**Bootloader**

so hello everyone um today we'll get introduced to the bootloader code in xv6 so this is the first time we are going to start reading the code of the xv6 operating system and we'll begin with bootloader because that is where everything begins the operating system starts loading with the bootloader we are not going to understand the bootloader code completely 100 because we are yet to understand and discuss some basic so the bootloader code involves certain code about memory management i will touch upon that code i will discuss few things about that code but it is not my intention to convey everything about the memory management part of the bootloader code we will revisit this part after we are done discussing some basic concepts about memory management so we'll begin reading the bootloader code so first of all let us try to understand what does a bootloader do once again so we all know that the bootloader code itself will be loaded by the bios okay now where will that code get loaded in memory that will get loaded at a fixed location why fixed location because remember the bios code is written by manufacturers the bias code itself is located at the fixed location the bios code will look up for the bootloader code on the boot device in sector 0 and now when it has to copy paste that code in memory that also has to be a fixed location these doc these locations are normally documented in the manuals given by your cpu manufacturers so what will the buyers do it will look up the sector 0 of the boot device copy the contents of that sector into a particular location in memory and then bios will make it run so now our job as os programmers is to write the bootloader code so the code that we are going to read now is this code so we are assuming that this code whichever code we are going to read is already there in memory it has been passed control by the bios and that particular code is running now let us understand what is the bootloader going to do so as you understand the job of the bootloader is to load the os in memory so what will the bootloader do it will pick up the code of os from a known location most typically on the disk so the bootloader is supposed to know where on the disk is the code of the os kernel and then it is supposed to load it in memory and then make the os run now the code of the xc6 bootloader is in these two files boot asm dot s and boot main dot c let us see the make file in order to understand this point so here i am on xv6 code and i am opening the make file and i am searching for so you are not audible it's not in terms of bytes typically so here we are saying read from the dave zero device yes yes so you weren't audible for a period okay so when what is the last thing you heard from me you said you were going to check for the bootloader code in the make file all right so this is the boot this is the make file and this is where we can see that the kernel is getting built this particular line specifies how xv6 is built that is the disk image of x36 is built we can see the use of the dd command what dd command does simply is it is a copy command with if specifying input file over specifying the output file so what this command means here is that read from the slash dev slash zero file which is an infinite source of zeros right into the exo6 dot img file how many blocks 10 000 blocks so basically you are creating a big file containing of all zeros and the size of that file will ten thousand block where block is typically five and two bytes once again dd command read from the boot block file right into the same file so overwriting is happening without truncating so the size of this file will not change so only whatever is there in the boot block will be written as the initial part of the x26 dot img file and then you are reading from the kernel file writing into the same file skipping one block so first block is kept so the writing of this file into this file will start from block number one that is after 512 bytes without truncation basically what is happening this file is nothing but a concatenation of boot block followed by the kernel followed by lot of zeros okay total size being ten thousand blocks so we have to now understand how the boot block is built because that is the first block that is the first 512 bytes right so boot block as we can see here is built from boot asm dot test which is the assembly code file and boot main dot c so how is the boot block created we can see that the bootmain dot c is compiled using cc with a minus c as you can see here so this will give us a boot main dot o then the boot asm dot c is also compiled using gcc because gcc can invoke only and the minus c it will create a boot asm dot o then the linker is invoked and you can see the input to the linker is now the uh boot asm.o and the bootmain.o and a file called boot block.o right so these three files are given input to the linker ah sorry these two files are given as input to the linker and boot block.o is the output of the linker because there is a minus o here so basically the linker will combine these two object code files and create the boot block dot o file the interesting input to the linker here is the minus t text hexadecimal 7 c 0 0. so this is telling the linker that you create object code file but in the object code file you write this that when this code gets loaded in memory the code should get loaded at this address particular address now by this address obviously this is the address documented by your processor manufacturer telling that we will entertain the bootloader code at this particular address 7c 0 0 that is why the linker has put in this information in the object file in memory it should be loaded at this particular address so this is how the boot block dot o is compiled okay and here simply you can see that is a obj copy command invoked which copies boot block dot o as boot block and that is a boot block that is used here so the boot block is essentially the boot block dot o i'll skip this line so what we are supposed to do now we are supposed to understand the code in these two files because these two files will lead us to the boot block okay and we will begin with boot asm dot s so we begin with boot asm dot s so this is the boot asm dot test this is the boot loader code right and the first instruction that it runs is cli please understand that whenever you see directives which begin with a dot for example this dot code 16 or dot global start these are basically additional instructions to the assembler to assemble that is convert to machine code particular way the act the first actual instruction to be assembled is this cli okay rest of them this first two lines are obviously additional directives to the assembler the start is just a label okay this will be used as a location by the assembler you know whenever let us say there is a jump instruction jump to start then this jump will refer to this start will refer to that particular location the first instruction is actually cli the the bootloader begins in the cli code now before we read some further code i will switch back to my slides so we in summary just now discuss that a bootloader is already loaded by bios in memory and it will start running so we are about to see that code now the code we are going to see is supposed to do this pick up the os from a known location and load it in memory now we know what is that known location we have already seen what is that known location what is that known location actually we have seen it in makefile what we saw is let me go to the boot block in main yeah what we saw is that this x36 dot img is a concatenation of boot block and kernel right the boot block code that is the first sector from the disk is already loaded in memory now where is the kernel as we know the kernel is basically just after the boot block on the same hard disk so basically you skip one block okay on the hard disk and that is where the kernel is okay so this kernel as we have already seen is basically the you know in a combination of lot of object code files consisting of the kernel code so where is it located as we know here it is just you know after the boot block on the xv6 dot img file that is the hard disk so all we have to do in the boot block code is go to the sector 1 of the disk and read it because who is there in sector 0 of the address the boot block itself which already has been loaded in memory so all that this boot block code needs to do is go to sector 1 and read the kernel code from there all right so once again in summary the bios code has already executed automatically through a hardware mechanism it has already loaded now the boot sector at address 7c 0 0. it has already done that okay when the bios code loaded the boot sector boot sector was this boot block file this boot block file was the boot block dot o right this boot block dot o was basically this boot block file and this boot block file was basically copied into sector 0 of the hard disk by this particular line and this has already been loaded by the bios in memory and it is executing right now so the bios is intelligent enough to read this code that is this code and it is knowing that it was to be loaded at seven c zero zero so the boot loader code is already in the ram at the address seven c zero zero and it will now start executing this particular code so the way boot loader is ensured to be at 700 is with this particular line that we have seen in the make file okay any questions so far before i proceed further oh there is already something in the chat okay that that was about me not being audible so i'll wait okay if you have some questions i'll take them before i proceed further so the boot block overwritten in the ram actual ram of our computer because the os which is already present like our linux thing that is also present at the same position right no it is not the os is not in picture right now the os is right now on the hard disk not in ram so no no like when it loads then ros the laptop which is running currently which will run that xv6 but the xv6 will will run in a virtual environment created by kmu yes so why are we writing it exactly at that place then because when the kmu creates a virtual environment it is independent of your host laptop for whatever code that is running inside chemo that is the bias code or the bootloader code or the os code they are running as if you know it was an independent piece of computer the virtual machine that is created by camu is a proper virtual machine you know it is a proper hardware and it is independent of your host operating system so when virtualbox creates uh sector 0 on the hard disk that is not the actual sector 0 of your hard disk it is a virtual sector 0 but then for the code which runs on top of it it looks like actual hardware got it so i think you are confusing the execution of a guest machine with the execution of a host machine we are running xp6 inside a guest machine right so the kmu software is going to create virtual processor virtual ram the virtual disk is already x26.img file virtual motherboard virtual bios it is going to create all of that it is going to execute all of that in a virtual hardware environment the virtual hardware environment is a as good as independent environment compared to your host operating system so that is not going to affect your host at all in fact it will not know anything about the host it will only know about the environment created by kemu what is it so did i answer your questions yes sir yes sir yeah so just to make sure the way you run it is this you know you say make camo which is basically invoking the camo target in make file so essentially it was running this command as you can see kmo system i386 so this is the basically 32-bit version of kmu and that is why kmu here is creating by default 32-bit environment okay not a 64-bit environment because the file that you are invoking is camo system i386 i386 as we know refers to 32-bit version so the program which you are running itself you know is a code compiled to create a 32-bit virtual environment that is why xv6 code also runs in a 32-bit environment not in a 64-bit environment this could be a nice assignment for all of you to actually make suitable changes to this code so that it runs in 64-bit environment but that is not our point of discussion right now but for those of you who are adventurous enough you can try to make changes to the make file and if necessary to the actual code so that it runs in a 64-bit environment so coming back what i'm doing is i'm invoking came with the moment i invoke camo a virtual processor virtual disk a virtual memory virtual motherboard have been created with a virtual bios bios which loads the boot loader code from this file x26 dot if you remember its argument to kmo so chemo will treat this as the hard disk and that is why the bios code inside kmu will read from this hard disk the sector 0 of this hard disk what is there in sector 0 of this hard disk the bootloader code who put the bootloader code bootloader code there when we did make you know the x26.imd was created with the first part of the file as the bootloader code so this is how you know the whole thing works right now we are at the stage when the bootloader has been loaded and the bootloader is about to execute more questions the location c00 7c000 is the location of the os right no it is a location where the bootloader has been loaded right now it is a location where the boot loader has been loaded the os is yet to come now the code which you are going to read is going to read the os code from the hard disk and put it into ram and then pass the control to the os code sir we are copying that w0 file into xv6 dot image then we are again overwriting it so why do we do that so that is basically part of the make file right so that is done as a part of creating compiling xp6 code and so on so when you compile xa6 code obviously what are you going to do you are going to compile each kernel code files independently so let me just do this i will revise some more concepts once again so i am going to close this session ok i'll say make clean okay so i have removed all the temporary files and i'll run make again so all i did i just said maker i did not do a make k mo if you can recollect so now you will see here okay what happened at the part of the execution of the make file you will see that you know the gcc is compiling uart dot c into iit.o vm.c into vm.o and so on finally you will notice that this is the command which executed to link the boot uh other dot or to any other dot to and then let us find the other commands that we are interested in yeah here you will see um yeah this dd you know which executed so this dd executed on my host just basically to create a file like this so what what this command did is it basically created a large file okay and the size of that file was 10 000 blocks of 512 bytes each okay no sorry not this command this command so basically you first created a large file and into that large file you copied the boot block first if your question is why i created a large file first i'll say just to ensure that i get that much space because if this this command fails then other commands will also fail later on so basically i'm ensuring that i have a 10 000 into 512 bytes file already created and into that file i am going to copy this file and then this file that is my answer like what all i am doing is i am ensuring that i have a large file ready does it satisfy you okay sir but sir in that dev zero file there is something written in there over there no that is a magic file we we will not see that magic right now we will skip that discussion towards the end of the course so it's a magic device file on linux systems which is the infinite source of xero you open this file and keep reading from it the file will never end okay and it will keep giving you zeros do you read any amount of data from it you will get all those zeros so it's a magic device file on linux okay sir so all the files in slash dev okay on linux they are called device files so all these files this zero file vcsu file tty file they are all device files so they are basically used to access a hardware device as if it was a file so for example this file dev sda this is your hard disk you can actually write a c code which does open slash dev sda and it will actually give you access to the raw hard disk from white number zero onwards so if you really want to read the sector 0 of your actual hard disk just write a c code which opens this file and yeah you are reading the sector 0 directly so that's a you know magic of all unix and linux systems they make hardware devices also available as if they were device files but you know that is not part of our discussion today we'll see it sometimes later when we study file systems as of now is it clear that we are now how we are going to run the bootloader code the exact sequence of things which happened after i started chemo so let me run make camo the moment you know i said make cameo it is compiling once again but the moment this command started running it actually access this hard disk file and the bios code which is part of this particular kmu code itself will actually access the sector 0 of this hard disk file load it in memory and pass the control to it and that is where we are right now any more questions okay there is a question here permission would be denied right for tampering with dave hdr would be dangerous so only root can do that only root can do that normal users cannot do that so if at all you write a code you have to run the code as a sudo otherwise permission will be denied again that is you know sideways discussion that is not part of our main discussion today so let's start reading the code of bootloader now i will touch upon some part of memory management because otherwise you know it's not even possible to discuss the boot loader code because he you are about to load the kernel and the moment you have the kernel the first thing the kernel also has to do is do some memory management for itself because the kernel also needs memory for itself so the initial code of the kernel and the initial code of the bootloader also have to do some memory management because it is so integral to the execution of even the first instruction that you have to do some memory management now you will notice that the first instruction is cli which is going to disable interrupts so what we we desire right now when we are running the bootloader code is that until i am done you know with a significant part of my my function i don't want any interrupts i don't want to handle any interrupts so basically disabling of the interrupts okay no interrupts will occur now you press keys and you press you move the mouse around and whatever happens the bootloader code will keep running okay ignoring all interrupts that are going to occur so keep this in mind with when we are reading all the code okay that this code is not going to be interrupted at all because we cannot forget this very simple fact that the moment the interrupt occurs the cpu will stop running the code it is running and jump to a particular predefined location that is what the cpu normally does so you don't want that to happen because you are in the very very initial early stages the os is not in picture nothing is in picture so simple thing disable interrupts okay we don't enter in interrupts right now now as i mentioned yesterday the processor will start as if you know it was in a 16-bit mode so right now we are in the 16-bit mode of the processor and that is why all the resistors will be 16-bit general purpose registers and the addressing you know using segment registers code segment data segment extra segment stack segment will be done using the formula that you know address is segmented to four plus address this we have seen yesterday in the 16-bit mode that is how the addresses are going to be calculated so this is a diagram which you may not understand completely right now we just have to understand the simple thing right now cpu is running a particular instruction after that instruction is over cp is going to send on the on the outside bus you know which goes to the memory and address the address from where the index instruction has to be fetched so basically who specifies the address of instruction the instruction pointer right ip but we have seen that whenever the instruction pointer is used the effective address that reaches the memory management unit through the memory management unit of the processor is calculated using the formula i mentioned just now segment into 4 plus the offset the offset is given by the instruction pointer but who is the segment in concern for the instruction pointer the code segment so these two boxes are actually part of the cpu itself so the cpu is actually this whole thing okay and these two parts shown here are called the memory management unit of the cpu the processor itself this part is outside the cpu chip and it goes to the ram so some kind of calculations are happening here the so called segment translation and sorry the so-called segment translation and the page translation which we are going to skip right now so some calculation will happen the term physical address is used to denote the address that is actually presented to the ram okay now the xps the x86 processor family actually does two type of memory management one is called segmentation the other is called paging it is possible for the programmers that is operating system programmers to actually enable either segmentation or paging or both together okay effectively when you say i am enabling only one you are configuring the other system you know as if it was non-effective how is that exactly done we are going to skip that right now we'll just go with the fact that there is something called segmentation and something called paging in the hardware in the hardware which is to be configured by the operating system code and that is why it is also to be configured by the bootloader because otherwise you cannot execute any code the memory management has to be set up before any code practically executes so what is the first line first few lines of code the first line of code is cli as we have seen and it will disable all the interrupts so while executing rest of the code no interrupts are going to happen now please understand when you are running this particular code what was there in the instruction pointer the instruction pointer had the value 7 c 0 0 and this particular location itself is 7 c 0. in memory so the instruction cli is at 7c 0 0. so obviously this next instruction is at the address 7c 0 0 plus 4. so obviously at this particular address 7c 0 0 plus 4 this instruction is that 7c 0 0 plus 8 and so on so the instruction pointer remember is automatically going to be incremented okay how that's how the processors work the instruction pointer automatically increments itself after running every instruction so this will execute in sequence now okay this will execute in this sequence now what is happening in the next instruction xor ax ax remember ax is the 16 16-bit register because we are right now running in the 16-bit mode of the processor so if you do xor of resistor with itself basically it will zero the resistor and then what are you doing you are copying ax into ds es and ss so all these three segment registers will carry the value 0 now you are basically going to set up the value 0 in all the 3 registers what about the code segment what about the code segment the code segment must have been initialized by the bios to you know have a particular value isn't it and that is why the code segment is not being touched here because the code segment refers to the code you know the location of this particular code we are not going to change that so you will notice that all these three segments are first zeroed then there is a weird piece of code which we can actually skip right now let me go to the terminal okay yeah and boot so we are going to see boot asm.s so we have seen this cli line of code we have seen these four lines of code okay they have zeroed the segment registers we are talking about these two pieces of code now we can actually safely skip them right now they are a weird bit of you know weird piece of code which actually you know exist in all boot loaders for some historical legacy purposes so let me describe you in short what happened um we know that the segment colon offset is a 16-bit calculation segment is the 16-bit segment itself ah with that you could address 20 bits of memory because you are multiplying segment by 4 and then adding it to the offset so 20 bit of calculation so that means you have 2 to the power 20 that is one mega point of memory addressable now it so happened that when the 80286 processor came they said okay we want one more bit of memory so 21st bit of memory okay and now whatever earlier code that was written on 8086 was assuming 20 bit so because these people always wanted backward compatibility they ensured that the bios disabled the 21st bit but now some but new operating system which will use eight zero two eight six they said oh we want the 21st bit so that we get two mb of memory addressable so now what was needed was you know to enable the 21st bit so all this code is to enable the 21st bit but remember you are running in the older mode that is the 16-bit mode where the 21st bit is disabled now that 21st bit in earlier 80868088 microprocessor was actually used for the keyboard controller okay and that is why there is this whole mention here as you can see in this code in b dollar 64. so this 64 is a port number which is the keyboard controller port you are reading that port right now here in is a i o instruction from io port so all this code is basically to enable the 21st bit and that line is called a20 starting with 0 820 so that is the 21st bit and that is being enabled here we can skip this part of the code right now so we are going to skip this code okay this particular code is referring to enabling the 21st bit so right now we can say that the 21st bit is you know enabled here so when i am here the 21st bit is enabled and 2 mb of memory is addressable so the eip and cs pair could actually address now 21 bits the interesting part is this code now you can see there is an instruction called lgdt which basically says load the gdt and then our argumentary gdt descriptor so this instruction lgdt is used to load what is called as the global descriptor table which is basically a part of the memory management unit so you are now going to load something into the gdt table what you are going to load the data given by this unity descriptor what is the gdt descriptor just go slightly down here in this file and you will find that there is this gdt descriptor which is basically referring to the gdt as you can see the dot long gdt the dot long agility is nothing but this and what is this it is a sequence of three numbers these are three numbers actually what are these three numbers once again what is this signal asm i am using let me tell you c scope and c tags integration with vi to jump to this signal asm you will see it is all zeros here okay and go back what is this seg asm this seg asm is a macro as you can see which takes a type base and limit and does some calculations on it so basically it is passed on these arguments and once again say gaussian is passed on these arguments so it is basically creating a number out of this argument so this is basically is basically a set of three numbers so what are we doing here we are saying load into the gdt the set of three numbers given by the gdp descriptor so coming back to the slides now okay coming back to the slides with this particular code we are going to load these three numbers into the jdt that is what we are going to do and after that there are three lines of code which is this crunch of code what are they doing read it carefully the cr0 resistor is moved to ex then eax is odd with cr0p what is now this cr0 p it is a number let me jump to that the cr0 p is this number okay so a particular bit will be enabled in the eax and that will be moved back to cr0 and then you are going to call an instruction called ljump so as a combination of these four the mode changes to 32-bit mode from 16-bit the processor changes to 32-bit protected mode from real mode to protected mode i must mention though the rear mode and protected mode are totally unrelated to the user mode and kernel mode okay the user mode and kernel mode will exist in the protected mode now okay that is the user mode in which certain instructions cannot be executed and there is a kernel mode in which all the instructions are going to be executed so the real mode protected mode are not related to user mode kernel mode so this particular flag which is set in eax now uh and then moved to cr0 so basically a particular flag is set in cr0 after running ljump instruction the processor will list to the 32-bit mode before you jump to 32-bit mode you have to set the memory management infrastructure that is being done here to put it in simple word so first you set up the memory management table that is the gdt table and then you enable the 32-bit mode after this we are into the 32-bit mode now what changes in the 32-bit mode in the real mode 16-bit mode all resistors are 16-bits in protected mode you can now you know access 32-bit resistor so you can say eax now if you say ax it that will also work but earlier you could not see ex now you can say eax to refer to the 32-bit version of the resistor you can interestingly now access 32 to register power 32 that is 4gb of memory huge amount of memory because you are in 32-bit mode and arithmetic can be done in 32 bits you can say add long kind of instruction which will add two 32-bit registers and the interesting thing that happens now is that the calculation of address which was happening using segment into four plus offset that is no longer going to work the value in the segment register will be used now as an index into something called as the segment descriptor table so let us try to understand this so there is this kind of a table okay it is called the gdt table and there is also another table called ldt but right now we are using only the gdt table the gdt table is an array it's an array of base limit and flags the way memory addresses will be calculated by the processor now is like this the value in the segment register will be used as an index into this table from there it will take the base value and the base value will be added to the offset to finally calculate the address that goes to the ram so now what is actually stored here you know in the gdt what we stored in the gdt is these three values right when we said lgdt gdt descriptor we actually stored these three numbers now if you look at these three numbers you know and the value the definition of the say a7 macro the sega asm macro is type base limit right three arguments to that so you will notice that the base is 0 and the limit is 4 gb the base is 0 and the limit is 4 gb so what is going to be loaded in the gdt table are three values right now okay where the base is 0 and the limit is 4 gb so if you add 0 to an offset here because the addition is happening of 0 you will basically get the offset itself as the final address so we are doing a memory management setup where in every address issued by the cpu when it goes to the memory management unit of the cpu will basically remain the same because the segments are ineffective because of the way the gdt has been set up now if you did not understand it completely that is fine we will revisit this when we discuss memory management so essentially what has happened here after the lgdt instruction is the gdt has been set up to actually have the base at zero the two limits as 4 gb the first is 0 because you know the first number was all 0 and then the third value was permissions so for permission here is right permission is read execute we will even skip that okay that is what has been done now and the calculation has been done will be now done because the base will be get will be added to the offset that is the value of eip or whatever register you will effectively get the same number as the physical address okay so now with these three lines which we already seen you know you load the cr0 register with the flag set to one and then you call the long jump now after this long jump instruction is executed essentially it will reload the cs and eip with 32-bit values but you will you will now be in the 32-bit world i am sure you have not understood this part completely i am fine with that because we will revisit this part of the memory management code when we see memory management once again now the interesting thing you can see what is happening here you are copying say k data shifted to 3 to ax and then copying ax2 esds and ss okay so let us go to the code once again ok so you will see here you are in 32-bit mode now the dot core 32 tells the assembler to generate 32-bit code for all this part so what are you doing you are copying secure data left shifted to 3 to ax what is segk data let us jump to the second data you will see it is 2 second is 2. so 2 left shifted by 3 so we are storing a particular value in ax and copying that value into dsess so as a result of that what is going to happen is the cs ssds will basically point to this value 2 that's all going to happen they will have this value 2 and remember the value in these is used as an index into the gdt so that is what is happening here then you are copying 0 to ax and these two these two segment resistors which we are effectively not using they are zeroed right now then you are moving dollar start to esp what is dollar start dollar start is this address what is this address it is 7 c 0 0 because this is where the boot loader started so this is the address 7 c 0 0 so here you are copying 7 c 0 0 into the stack pointer and calling boot main so the stack pointer now is pointing to the address 7c 0 0. you will say oh it is the same address where the code is located but the interesting thing is code grows from 7 0 0 upwards while the stack is going to go downwards so the 7c 0 0 address is the boundary between the code and the stack stack will grow downwards and the code is already there the bootloader code is already there upwards from 7c 0 0 and now you are going to call boot menu remember it is a call instruction so what it will do it will push the written address onto the stack it's a call instruction so it will push the written address that is this particular address address of whatever is here it will push this address onto the stack and jump to boot main so now let us go to boot main and if i go to boot main you will see it is in bootman.c alright so from assembly code now we are into c code but remember when i compile these two programs they were part of the same dot to file let me once again show you boot block let us go to lock so you will notice that boot main dot c was compiled into boot main dot oh boot acm dot test was compiled to boot acm dot o and these two together were compa linked into the boot block. so as far as the final boot block is concerned that is this boot block it contains the code of both the assembly code formed by the c code okay they are both concatenated together into this so let us go to boot block boot main dot c boot main so now we are going to run this code boot main i'll wait for few questions before i read this code okay please be more patient if you could not understand the memory management part of it right now my focus is on not memory management i just want to show you how the kernel is going to be loaded okay and a particular sequence of code execution okay let me proceed sir yes every operating system starts to run at the same address in the ram on x86 32-bit yes not the os but the boot loader okay 32-bit x86 yes the bootloader will be loaded by the bios at 7 0 0 and the code will execute now so even now we are running the bootloader code now we have not we have not even you know gone to the os right so far right now we are running the bootloader code so let's see what the bootloader code is going to do now remember this is c code which was compiled into object code and put it into the bootloader file you will see there is an interesting piece of code here there is a pointer called the alf pointer which is pointing which is the pointer of the type elf header now why do we have this a lifting now here that is because when i compiled my xv6 i created a file called kernel right and this kernel file is a elf file because elf is the default format and i have created ela file so the bootloader will have to read the elf file the kernel file itself with the elf file so there is the elf.h you will see there is a elf.h which basically gives you structures to access the elf file format and alf header is a one particular structure what is being done here let me show you the essential part of the code so that you understand what is happening and then we'll again revisit the minor details if you notice the alf pointer is pointing to one one mega this is one mega okay it is pointing to one mega location now then you are calling read segment function to which you are passing elf as the target address and you are saying read from 4096 okay on the disk read and then let's see the read segment read segment first is the address in memory second is a count and third is offset so what you are saying read from actually byte number zero how much four zero nine six bytes let us see what read segment does you will notice that the read segment is doing some calculations here pa plus count equal to epa and then some modular arithmetic but what you will see is it is doing is it is doing a loop in terms of sector size where sector size is 5 and 2 because see the disk is always accessible in 512 unit size this is a very important fact all hard disk all pen drives all cds they are all accessible in a unit of 512 you cannot access a particular byte on the disk you have to always accept access a particular block on the disk so block size is sector size that is 512 and it is calling the read sector in a loop so in order to read these many bytes it is splitting those into 512 sizes and calling read sector let us go to read sector so right now we are in boot main dot c but the read sector is also here in bootmain.c what is it doing it is calling a function called weight disk and then instructions like out b now what is out b let us go to out b you will see out b is a c function which basically has assembly code called out so in and out as we know our i o instructions they are used to write instructions into particular locations for the hardware controllers so they basically act as a bridge now between the software and the hardware world when you run the in or out instruction you are basically writing into the hardware controller and accessing the hardware device and obviously these are privileged instructions and by default we are running in the kernel mode right now when you switch from 16-bit mode to 32-bit mode you are by default in the kernel mode so you are in kernel mode and you are running out instruction all together what are these doing they are actually instructing the hardware controller to read from the disk and make available the data at a particular memory location okay so at the end of all these i o instructions what will happen is that the data will be read by the hard disk controller from the physical hard disk and put into memory so at the end of this repeated calls to read sector you have read the 4096 bytes from byte number 0 of the hard disk and put into the address 1 mb why 1 mb because here we said elf is 1 mb so the 1 mb location in memory will now contain extra 4 will contain the 4k data from the first 4k bytes of the hard disk okay and what we know we know that out of the 4k the first 512 were the bootloader and then the next code was the kernel so you will now notice that the alf pointer in that you are now going to access the particular location called alf plug ph off and then you are again calling read segment to read further parts of the kernel into memory okay few more things and then essentially it is calling a function called entry this entry okay let me jump to the entry now uh okay is already a location which was set by the linker while compiling the code so the moment you jump into this location the entry location it will jump into the operating system so let me explain few things from the slide the bootman.c here will expect to find a copy of the kernel on the disk starting at the sector 1. why sector 1 because sector 0 is bootloader code the kernel is in ala format the boot main will load the first 4k bytes of the alf binary and place them at the address 1 million okay this is the code we have seen now boot me what does the boot main do because you know the elf is pointing to the one mega location and now it contains the you know the code copied from the disk it will first check for a magic number so the kernel code is supposed to carry a magic number in first few bytes if that is not true then you will basically conclude that you know the kernel was not proper and then you will return return where in boot asm.s because you did a call so you can always do a return few things about the elf file format now obviously these are very few things about the ela file format in any ela file now you you should know that the executable file of ls is also alia file when you compile your x dot c as x dot o that is also an ela file the ellipse file is also an alf file and this kernel that we have compiled is also an elf file all of these files what what do they have in the beginning they have a lf header and that alf header is this structure now i am sure some of you must feel excited enough to write a c code which opens the alia file and starts reading please do it please do it start reading the binary files like ls and this kernel and a dotto file you will find a you know a structure of this type in the beginning where the first thing is the magic number some of the interesting things there are you know a number called entry a number called ph off okay uh which is basically the program header offset into the file and then there is a ph number which is basically number of program headers so there is something called as a program header which comes after the elf header okay and then in the end there is the section header essentially both of them will contain offsets into the same file although this looks like pointer they are actually offsets in the same file which give you you know locations of different sections of the lfi now in the alf file there are different sections there's a section for your code is the section for your global data there's a section for read-only data and so on and the program header will give you the location of those the lf header will give you the location of the program header what is there in the program header imported things are physical address and size so the program header will also tell you how big is the text and where should the text go in memory okay so file size will give you how big is the text and where should it go in memory so if you run this command obj dump on the kernel because ob gynum handles ela files and first see the first 15 lines there is a lot of output so you only see the first 15 lines you will see the program header also because object shows you program header now let me actually run this so obj dump minus x minus a kernel y plus so this is what i'm talking about so you will see this is the program header in the file and then there are sections in the file we will not go into details of what does the section mean actually but if you go to the text section down here it's a symbol table and after that is a symbol table i should have said minux d also know minus d will disassemble it otherwise it will only show the headers yeah so you will see here now in the kernel see the code of entry something called multi boot header b in it so these are all the functions of the kernel code these are all functions so this is the program header i'm talking about you can also run obj dump minus x minus a slash main slash ls and do a less you will see the ls also has a program header okay and it also has something called as dynamic section it will have section headers so these are all section headers now what do the section headers mean and all we will not go into details of that but what we see here is that when the kernel file was created you know it put in these values into the kernel file what does it say that the physical address to which this should go is 1 mb and we have taken it to 1 mb already isn't it we have taken it to 1 mb and there is some size okay actual size of this one it also has a section called stack and you can see that the stack is all zeros okay why all zeros and all we will not go into the details of that right now so what is the what is happening actually here in this pieces of code that is sorry boot main dot c so we saw this code what is happening in this piece of code in this piece of code you are actually loading each program segment okay this is a pointer into the alf file you are iterating over each program header each program header is going to give you the physical address you are going to read every segment okay at that particular memory address okay so it's a very generic code to read any lf file and load the elf file into memory so obviously when your kernel does when your kernel does exec of any particular program it will also do the same thing for exact because the executable file is a alf file so it will do a similar kind of code even when while doing exact it will read the lf file the header then the program headers then the section headers and copy each content into a particular location in memory okay after this basically you know it is jumping to the location given by the entry this entry alf arrow entry was actually set by the linker using the data given in kernel.ld and this is the address you know hexadecimal 80150 and so on so we will not go into details of this right now we will revisit this when we discuss how the kernel is compiled and linked and what is there at this address and how is the memory management done as of now what we can conclude is you know after this you know we are off this file we are not going to return back we will jump into the code of the entry function okay so when the entry function is executed entry is actually in entry dot s entry is in entry dot s so this is the code of entry yeah this is the code of entry i'll very briefly tell you what is this assembly code doing first of all it is setting up a flag in the cr4 register and that way it is doing something related to memory management then you will see it is loading something into cr3 that is also related to memory management then it is setting up some value in the cr0 that is to turn on the paging memory management then it will it is as you can see setting the stack pointer so the stack pointer you will see now is set up to a very high value like this is a higher value and then it is calling you know main because it is moving the address of main into eax and then jumping to that after this you are into the main function the main num the name main is only taken here to you know match your conventional knowledge it the name could have been anything okay there is no relationship between the word main here and the main in a c program the function name could have been anything this is the function where the os code will start running okay after you have done this okay you are now going to jump into main and that main is actually the main of the kernel okay so we'll we'll see the code of the kernel now later on okay after this entry you are after this code in entry you are going to jump to main and that main is the code of the curve so in essence okay what i want you all of you to really understand out of all this code is before you came to boot main there was some code executed we changed the mode from 16 bit to 32 bit board here actually you loaded the kernel in memory by calling the read segments which called read sector which called in and out instructions on the hardware controller and the code was in memory with this you loaded remaining part of the kernel into memory and jump to the code of main as a part of the code of entry and now the kernel is going to run okay so if you have understood this much that is sufficient right now i will request all of you to read this code read the description of all the assembly instructions and try to make sense on your own you will make it will make perfect sense only when we have studied memory management so we will be patient to understand this code completely before we study until we study memory management so basically the code from boot asm boot main is over the kernel is loaded and the kernel is now going to prepare certain things like creating init you know and so on the kernel is going to do lot of things but basically the kernel has started running now let me see if there are any questions so can you quickly just write a comment uh like how much of it did you understand like 10 percent 50 60 percent i want at least few comments like just tell me how much you have understood 30 which is good number actually is a good number 30 percent is a good number anyone else like um okay so 30 is good 30 is a good number i didn't expect anyone to say 50 percent 30 is a good number because we are yet to see memory management and large part of this code was memory management so 30 is good no problem with that all right so you please read this code once again and keep your questions pending until we study memory management i just wanted to visit a bootloader code to show you that you know how the bootloader code goes into the ram how that brings the kernel into the ram and passes on the control to the os okay so you have at least a broad notion of you know how the bootloader loads though is that is sufficient okay we'll stop now

**CONTEXT SWITCH**  
  
is being recorded okay are there any questions on uh nah the discussion we had yesterday uh i'll repeat uh do you have any questions on the discussion that we had yesterday so we started the discussion on understanding processes are there any questions on the discussion on processes that we have we had yesterday uh i'll appreciate it uh if you can actually speak because [Music] reading the chat messages is actually a boring task so it will be better if you can speak also but of course i don't insist you can write in the chat box as well can you please go over process state yes i am going to do that again obviously so i'll cover that the process state all right i'll begin the lecture today i'll cover some of the concepts we have covered yesterday once again and we will obviously cover the concept of process state more okay so we saw yesterday that uh process control block a pcb is basically a record representing the process in operating systems data structures okay now obviously the os is going to maintain a list of such process control blocks and there will be one pcb for each process i also mentioned that the name of the structure in linux kernel code for the pcb is task underscore struct and the name of the pcb structure in explicit code is proc now what are the most typical fields that we should now expect in the process control block all those fields which help us identify a process and all the resources that the process is actually using when we say resources we are referring to the files opened by the process the memory occupied the process and so on now out of these ah in the last to last class we have already discussed the array of file descriptors with respect to the open read write system calls and we discussed how open ensures that you know the lowest available file descriptor is always issued obviously every process has a unique identifying number the process id that will also be stored in the pcb so this is the process number or process id and the list of open files as shown in this diagram here the program counter and the set of registers that the process is using why do they need to be saved in the pcb remember pcb is a structure so the structure will be part of the ram obviously why do we need to store the values of cpu resistors and the program counter and program counter is also another resistor actually most of the time so why do we need to save them in the pcb because when let us say the process is running and we want to take the process out and should learn the process then whatever resistance that process was using they have to be saved because when the process gets rescheduled it should resume from where it left and that is why these things need to be saved obviously we need to save information about the memory limits of the process some accounting information the i o that the process is involving and so on and there is a there is a field called process state about which there is a question also i am obviously going to discuss it again now what operating systems do is in order to do various operations on the processes they maintain various queues or list or some kind of data structures which combine lot of process together now obviously for example if you want to do a round robin scheduling that is scheduling processes one after another and let us say giving equal time quantum to each of them maybe you can just maintain a queue of the processes so you can think of a queue of processes that the queue of pcbs actually processes which need to be scheduled right and then there may be other type of queues for example you know queue of processors waiting for keyboard input and so on if there are multiple cpus multiple cores on your processor then obviously each core will run a process independently in a symmetric multi processing system and even the os has to keep track of those processes which are running on each cpu so let's come back again to the concept of process state this diagram is illustrative to understand what happens to a process during its lifetime so when you do a fork the process is created the process will at some point in time call exit or it will do a return from the main function and the process will be over so what happens in between you know what happens in between you know from the moment the process is created till the process is terminated we can understand the lifetime of a process in terms of different states in which the process actually keeps moving so initially after the fork we said the process is a newly created process and basically what happens is here the os will create a task struct or a struct proc in xq6 for that and then the process will be moved to a list of processes which we can see uh is a queue of processes that is most typically called the ready queue and we see the process in the ready state now so when the process in ready state the process can get scheduled by the scheduler and at whatever point in time when the turn comes it will get scheduled by the scheduler and it will now start running so we now can say that the state of the process has changed to a process which is running now when the process is running multiple things are possible one some interrupt can occur or the process can actually schedule an io like process does the scanf and if it does the scanf it has to do something called as a weight or what is also called as blocking and we'll see more about blocking today in any one of these cases you know whether the process does wait for io or the process is interrupted or possibly the process says exit or the main function returns so if it does exit obviously the process is over and then its life is over so we said the process is terminated and basically after this the process does not exist if it gets interrupted like for example for whatever time or interrupt when it was running then obviously the timer the interrupt code will run the interrupt handler will run and the process cannot run so the process we say again goes to the ready state and practically what happens is that this pcb or the the struct proc is moved back to the list of processes which are ready to run if the process is going to wait for some i o then we say it has moved to the waiting state and it can come out of the waiting state basically only when the i o is completed now how is how is it known that io is completed basically whenever the io is completed and hardware entropy locker once again that hardware interrupt will occur now in the context of some other process which was running isn't it because this process is not running the process that we are discussing it is waiting so in the context of some other process running the interrupt will occur then the kernel code will run it will then move the process out of the waiting queue to the ready state you know the list of processes that can be scheduled so this is you know vaguely the concept of process state we will see it now in the context of x364 so this is the xv6 code for a process so what i'll do i'll not show you this code to you here i'll rather show it yeah in the actual x36 code so i have to i think stop change my screen sharing let me share my terminal with you i'll stop sharing i'll share my entire screen with you so i am going to the terminal here so here i am on xv6 code as you know i use c scope and vi i have started browsing this code now because i have studied x36 code i know that the structure is called proc now when i search for global definition of proc i get four uh definitions but i am sure i am interested in this because this looks like a definition so this is the struct proc so this is the pcb process control block in xp6 code what are the fields that it contains a size this is more or less related to the memory management okay of this particular process same is the case with this pgdr pointer so we will right now skip these two because we are yet to see the memory management so the case stack pgdr size they are all related to the memory management then the process state is here and it's a enumeration in c with the possible values that you can see are listed here in your proc state so these are the possible values now as you know enumeration in c allows you to use this symbolic names they are actually numbers but you can use the symbolic names so some of the states that we have already discussed right so runnable means the process is now in the ready queue you know it can be scheduled running means the process is actually running means the process is waiting for an io embroidery means the process is just now created but not made hunible yet right okay zombie we will see okay just hold on for some time zombie is a special state we will come to zombie so that is the process state here is the process id the next field is the process id as you can see here then there is a parent pointer you know struct proc self referential pointer to the parent process the traffic and context again they are used for actually scheduling of the process saving the what is called as a context of a process about which we will see today and you know in order to save the and save the entire state of the process when it was running so that it can be rescheduled again so that is captured here the interesting thing which we already seen is this the list of open files so basically and this is the array of struct file pointers and an index into this is the file descriptor right here is a process name you will see there is a pointer called struct inode pointer called current working directory cwd we will revisit this okay this is basically you know as we know uh going to give the present working directory or current working directory of the process so some of these fields you know we are obviously not going to look into more detail today uh we will keep visiting these fields as we keep learning more and more concepts let us first understand so this is just the declaration of the struct proc where is the actual list of all processes now in x 6 it's very simple xq6 code is very simple code and i'll give it this file and as i know this line of code okay in proc.c what we have here is a variable called p table a global variable called p table which is basically containing a list of struck proc as you can see where this n proc is 64. so apparently this is an array of size 64. so x36 can actually support maximum 64 processes okay concurrently running so this is basically what we say the total set of all the processes okay now interestingly x26 has no fancy data structures okay and it does not have different queues also all it does is you know in this is the single array that xv6 manipulates no matter whether the process is in ready state or the process is in waiting state or whatever you know all the process are only in this array because xq6 is a very simple piece of code if you look at linux kernel code you will see actually lot of different list and not arrays but list doubly circular linked list in the linux kernel code what i'll do is i'll once again go to the struct proc and here is the state so what i'll try to do now is i try to find all occurrences where the state variable is accessed now you can see here that in alloc proc function there is a code which is looking at the state in the alloc proc function the state is being set to embryo as i as i said embryo is the states used when the process is actually getting created and here you can see the state is being set to unused you can see that in the function user init the state is being set to runnable in this function fork the state is being set to unused in the fork again the state is being set to run able now exactly where and why it is being set to a news and runnable we can understand when we go to study the code of fork similarly you can see that the state is being set to zombie in the exit function and it is being checked for zombie in the wait function and you can see there is a function called scheduler which is actually stating the set to running now even though we have not seen any code of the scheduler i think it it should make sense to you that it is the scheduler code which is going to pick up a process for execution and when it actually passes control over to the process before that it has to say that the process is in a running state and there is a function called yield and is a function called which is basically setting the state to runnable okay so runnable means the process will be in the runnable state it could be picked up by the scheduler and so on so we can see that there are different pieces of code which are accessing the state variable and different parts of the kernel code are modifying the state you know to reflect the various phases of the process we could also see some common sense code here that you know it is exit which is setting the state to zombie it is scheduler which is setting the set to running it is yield and basically yield is called you know when the process time quantum is over or the process is about to hand over the control to some other process in that case it is being made runnable and sleeping i think we did not see sleeping but we can actually try to find somewhere where the sleeping state is being set yeah so there is a function called sleep here as you can see and in the function sleep the process is set to the state sleeping and you will see interesting there is also a function called wake up which is checking if it is sleeping and wake up is setting it to runnable so you can see that this state of the process is being changed by different kernel code so this obviously doesn't explain completely how the state changes but we will revisit this today itself after we see some more details of what exactly happens with the process state let me switch back to my slides so we have seen you know this xv6 code in the case of linux kernel code and let me also go to linux kernel code okay i'm going to linux kernel code and i'm using cisco i think there is some issue here i may not be able to jump yeah there's some issue here okay so that was some issue with the way i use my cisco i'm going to a header file i think uh yeah so there's a file called include linux dot h and i know this contains star struct yeah so this is structure struct this is the pcb in linux kernel code okay and you will see i am on the line number 629 right now and i'll go to the end of this that is line number 1299 so all this structure itself is 700 lines including lot of comments of course but then again there are lot of lot of fields here because linux is a more complex advanced operating system so to you know give you all that complex advanced features the processes have to be in advanced state but what i have done on my slide i have picked up certain fields okay out of all these fields that you see here and some of these fields are actually inside a hash if diff kind of macro these are the interesting fields there is a state and there is some scheduling information and the parent pointer and a list of children and this is the file array okay so it's a pointer but basically the pointer is going to point to an array and then there is a complicated structure mm stuck for memory management of the process okay so that is the linux kernel code i'll only quickly try to tell you what is the listed in linux kernel okay i think yeah listed i can show you cisco yeah so you will see uh now i know this is all drivers code so the list head i really want to see should be this so you will see that the struct listed okay is nothing but a listed type of pointer next and previous okay and uh what happens is in the task tracked okay so you will see in in the task track of linux kernel uh there is a listed called rcu node entry there is a listed called rcu task foldout list there is a list head called tasks there is a listed called children there is a listed called sibling and so on so basically this is a structure which is containing a variable of the type listed and the listed contains two pointers to the structure of the same type that is listed so self referential pointers so there is a very interesting way in which the structures actually get combined so this pointer okay sorry so this pointer and this pointer they are actually pointing not to the outside structure but actually to the listed structure inside this so as you can see the listed structure is something like this and the pointers are actually pointing to this and to this and so on so the question that now arrive this is if you know you start traversing let us say this is the listed structure outside here and the listed structure outside here has a head and a tail pointer and obviously you know they actually create a circular linked list it will be a circular link list okay like this of the head and tell now the real fun is actually if you want to traverse the list of you know structures like this you can follow basically this chain of let us say head and next pointers you can traverse this chain but if let us say you have a pointer p and pointer p let us say pointer p is a pointer to this now what you want to access is not this structure but actually you want to access this structure so the interesting question in c is how do you go from this location to this location so to define the problem given a pointer to a internal variable of a structure how will you get a pointer to the structure the problem here is given a pointer to an internal member variable of a structure how to get the pointer to the actual structure and there is a very nice intelligent c code which solves this problem um i will cover that in one of the lab sessions but those who are interested can actually speak more about it so what happens in linux kernel as we have seen there are lot of listed as we have seen just now so as a part in linux kernel one particular pcb can actually be on multiple lists one pcb can be on multiple list okay just to indicate the different roles that a process can have in the overall execution of the system so remember in in x36 code which is a very simple code forget multiple list there is actually just one simple array okay for everything to be done with processes in the case of linux kernel there are lot of lot of queues and there are a lot of list heads inside the task truck which enable us to put the process on one or multiple list so basically you know in a typical os code you will have doubly linked list this is a null terminated list but they are most typically circular and there can be a list of processes which are ready to run there can be a list of processes which are waiting in a queue so i'll skip this slide right now okay let us come to the important concept called context switch so first of all let us try to understand what is the context and what is the context of a process actually so by context we mean the entire execution context of a process so all the things needed basically all the hardware things needed all the resources needed for the process to execute what does it include it includes the values of all cpu registers the current state of the process the memory management information about the process actually all configurations of the cpu which are specific to the execution of the process same we can say about kernel also because when the kernel is running it will have certain values in the cpu registers and it will not have a process state because it is not a process but kernel will have its own memory management information also so that is what we mean by context now what is the meaning of context switch basically if let us say a process is running on the cpu and from the process you want to switch to another process or for that matter even to kernel then obviously the registers of the processor need to be loaded with the values of the other process or the kernel and that means whatever were the values in those registers on behalf of the currently running process will be wiped out they will be wiped out because there is only one set of resistors isn't it no matter whether a process is running or kernel or on the process or whoever is running is only one set of resistors most typically in a cpu there are some modern day cpus which provide multiple set of resistors but let us not go into that essentially if from one process you change to another process then because the code of those processes and kernel was compiled independently it was compiled assuming that all registers are available to me and that is why the set of registers of the earlier process now needs to be saved before you switch on to another process so what you have to do we say is you have to save the old context and load the new context the question is where to save the context of a process the simple answer is in the pcb because the pcb is the data structure which uniquely identifies the process and the very purpose of the pcb is actually to store all the information necessary to control the process it's a process control block right so the context is the part of the control information now the context switch is actually an overhead because when the one process is getting switched to another process let us say during scheduling then no useful work is happening and then the time required for this can vary actually and some processors they actually have special instructions to let us say save all the resistors in one go okay so like you can just write one machine instruction give it an address and all the registers will be saved to that particular address so cpus have these kind of instructions as well now this is a diagram again taken from your textbook which is trying to explain what happens during the context switch so this is the timeline okay down here the time grows let's say now process p0 was executing what does that mean that the cpu registers were occupied with the values needed for executing p0 processes code now let us say there is a interrupt or a system call which takes place now now obviously after this when there is a interrupt the os code is going to run right what is the os code going to do now the os code will first of all save the state of the p0 process into the pcb of that process that is pcb0 what is missing here is that the set of resistors to be used by the operating system you know will also have to be loaded again you know in the set of resistors so the os code will now run and let us say it was uh the os code was over and let us say it was a timer interrupt and when the timer interrupt for os code the the timer interrupt handler in the os code runs the scheduler runs and scheduler has decided not to run process one so what it will do now it will read the state of all the resistors from pcb of that process reload okay reload state from pcb pcb1 means what take the values of variables in the pcb structure load them into the resistors and then pass the control to p1 and then p1 will start running what do we what is exactly going to happen here it is basically the instruction pointer okay the eip or what is also called as the program counter will be changed the moment to change the program counter p1 will start running okay because the program counter will now point to p1 support now same same way p1 can get interrupted when it gets interrupted first thing to happen is to save the state of the process into its pcb do whatever is required okay this will be done by the os and then at the end eventually os will load the state of another process it may not be zero here the diagrams is pcb0 it could be some pcb3 or pcb4 also and then the other process will start executing so basically what happens to summarize whenever there is an interrupt or a system call the mode will obviously change from user mode to kernel mode at the same time the context of whatever was executing currently will be stored okay the context of a running process will be stored in its pcb then the os code will run os code will then restore the context of another process from the pcb to registers and make that process run so this will keep repeating all the time now what is the particularity of a context switch code now suppose the process is running then it will do function calls and function calls as you know work in lifo fashion that is made using the calling convention because we have seen how the calling convention enables execution of function when an interrupt occurs it can occur any time there is no no prediction you can make about when interrupt will occur in fact you know during the one second of my speech so many interrupts might have already occurred on my computer so interrupt can occur any time and the context switch can happen actually in the middle of execution of any function so you never know which function was running when the context which was required after context switch one process will take place of another on the cpu and that is why this switch is obviously not going to happen using the calling convention because you are not going to call anything when i say call anything i mean use the call instruction because the call instruction as we have seen is central to the functioning of the functions and when the context which is happening there is no call instruction it is basically a interrupt occurred os is running and os is saying some to some other process now you run that is not going to happen using the call instruction so the calling convention is getting broken here and that is why the code for context switch has to be in assembly because if you ask the compiler to do it the compiler will keep generating code using the calling convention and you cannot do context switch using the calling convention because it is precisely breaking the calling convention okay in the place of one process you are bringing in another process this is not a function call okay one process goes os comes os goes another process comes this is not a function call right and all these have to happen on the cpu through some instructions so those instructions which need to be run to change the context they cannot be generated by the compiler compilers are supposed to generate code as per the calling convention so this this code has to be written in assembly so you will notice that in all operating systems the code of switching context is always in assembly which is handcrafted to actually break the calling convention but still ensure that the handover happens properly let me check if there are any questions there are some questions in the chat my voice is it still breaking is my voice still breaking no sir okay fine now let us try to understand a particular concept which we use very often okay the concept of giving up of a cpu by a process or what is called as blocking so one way that a process can be taken out from cpu is when let us say timer intro occurs or for that matter any interrupt occurs so the process will be moved out of the cpu and os will run in the place of the process but sometimes a process actually gives up the cpu what do we mean by that let us try to understand so here is a code okay and this is a code which is just trying to do some scanf so as you know when you code the scanf scanf is in the c library user level code that will run scanf will eventually do a read 0 where 0 is the file descriptor there is a function called function called read in the standard c library again user level code which we know will essentially do some you know variant of the int instruction the interrupt instruction now the moment you know this instruction runs we know that the control will jump into os code now through a particular setting of the hardware you know done by the os when it was learning this particular location is occupied by the system call for the read so let us say the os code for readings is underscore read now we can actually predict few things now because we have seen some data structure when you call read you are passing the file descriptor to it it will actually go to the pcb of the current process so let us say current is a global variable giving the pcb of the current process then you'll go to the file descriptor array index it using fd get the struct file in the struct file you will get the file position right now eventually let us say you are reading from a file then the read will happen from the disk so through some lot of lot of code eventually you'll read some function like a discrete isn't it and you will tell from where you know on the disk the data should be read what will discrete do discrete will effectively do some kind of a privilege instruction that will have to be written in assembly to write to a you know some port used by the particular device so their in and out kind of instructions will be run now it so happens that all these ios the reads and writes we say they happen a synchronously so because see this is going to take a lot of lot of time one may now is you know one way that we can actually handle things here is that you keep doing a loop here instead of returning we just keep doing a loop you know a loop which keeps continuously checking whether the disk io is complete in that case this particular os code will keep running it will keep consuming the cpu all that it is doing is you know something like a while one wait till the data is ready so cp is being consumed cpu is being consumed on behalf of this very process that we are discussing without doing anything meaningful so obviously that time quantum for which you will keep doing the while one busy weight here will be wasted so what the operating systems modern operating systems do is they do a a synchronous io so a function like disk read will basically schedule an reading activity it will return it will return in the system call the question is what to do now so because it's a synchronous read now what we want is that the process should not run until the data is ready and particularly it's a scanner so you never know when the user presses the key so what we want is that this process should be taken out of execution it should be taken out of the ready state it should be now taken to a waiting state okay so what is done now that the pcb of the process is moved to a waiting queue and then the code will call the scheduler the moment you call scheduler scheduler will pick up now some other process because this process is no longer in the ready queue this process has gone to some weight queue so when you call scheduler it jumps into the scheduler code and scheduler will now select on the process and make it run and this process will now wait in some other way queue when it will come out of weight queue when on when some io interrupt so when the disk io is complete the disk hardware will raise the interrupt that interrupt handler okay will actually move the process again back from the redicu the waiting queue to the ready queue so what we have essentially here is a code to move the current process control block from one queue to another and calling the scheduler this is what you call blocking when you say a process is blocking itself for io we essentially mean that in the os code at some point in time the pcb of the process is moved from the ready queue to some waiting queue and a call to the scheduler is made so then the scheduler runs and it selects some other process for execution any questions a new state is actually you know this is specific to the xv6 code because xv6 is a very simple code okay let me go to xp6 code so what we have seen is all that x26 has is this array that's it whether the process is ready running waiting whatever all processor only in this array there is no other list no other queue nothing so how do you identify out of all these processes which is ready which is waiting which is running and so on basically you set up the state inside each block basically you set this state in each of this proc right so let me go to the scheduler code now of xv6 so the function is called scheduler i want all of you to actually start reading x36 code now whatever i am doing in this class i want you to repeat after the class is over i want you to go and see the code now we will understand the scheduler code partially right now fine so you will see the scheduler is basically some infinite loop you will notice what it is doing here it is actually iterating over the array called p table we have seen the array p table the global variable p table you will see it starts with the first element of the array and iterates over the entire table right all it is doing is it is trying to find a process whose state is runnable and once it finds the first process whose state is runnable you can see what it is doing it is setting the state to running and then it is calling the switch function which is going to do the context switch okay it will switch the context from the kernel code to the process and after the switch is over the process will be running kernel will not be running now what is this switch let us jump to that you will now notice that this switch is a assembly code the code to actually switch from one context to another context is a assembly code because the compiler cannot generate this code okay compiler cannot know how to switch from one process to another compiler will basically generate code using calling convention so the point is there is only this single array okay everywhere that's it so all that the scheduler does it it looks for a process which is runnable makes it running that's it now in this array there can be lot of other stuck procs some of them will have the state waiting some of them will have the state runnable and so on unused means you know it is just not a used structure that's it out of the 64 structures let us say have created only 10 processes so out of 64 only 10 structures will be in use what about others you have to set a proper state value in them to indicate that they are not in use so the unused value will be used for such structures okay i hope it answers your question a new struck proc is basically a proc which is not used for any process when you create a new process actually in fork so in fourth now as you can guess in full there should be a loop which will iterate over the same array and look for a process whose state is unused so we can actually logically guess that and let us see the code of 4 so here is a code of work let us keep all this fine we can skip some of these but now there is a function called unlock proc which is getting called if i go to unlock proc you will notice that it is iterating over the same array and if the process state is unused then it is as you will see doing something with the same p and returning the process p so the unused state is actually used in fork okay in the unlock proc does it answer your question i'll wait for more questions shhh [Music] [Music] so how does the os understand like whom to give the input to so the simple answer to that is the process in the foreground right the process in the foreground will basically get the keyboard input the next question then comes how does the os know which process is in the foreground right so we will wait for the answer to that question like what the context which is happening and that time the keyboard input is present at a keyboard input so then what do you like so you are saying so now the interesting question here is this fine you you said the context switch is happening right now who is going to do the context switch the kernel code so what is running when the context which is actually happening kernel code is running right when the context which is actually happening so yeah so basically the the processor is actually executing in the kernel mode because the kernel code is running now there are two options from here okay option one is you know kernels will disable interrupts when the kernel code is running and then that way your keyboard press will be lost because the kernel is saying when i am running i don't allow any hardware interrupt now obviously this kind of kernel is not a very attractive kernel but this type of kernels are called non-interruptible kernels a kernel code which cannot be interrupted so these kind of kernels are called non-interruptible kernels the other type of kernel code is when the kernel code is itself running it can be interrupted so that is called an interruptible kernel all modern operating systems that you see today those which are popular are all interruptable kernels so now if you have done the hardware if you say press the key when the kernel code itself was running no matter whether it was doing context switch or something else then the key should ideally go to the kernel but then the kernel is not waiting for any i o right so then the kernel code programmers have a choice of either to ignore the keystroke until your data may be lost or deliver it to either the earlier process or the next process so they have to keep it buffered and whenever the next process tries to read data they will try to give it to the next process any other questions okay there is a question if there are two processes waiting in the wait queue and they are waiting for different kind of interrupts will the fifo manner of queues still be used or the process whose expected interrupt occurs first is scheduled first the process whose interrupt occurs first will be moved to the ready queue scheduling is an independent thing so whatever you are saying is if a process is waiting for keyboard input and other process is waiting for disk io to be complete then which one of them will get scheduled first i'll say i'm i'm reframing the question which one among them will go to the ready queue first so i'll say whichever interrupt occurs first that process will go to the ready queue first now being in the radio queue first does not guarantee you earlier execution because it all depends on how your scheduler works does it work in a first in first out fashion or does it have some kind of priorities for processes and so on so the scheduling will definitely happen if you are in the ready queue and your scheduling algorithm is fair enough it will give you chance now or later but essentially that process will go to the ready queue first whose interrupt occurred first i hope i answered your question rishikesh yes somebody has turned on the mic please speak somebody turned on the mic please speak okay let's move to the code so xv6 and we will keep revisiting exclusive code what we saw just now is that in exclusive there is a single array of procs and all other code like the scheduler code the four code even the waiting code all of them will basically access the processes only from this array that way the data structure is extremely simple in xv6 okay um linux's will have more complicated data structure for dealing with processes okay so we saw the concept of giving up of a cpu by a process so that is actually done in yield in so there is a okay yield yield means give up right so the the yield function in x26 is actually uh to give up the cpu where is yield this is yield okay as you can see what is happening in yield in xq6 code uh is that the process here is setting the set to runnable that's it the process this code which is running on behalf of the process its a kernel code but obviously it is running on behalf of the process all you are doing here is setting your own state to run able okay the myproc will give the currently running process the state is being set to runnable in a more complex kernel this single line of code will get replaced with you know cutting a structure from a doubly linked list and pasting it in another double increase okay in exclusives this code is very simple depending on the data structure this could be more complicated and what it is doing after that calling scheduler that's it it is calling the scheduler so basically once you call the shared function which is a part of the scheduler code some other process will run so this is what we mean by giving up the cpu okay and the ultimate function in x56 which does this is the yield let us see who is who all are calling yield okay so i am trying to see what are calling yield so there is a function called trap which is calling yield and uh so as you can see here is the function yield being called from the function trap and when is the trap called who is going to call the trap so let me just tell you the trap is called on the hardware interrupt the trap is called on a hardware interrupt or a you know any kind of uh interrupt even a software interrupt essentially all software interrupts will also lead to the trap function so whenever a process does a re write or those kind of functions they will eventually come to the trap and they will lead us to the yield function so that is the concept of giving up of the cpu again we will keep revisiting these functions in future also when we see the code of xv6 in more detail there is another question in the scenario of kernel buffering the input is it possible that the input is not given to its intended process the question is who is the intended process how do you know who is the intended process so if there was a process which is already running and the process was waiting for the keyboard input then that is the intended process so in that case the kernel will buffer the input and deliver it to the process when it is scheduled next so exactly how the io is handled how it is delivered to the proper process which was waiting for it we will postpone to our discussion on io that will postpone to our discussion on io more questions all right we will continue tomorrow and tomorrow we are going to read some interesting but complex x36 code which deals with context switch which deals with interrupt handling and which deals with the scheduler there will be assembly code obviously and a few more concepts but we'll do that tomorrow i will advise all of you to read x26 code the code of scheduler the code of the trap function and even if it doesn't make sense complete sense to you that is fine tomorrow we will try to you know understand the magic that is happening so we will stop now and tomorrow will continue with the xc6 code