit which says page size alright. If the page size bit is 0 it means it is a normal page; if the page size bit is 1 – it means it is a large page.

And, if it is a large page it treats the 20-bit pointer not as a page table, but as a pointer to that 4 MB page ok. So, if it is a large page then this pointer is pointing to a 4 MB page directly right because you can imagine that you know a large page if your machine is talking about a 4 MB page you have divided your physical address space into pages of size 4 MB right.

And, so pages of size 4 MB means 2^{22} bits I will basically use for an offset right and the top 10 bits; 22 bits are used for the offset and the top 10 bits are used to identify the page number right. And, so the top 10 bits of the virtual address are used to index the page directory entry to get the large page number right. Even large pages need to be aligned at page granularity right.

So, large page cannot, a large page just always starts at 4 MB boundaries right. So, there will be a large page at address 0 and then the next large page will be at address 4 MB and 8mb and so on; just like small pages were aligned at 4kb large pages are aligned at 4 MB right. And, so, 10 bits are enough to name a large page and so, the top 10 bits of the virtual address are enough to name a large page. And, so, the page directory entry directly points to a 4 MB region right.

If you are using small pages, then you allow this 4 m in any case the page one entry in the page directory is quite capable of mapping 4MB of virtual address space. If you are using small pages, this 4MB can be fragmented in 4kb chunks across the physical memory if you are using large pages then this 4MB chunk has to be contiguous in physical memory right that is the only difference. So, what is the advantage of doing this?

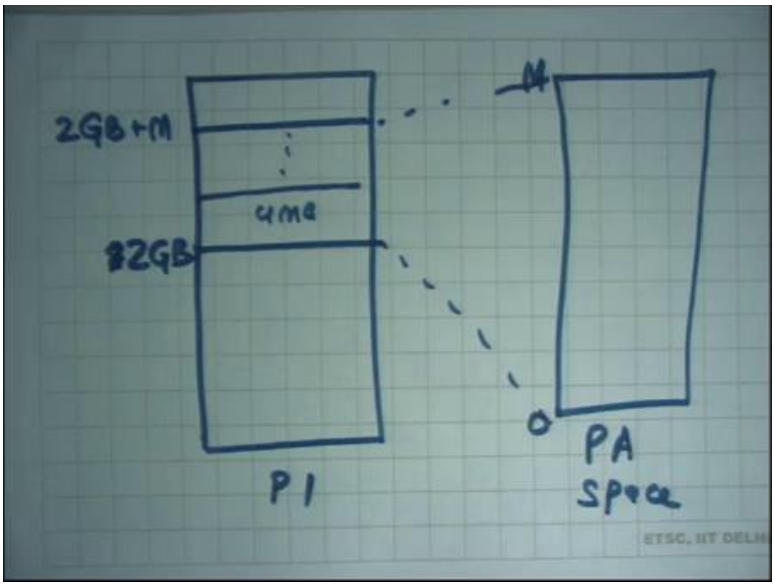
Student: Time.

First advantage is the time it takes to actually walk the page table has reduced. You only need to dereference one, but you know make two accesses for one even if there is a TLB miss. More importantly the number of entries that need to be cached in the TLB has reduced alright. As supposed to 2 to the power 10 entries for a 4-megabyte space you need only one entry, if you can ensure that the entire space is contiguous.

But it also means that the program should you know the operating system should take care of things like fragmentation right. For example, small pages small processes should

not be given large pages. Large processes can be given large pages, but in make sure that they are actually using those large pages because an entire address space, if they are not then I am basically worrying about fragmentation and all that kind of issues. One big place where large pages are really useful is to map the kernel itself alright.

(Refer Slide Time: 18:57)



So, we said that every process let us say this is P1 maps the kernel starting at you know starting a 2GB, let us say right and we said this mapping of the kernel is basically a one to one mapping to the physical address space. This is the physical address space, and this is going from 0 to M, then this is going to from 2 GB to 2 GB plus M right.

So, it is a completely contiguous mapping in the kernel right. That is what we saw that the kernel just maps the entire physical memory into a charger space; at least in xv 6 and we said other operating systems can recycle virtual address space for if you know the amount of physical memory is greater than what can be supported in the address space.

So, for this kind of a mapping which is completely contiguous it does not make sense to use small pages you can just use large pages right. So, you can just have you know 4 MB pages to store this mapping from for the kernel; that way, the kernel the size of the there few advantages of this number 1 – the size of the page table has reduced right because you only need few entries to map the kernel you have made the generality bigger.

So, the overhead that we had so, we said that every process needs to map the kernel. So, every process has to has this extra overhead of having these extra entries for the kernel address space. If you are using large pages, then this overhead has significantly decreased number 1. Number 2 – that most more important the TLB pressure has decreased.

The number of entries that are needed to be in the TLB to cache this mapping has in decrease and so, you can have better hit ratios in your TLB right. So, large pages can be used other places, but one place where they have a direct use, immediate use in the operating system designs that we are talking about you know you just use these large pages to map the kernel itself.

Student: Sir, while is saying while caching a page we (Refer Time: 21:04) to a 4MB in the cache it is very large value.

While catching a page, no, you mean in the TLB?

Student: Yes.

So, in the TLB you only store one mapping right.

Student: Yeah.

VPN to PPN. If you are using 4 kilobyte pages.

Student: We (Refer Time: 21:18) can get 2 to the power 10.

You could potentially have 2 to the power 10 entries for to map an entire 4 MB space right; on the other hand, if you're having large pages then you will have only one entry for that entire space. But, large pages has have problems of fragmentation and wastage of space potentially if you are not careful and so, that there is a trade off, but here is one here is one case where it is a no brainer to always use large pages. Yes, question?

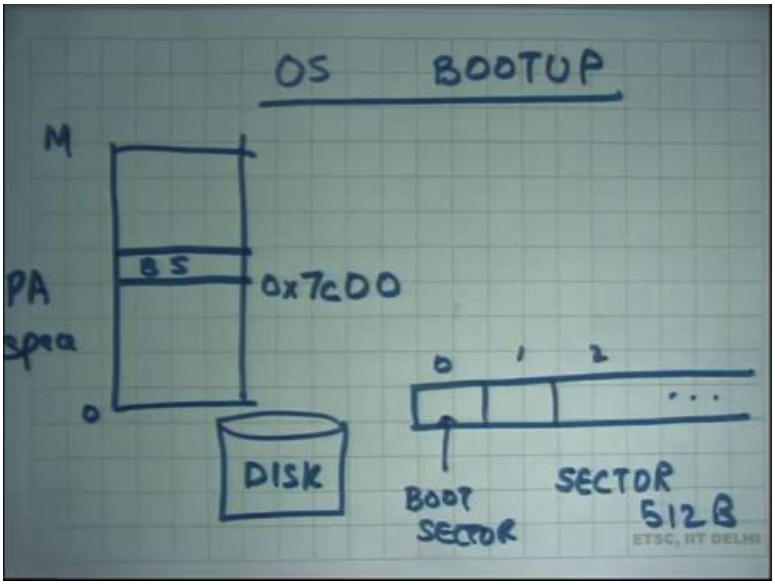
Student: Sir, since that space going to be constant so, why not use directly map bit? Why use any sort of paging mechanism?

So, question is why use any sort of paging mechanism to map this space. You know it will be great if the hardware could provide me a mechanism saying map the entire space

here, but the hardware does not do that right. I mean once you have enabled paging you have to go through the page table alright, and you know the one way the hardware designer code had said was you know have a other bit saying that this is not going to go through paging. This particular address is not going to go through paging you know it complex it makes the hardware even more complex.

So, you know one way to do that is just use the existing page table hardware and large pages it can be used both for the kernel and for other things also ok. So, the hardware designer does not necessarily want to complicate his hardware just to support the kernel space mapping when it he can achieve the same effect by using a more general mechanism of using large pages alright.

(Refer Slide Time: 22:58)



So, with that let us continue our discussion of from yesterday. We will be looking at how does the OS boots. Let us look at the OS bootup and we said let us say here is a disk and here are it is blocks block 0, 1, 2 and so on and we said this particular block or sector is special right.

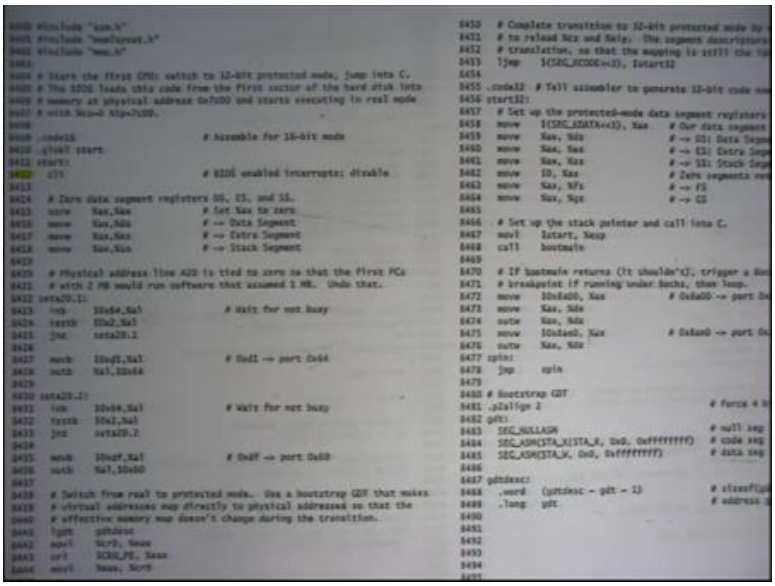
So, each of these sectors is 512 bytes and sector number 0 are special, it is called the boot sector and the contract with the hardware is that it will load this boot sector at a particular memory location. It will copy this boot sector in the particular memory location.

So, let us say this is the physical address going from 0 to some value let us say M. So, it is going to copy these 512 bytes at some location the boot sector BS and the address at loads it is as it as 6 7c00 alright. I mean these addresses are just historical in nature you know. This must have been the address at which it had loaded you know back in the 1980s and it still continuous because for backward compatibility we want that the operating system that was written in 1981 should still run right. So, for that reason it has to do the same thing ok.

So, and we said that what and so, what the. So, the operating system developer needs to write a boot sector code that should know where the kernel lives in the disk number 1, and it should load the kernel into memory and then jump to the kernel.

So, all that operation needs to be coded up in the boot sector and all this the code to do all this should fit inside 512 bytes I means that is the constraint and that is relatively easy to meet it is not a big deal and we are going to look at the boot sector code of xv6 alright ok. So, please take out your and this thing code listings.

(Refer Slide Time: 25:22)

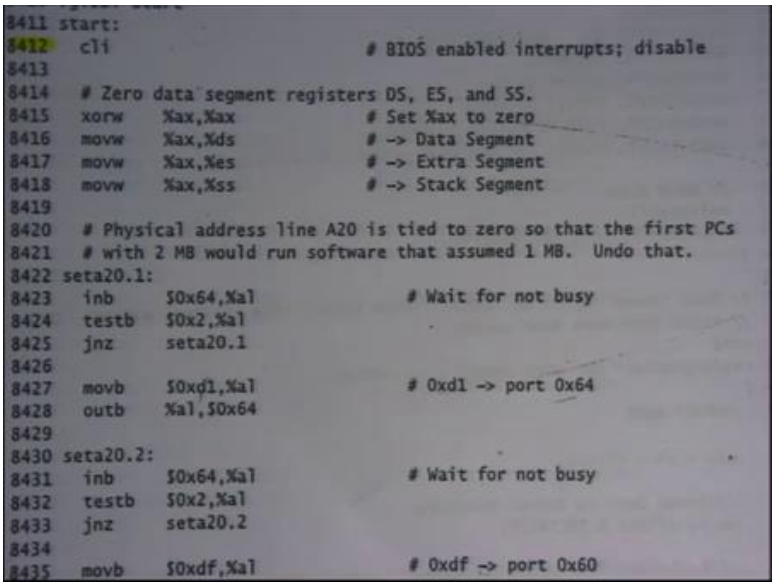


So, firstly, we are looking at code of xv6 right this code will get compiled using let us say a compiler like GCC and we have discussed this before and then the code will get linked right. And, then you will get a an executable; that executable is what we call the kernel you know the xv6 links into what is called in an executable file which is called the

kernel right and that kernel will also and there will be another sort of file which will be the boot sector code right.

So, and the linker will set up the addresses and set up things in such a way that the boot sector code will live in the first sector of the disk number 1 and number 2 the boot sector code know as that will start at a certain address called 7c00 right that is the hardware specification.

(Refer Slide Time: 26:35)



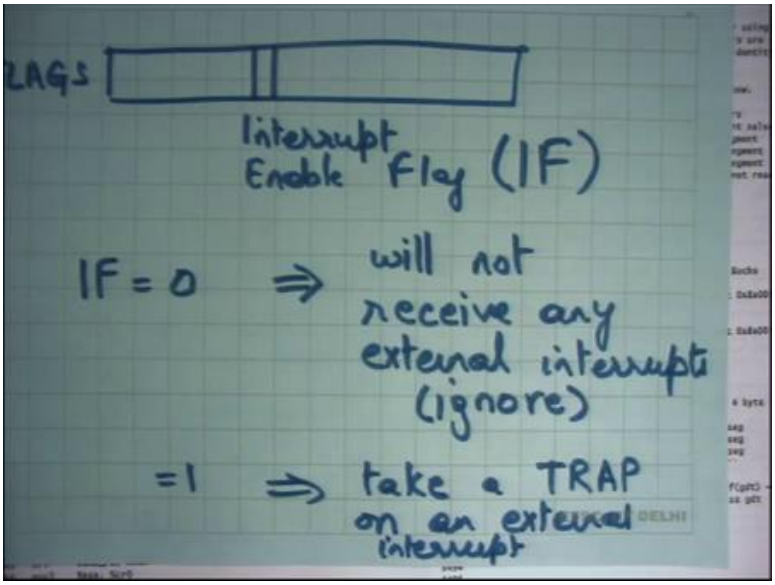
So, let us assume that the boot sector. So, the boot sector code actually starts at this point right 8412. So, this is the assembly code of the boot sector alright and the line 8409 is saying code 16. It basically means treat this code as a 16-bit code right. So, this is a 16-bit code.

The next line says global start which basically means consider this symbol start as a global symbol what it means we can talk about it later. So, this is the first instruction that gets executed on when the computer gets boot up booted up in xv6 the first instruction; in this case is cli; cli is basically just disabling interrupts right. So, what does the cli instruction do? Recall that the x86 architecture has this register called EFLAGS.

And, one of those one of the bits in the EFLAGS is whether interrupts are enabled or disabled. If interrupts are enabled, then an external device is allowed to assert the interrupt pin and my execution will switch to the handler immediately. If interrupts are

disabled even if the external devices assert interrupts pin, I will not switch to a handler right ok.

(Refer Slide Time: 28:00)



So, first recall that the x86 has a registered called EFLAGS. One of the bits in the EFLAGS is what is called the interrupt flag or interrupt enable flag or IF ok. The semantics are if IF = 0, implies will not receive any external interrupts. In other words, just ignore the external interrupt and if it is 1, then it is the usual thing you know take a trap on an external interrupt ok.

So, if IF = 0, you just ignore the external interrupt; if I = 1, then you will take a trap on the external interrupt and recall that the trap goes through the interrupt descriptor table to call the handler. Why the first instruction is clearing the interrupt?

Student: Because there is no handler.

Because I have not set up any handlers yet right. The BIOS may have had set up its own handlers some sort there is some you know BIOS that has run before the first instruction has run it may have set up its own handlers and, now I want to completely forget what the BIOS has done to the system. I want to now you know reinitialize the system according to myself and while I am doing that I just I do not want any disturbance from outside world right.

So, if an outside there is if there is a network packet coming or anything of that sort or you know disk wants me to want some attention just to ignore all that. I am not in the state to actually be serve all these people. So, let us clear the interrupts alright. So, that is what. So, cli basically sets IF to 0 ok. Then the next thing you do is you say xor ax; ax the effect of xoring register with itself is there you just 0 out the resistor ok. And, then you move ax to all the segment registers DS, ES, SS. Why am I doing this?

Student: (Refer Time: 30:31).

To have a flat address space right. I do not to have do not have a segmented address space. Let us set up my segment register recall that in 16-bit mode the segment registers the segmentation works by simply multiplying the segment register value by 16 and adding it.

Student: VA.

VA virtual address to get a physical address. So, I do not want any segmentation so, I just set it to 0. So, my virtual address is equal to physical address from now on. So, from now on if my virtual address is equal to physical address.

Student: Sir.

Yes?

Student: Like instead of xoring could we have directly moved dollar 0 into x?

Yes, could you have directly moved dollar 0 into x? Yes, you could have. Why is he doing it in this way? It turns out it is more efficient to do it and this is you know this is a standard thing that assembly programmers use that you know instead of moving 0 to ax just xor it with itself that turns out to be more efficient counter intuitively right. Anyways, I mean you could have done the other thing also.

Student: Sir, other than we could have xored all three of them with themselves?

So, you know, but see DS, ES, SS are special registers. They cannot only you know a certain instruction move can be used on them. You cannot use arithmetic instructions on segment registers. You can only use arithmetic instructions on general purpose registers which are a es a, b, c, d, esp, ebp, esi, edi alright. So, you can only use arithmetic and

you cannot even move an immediate value to the segment register directly, you can only move it through a general-purpose register. So, these are just constraints of the architecture alright. So, that is why he is doing it like this alright.

Then there is some code to allow addresses above 20 bits. So, original 8086 machines did not allow addresses which are greater than 20 bits right at that time the machine was 16-bit, but now we want a 32-bit architecture. So, there is some code to allow addresses above 20 bits in any case we can ignore this alright. So, let us ignore this. It is needed for a program to run, but it is not needed for us to understand what is going on alright. So, let us just ignore this.

(Refer Slide Time: 32:38)

```
8423   inb     $0x64,%al               # Wait for not busy
8424   testb   $0x2,%al
8425   jnz     seta20.1
8426
8427   movb    $0xd1,%al               # 0xd1 -> port 0x64
8428   outb    %al,$0x64
8429
8430 seta20.2:
8431   inb     $0x64,%al               # Wait for not busy
8432   testb   $0x2,%al
8433   jnz     seta20.2
8434
8435   movb    $0xdf,%al               # 0xdf -> port 0x60
8436   outb    %al,$0x60
8437
8438   # Switch from real to protected mode.  Use a bootstrap GDT that mak
8439   # virtual addresses map directly to physical addresses so that the
8440   # effective memory map doesn't change during the transition.
8441   lgdt    gdtdesc
8442   movl    %cr0, %eax
8443   orl     $CR0_PE, %eax
8444   movl    %eax, %cr0
8445
8446
8447
```

And let us instead come to this instruction lgdt gdtdesc. What am I doing? I want to load up load a.

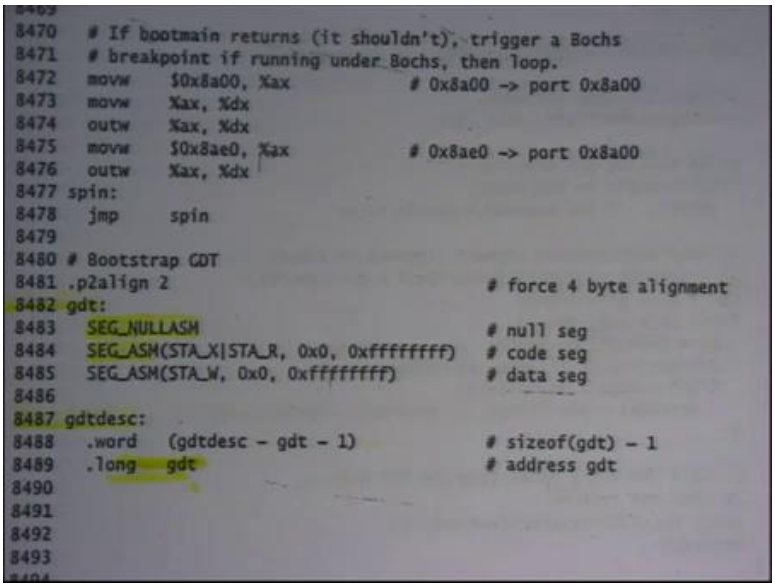
Student: Global descriptor.

Global descriptor table and notice that this lgdt instruction is actually a 32-bit instruction alright, it is not; it is not a 16 bit instruction because in 16 bit there was no gdt. In 16-bit segmentation was just multiplied a segment selector by something and that is it the gdt is only in this protected mode.

And, so what I am going to do in the next four lines is basically switch to 32 bit mode right, but before I switch to 32 bit mode I need to set up my gdt and all these things, so

that when I switch to it knows exactly where to where am I standing and all that right ok. So, what are the semantics of lgdt gdtdesc? gdtdesc itself the descriptor is defined here at line 8487.

(Refer Slide Time: 33:33)



Student: (Refer Time: 33:34).

And, it contains the location of the gdt, and it contains the size of the gdt right. Recall that we said that gdt can specified by a size and a base also right. So, this is the base of the gdt; so, 0, 1, 2, 3 and there is a maximum size of the gdt. So, that is what you specify the gdtdesc and let us look at where gdt is that is at this line 8482 right which contains your segment descriptors right.

So, segment descriptors are nothing, but numbers right they are just numbers which basically specify this flag should be set up, this is the base, this is the limit and all that kind of stuff right. So, in this case here is a number called SEG NULLASM which basically says 0th descriptor is null – no never use it, nobody should ever use it alright.

The second one says it is using a macro. So, these are all macros. SEG NULLASM is the macro which will get macro expanded. So, you can actually browse the xv6 code to see exactly what number this SEG NULLASM represents, but you know whatever the number is it basically sets up the gdt 0th entry to say that this should never be accessed basically dereference. The second says SEG underscore ASM which is again a macro.

STA X which says give execute privileges. X is for execute STA R is saying give read privileges. So, these are basically specifying what flags I want in the gdt descriptor. 0 says what is the base of the gdt descriptor and this ffff says what is the limit of this gdt descriptor, alright. So, you use a macro to say construct the gdt descriptor with flags, execute and readable and base 0 and limit 2 to the power 32 minus 1 ok.

So, the intent is that the descriptor number 1 is going to be dereference for all your code right. So, you are going to load the value 1 into cs the descriptor number 1 into cs and so, code will always go through this and so, that is why it will become executable alright and all the other ones will have writable which means writable means it is both readable and writable and base is 0 and limit is again 2 to the power 32 minus 1 right.

So, you have a gdt of size 3, the 0th is a null entry, the first entry is an execute is pointing to an executable segment which is the entire address space, then the second entry is pointing to a writable segment which is again the entire address space. The developer has separated its code and data in this way. So, we will load code in cs and data in all the other segments right.

You may say could I have had a one a single segment which is readable, writable and executable and load that the same thing in everything that is also perhaps possible depending on you know whether x86 allows that kind of thing that the segment is both writable and executable alright.

But, in any case they are you know they are sharing the entire address space. It is not like cs is living somewhere else and all the other segments are living somewhere else. They actually living in the same space the different segment descriptors are only being used to check the type of access execute versus right clear alright ok.

So, see you will load the gdt and then you do some things to make sure that you are moving into 32-bit mode. So, CR0 PE basically says move into protected mode. So, the 32-bit modes that you have discussed is also called the protected mode because it offers protection right and so, when you and then you move this value into CR0. So, there is a control register 0 which indicates which mode I am running in and I am running in 16-bit mode, 32 bit mode or protected mode etcetera and so, that is how you move to 32-bit mode.

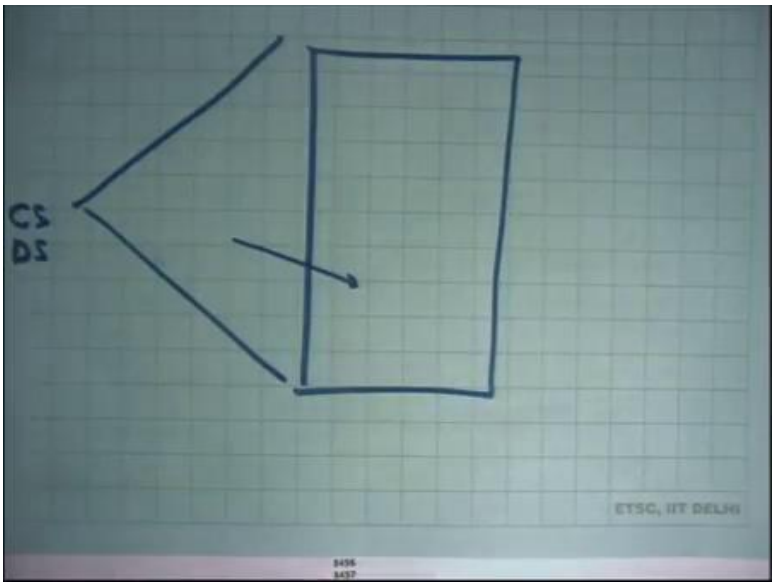
Student: Sir, why we have taken the limit for both data segment and code segment from 0 to 2^{32} - 1.

Why have we taken?

Student: The limit from 0 to 2^{32} – 1 because code and data are two different part workflow limits should be like 0 to for example m and from m to 2^{32} - 1.

So, question is why both of them are referring to 0 you know are pointing to the same region 0 to 2 to the 32 minus 1? Why not code referring from 0 to some m and data from m to 2 to the power 32 minus 1? Well, I mean he could have done that, but the prefer I mean that complicates the programming model. You know sometimes you want to access code just like data.

(Refer Slide Time: 38:31)



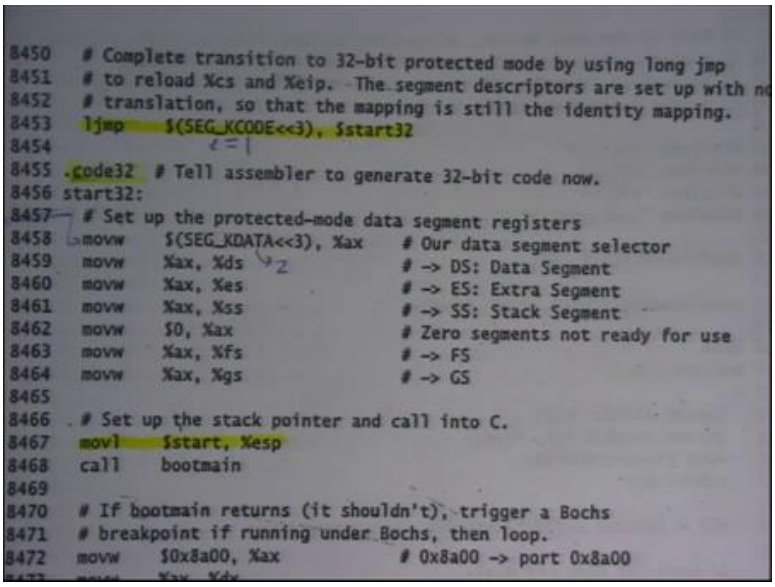
Student: But we will not be differentiating over rather we want to execute it or we want to write over it.

Right. So, basically this kind of organization allows you to see a flat address space right. It does not matter which segment you are going through you will always VA will be always equal to PA right. On the other hand, if you do the other thing then I will have to worry about you know whether I am going to through this segment or that segment and if you have a flag address space it allows you to change code using directly it is address

and then execute it and because all of them are pointing to the same region you do not have to worry about you know CS is pointing here and DS is pointing here.

So, all these segments CS, DS etcetera they basically see identical things right. So, the program needs to only worry about the offset. It does not need to worry about the segment at all in this organization. So, it is a flag segmentation model right we have discussed this before.

(Refer Slide Time: 39:38)



So, I have moved into segmentation mode into 32-bit mode, but there is one more thing that the x86 architecture requires you to do which is to say that you know switch. So, right now what happened was I changed the control register 0 to say that I want to execute in 32 bit mode and then I executed this instruction called l jump right long jump we have seen this before, segment id and offset right.

Recall what the l l jump does it just loads CS with this value the value of SEG_KCODE is 1 alright. So, this is equal to 1. So, it just loads the first descriptor into CS; segment descriptor number 1 into CS and what is start 32? And, start 32 is just here alright a start 32 is the value or the address of this particular instruction whatever comes after start 32 alright.

So, it basically saying jump so, it is by using l jump it is basically causing CS to get overwritten; right now, CS was 0. Now, you are overwriting CS with 1; 1 shifted by 3

bits right. So, that way you basically are looking at the first segment descriptor and now you are actually executing through the gdt ok. And, start32 is the address of this. Once again because you know everything was 0, so, I did not have to worry about you know start 32 is an offset etcetera start 32 is just an address and offset is equal to address so no problem right.

So, now, at this point I have loaded the code segment and I have loaded the eip and I have moved into 32-bit mode. So, the moment I loaded the new code segment I basically have declared that I am executing in 32 bit mode and that is why this particular directive your code 32 is telling the hardware that telling the assembler that interpret all the next instructions the 32 bit instructions ok.

And, now the first few things that you do is that you load all the segment registers with k data k data is you know 2 let us say so, 2. So, you load segment number descriptor number 2 into ds, es, ss all the other segments right.

Student: Sir, why are the bits should be in the 3.

Recall that the segment selector. So, the segment register had the last 3 bits reserve for something else the top you know the segment selector itself was living only in the last in the top sort of 13 bits. So, that is why we are shifting it by 3 right. Also, recall that the last 2 bits of the CS register were meant to indicate the privilege level. In this case when I am shifting it by 3 the last 2 bits are.

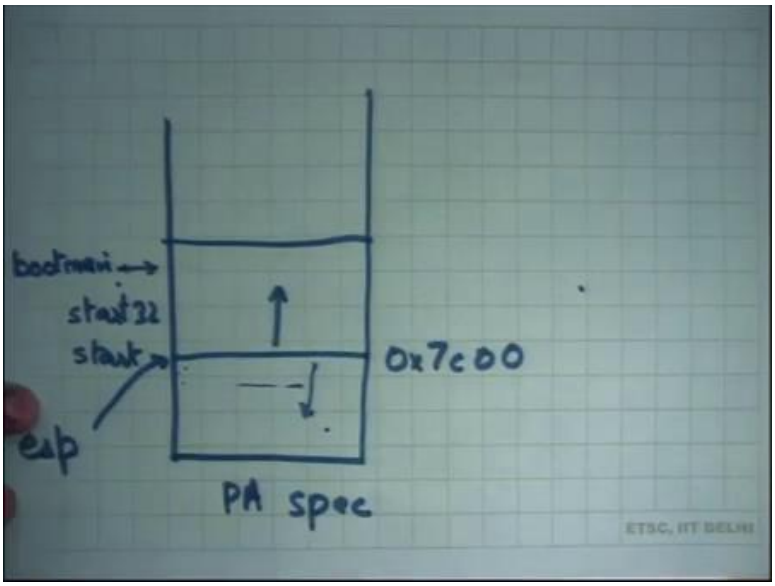
Student: 0.

0 and so, I am still executing in privilege level alright. So, I load up all the segment registers just like before, but this time is a different value SEG_KDATA which is shifted left by 3 alright and then what I do is I move the dollar start to esp. What am I doing here? Let us see. What is dollar start? Dollar start was the address of this place which was.

Student: 0.

0x7c00 right? We said that you know that is where the code is going to start and so, the linker has organized it in such a way that the address of this place is 0x7c00 that is why we started at that place alright.

(Refer Slide Time: 43:35)



So, the value of dollar start is basically 0x7c00 and then putting it in esp. What I am really doing is I am initializing my stack alright. So, what happened was let us say this is my PA space and I said this is 0x7c00 all this code that we are looking at is actually living in this area right. So, this is start and let us say this is somewhere here to start 32 right and up to 512 bytes ok. So, all this is living in this area.

And, what I did right now was I set my esp register to point to this location and recall that the esp grows downwards. So, all these areas here can be used as stack now right. So, when I am going to make a function call the stack frame is going to get pushed somewhere here in the lower area ok. So, that is what I am doing here. When I say move, dollar start dollar percentage esp I am putting the value 7c00 into esp and then the next thing I do is I call a make a function call alright.

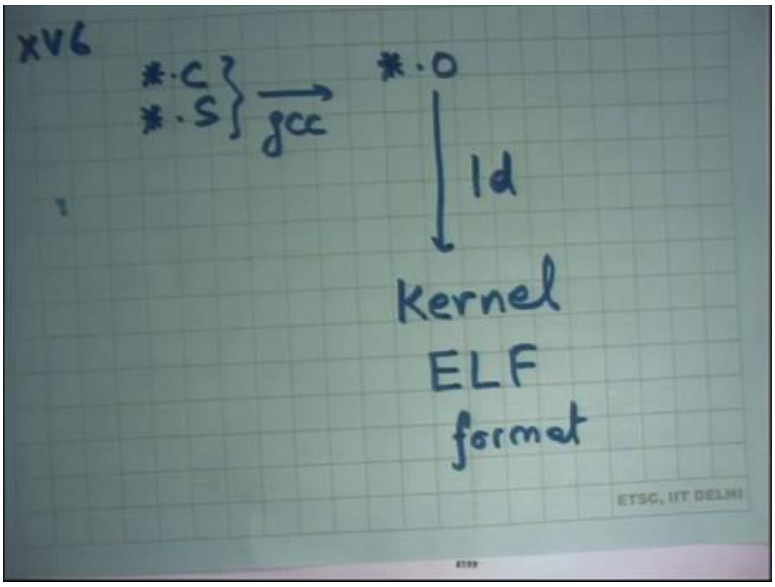
So, make a function call to boot main and so, the return address gets stored at 0x7c00 alright; actually, 0x7c00 minus 4 that is why I written address will get stored and esp will get decremented by 4 right ok. Where is boot main living? Where did this boot main come from? Boot main is so, we have at you know we have an we were executed the first few instruction is assembly. Programming in assembly is difficult. So, let us write the rest of our code in C alright.

So, we are going to. So, the boot sector itself the rest of the boot sector is living is has been written in C. So, boot main is also living in the boot sector somewhere. So,

somewhere here in this space itself there is boot main, alright and then linker appropriately arranged for it we are in that address and the code for boot main is in the next sheet 85 and now, we see some C code and that is you know more familiar and easier to sort of understand.

So, what this function is going to do is it is going to load the kernel from the disk into memory. So, you know 512 bytes is too small. So, I want to load the other bytes of the kernel into memory and jump to the first instruction of the kernel, that is what this function is going to do right and the compiled code of this function is living in the boot sector alright. And, so let us understand first how the kernel is organized.

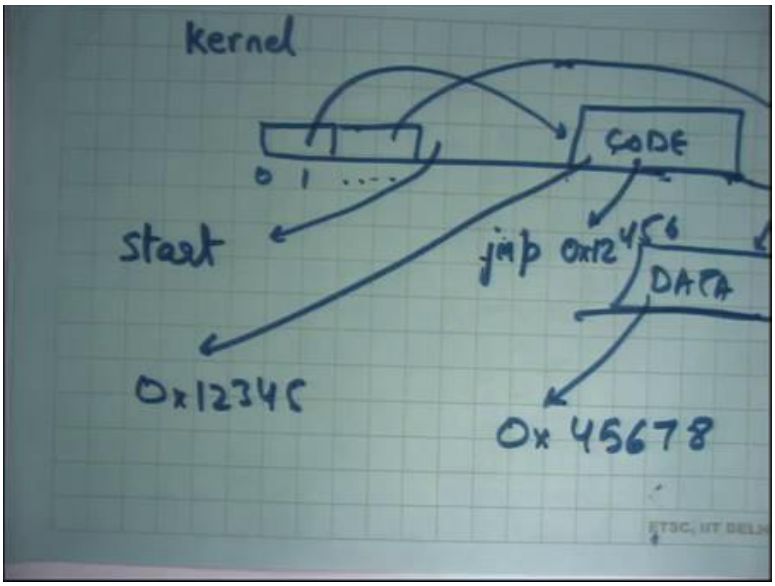
(Refer Slide Time: 46:53)



So, what happened was you had some dot C files and dot S files in your xv6 alright. You converted them to start dot O files and then you let us say g through gcc and then you linked all these dot O files and got some image which is called the kernel right. This kernel is an executable file in a format called ELF even your programs a dot O files etcetera they are also you know in this format called ELF.

And, this there is a there is standard format which is specification that you know this is the format of an executable file.

(Refer Slide Time: 47:49)



And, what the executable file will have is basically it will have things like if I look inside kernel it will say, by the way you know hey this is my file and you know this is byte number 0, byte number 1 and so on. And, you will say byte number 0 has to have certain header which will indicate where my code is. So, there will be some pointer here which is here is all your code right here is where it starts and here is where it ends so, all the size and all that.

And, then it will say you know let us say here is all your data in the file and so on right. It will also say that this code should be loaded at what address. So, it may say load code at address you know 12345 right and load data at address you know whatever 45678 right. So, it has all this information. It says, here is your code, here is the data, load code at this address, load data at this address alright. These are virtual addresses right I mean this is this basically whatever address space I am currently executing in.

So, if I am executing a program, I will look at the clf file and the process has some address space and so, ELF file will tell me in this address. So, let us the process has an address. So, from 0 to 2 GB the ELF file will say load the code at address 5 MB and load the data at address 7 MB alright and that is it and here are the first instructions.

So, the other thing it has is start, where to start? So, you have loaded the code and you have loaded the data in the into the address space placed them there and now you want to know where I should start. So, what is the first instruction I should execute? In other

words, what should I initialize eib 2 right and so, that is also stored in this file basically says after you have loaded it set eib to this and you are good to go after that you just execute whatever you like and you will probably execute some instructions, you will make some system calls whatever you do right.

Similarly, in this case in the kernel case it is not really an it is not the kernel is not going to be loaded inside the process, but it is going to be loaded by the boot loader into physical address space initially right ok. Also, then you say that the code should be loaded at a certain address the code internally could have pointer to itself right.

So, example inside it I could say you know jump to some address 12456 right. So, 12456 should be meaningful right because I know that I am going to be loaded at 12345. So, 45 I know that what instruction is going to live at 456 and so, internally I could say you know jump to in 456 so, I know what instruction is going to get executed in that time. For example, you know functions are named by addresses, variables are named by addresses right.

So, a variable will live in the data section, a global variable will live in the data section right. A function will live in the code section and these things will have internally pointers to each other right. Code some instructions will have pointers to some functions right or some instructions will have point some data, but these are all at fixed addresses because the ELF has dictated that this particular data should be loaded at this address and you have already pre-computed the address of the variable inside the data section you require right.

So, similarly what is going to happen is that this kernel will say that this is my code and this is my data and we will say load this code at this address and load this data at this address and so on. And, what my boot loader is going to do is respect whatever the ELF file is telling in and put it at the right address. And, now and also the ELF file will contain the start address and based on that it is going to jump to the kernel right.

And, so the kernel has so, the boot sector job is over, now it is the kernel which takes over after that. So, the boot record loads the kernel and jump to its first instruction and how it does all this is basically dictated by the ELF file alright.

Let us stop here and we are going to discuss more code next time. So, highly recommend that you familiarize yourselves with xv6 and all that and we are going to do more of this example next class.