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

Lecture - 14
Loading the Kernel,
Initializing the Page table

Welcome to Operating Systems lecture 14. So, so far we have been looking at Xv6 code and we said an operating system writes some code to the boot sector which is the first 512 bytes of the disk, which is responsible for execute, starting execution in 16 bit mode and then loading the kernel from the disk and transferring control to it right. We saw that the first few instructions in the boot sector are written in assembly and very soon the assembly instructions initialize the stack and initialize the environment to a 32 bit environment, initialize the address space using segmentation and then jump to C code right.

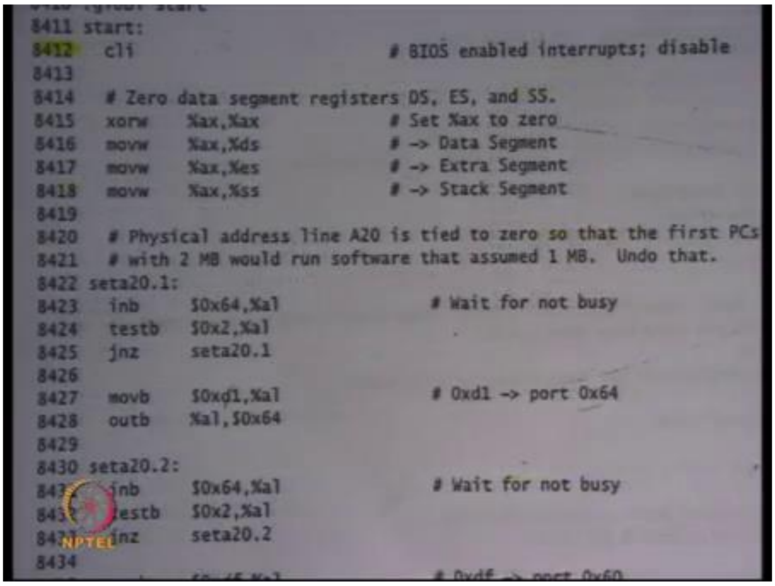
So, the reason that it jumps to C code is because it is easier to program in C, it is a slightly higher level language and so, it you know it makes it easier to read the program as opposed to keep coding in assembly. On the other hand there are some things that can only be done in assembly right, because things like execution of the LGDT instruction right; there is no C counter part of the LJUMP instruction or there is no C counter part of the l jump instruction right. So, there is certain instructions that have to be done in the assembly and for other things you just use C right.

There was a question just now that why C why not some even higher-level language like Java, right. So, I mean it is a tradeoff, C is reasonably high level and yet the compilation of C usually gives you know the assembly code that comes out of compiling C is usually very close to in performance to hand written assembly code right. And so, performance is not lost really, significantly if you are writing programs in C; also, the environment that needs to be initialized for a C program is pretty small.

If you saw what got initialized, you initialize an address space and you initialize the stack right and that is it and you could jump to a C program. On the other hand initializing in the environment for something like the Java runtime which involves a JVM, it involves garbage collection and things like that will involve more steps to be able to run execute the first Java instruction right. Also, Java, because the I mean languages like java assume the presence of a virtual machine right. So, the reason Java claims to be a platform independent language is because there is a virtual machine layer sitting JVM, Java Virtual Machine layer sitting between the real hardware and the program. And the so, the JVM needs to actually induces some level of overhead, performance overhead.

So, if the performance of an operating system is the lowest layer of software and performance is often critical right. So, because if your lowest layer of software is not performing to the best then every other thing will get affected. So, performance is really critical so, you know you make a tradeoff between convenience and performance and. It is actually so, as we as I said it is not necessary that java is more convenient either so, alright. So, we were looking at the boot sector code.

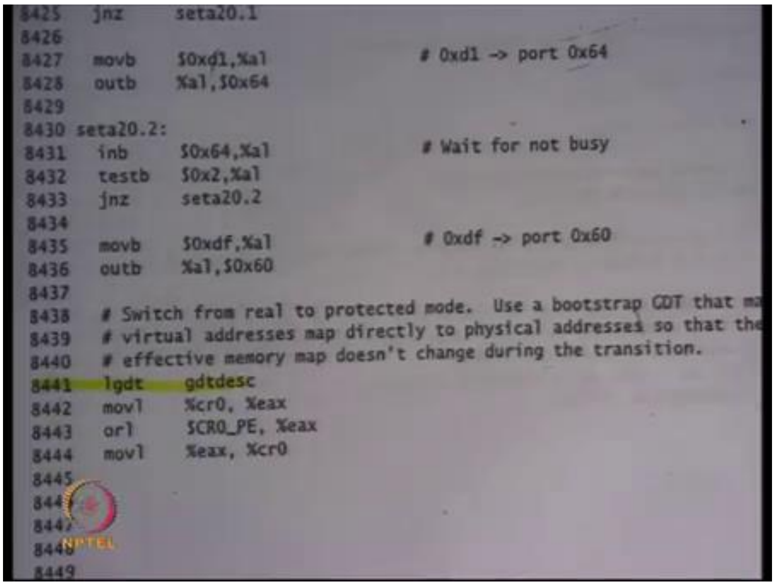
(Refer Slide Time: 03:53)



```
start:
  cli                          # BIOS enabled interrupts; disable

  # Zero data segment registers DS, ES, and SS.
  xorw    %ax,%ax              # Set %ax to zero
  movw    %ax,%ds              # -> Data Segment
  movw    %ax,%es              # -> Extra Segment
  movw    %ax,%ss              # -> Stack Segment

  # Physical address line A20 is tied to zero so that the first PCs
  # with 2 MB would run software that assumed 1 MB.  Undo that.
seta20.1:
  inb     $0x64,%al                # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.1

  movb    $0xd1,%al                # 0xd1 -> port 0x64
  outb    %al,$0x64

seta20.2:
  inb     $0x64,%al                # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.2

```

And we said that the first instruction that executes is a cli instruction cli, which clears the interrupts because I am I cannot just I have not installed any handler at this point. So, let us just clear the interrupts.

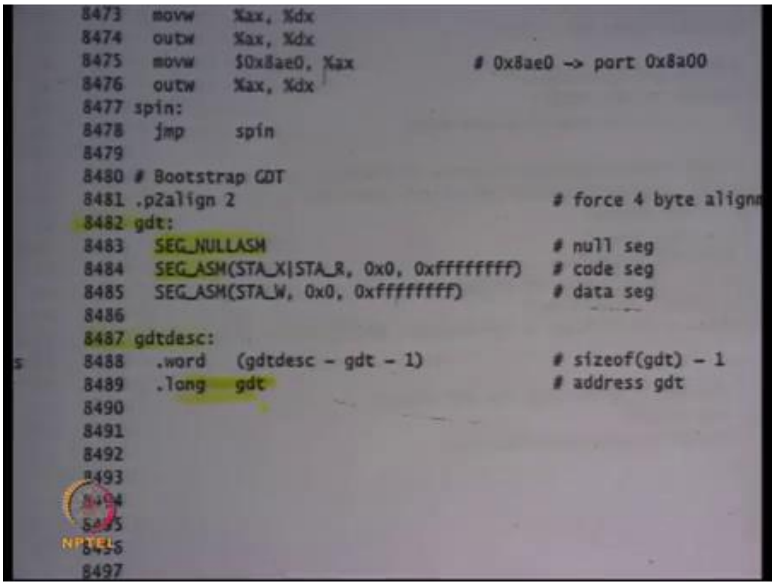
(Refer Slide Time: 04:09)



```
  jnz      seta20.1

  movb     $0xd1,%al                    # 0xd1 -> port 0x64
  outb     %al,$0x64

seta20.2:
  inb      $0x64,%al                    # Wait for not busy
  testb    $0x2,%al
  jnz      seta20.2

  movb     $0xdf,%al                    # 0xdf -> port 0x60
  outb     %al,$0x60

  # Switch from real to protected mode.  Use a bootstrap GDT that ma
  # virtual addresses map directly to physical addresses so that the
  # effective memory map doesn't change during the transition.
  lgdt     gdtdesc
  movl     %cr0, %eax
  orl      $CR0_PE, %eax
  movl     %eax, %cr0
```

Then I do some things to enable 20-bit addressing, above 20 bit addressing; then I load the global descriptor table, we also saw that the global descriptor table had three segments in it, three descriptors.

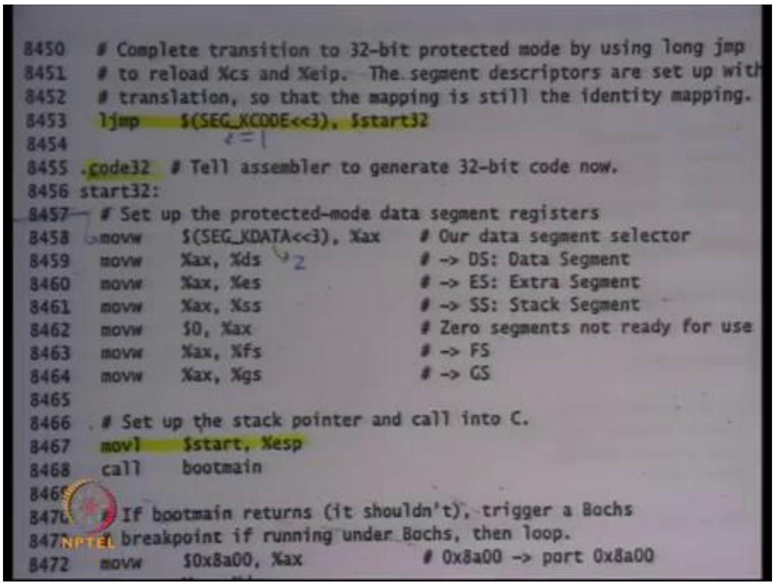
(Refer Slide Time: 04:15)



```
  movw    %ax, %dx
  outw    %ax, %dx
  movw    $0x8ae0, %ax              # 0x8ae0 -> port 0x8a00
  outw    %ax, %dx
spin:
  jmp     spin

# Bootstrap GDT
.p2align 2                                     # force 4 byte align
gdt:
  SEG_NULLASM                                  # null seg
  SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)        # code seg
  SEG_ASM(STA_W, 0x0, 0xffffffff)              # data seg

gdtdesc:
  .word   (gdtdesc - gdt - 1)                  # sizeof(gdt) - 1
  .long   gdt                                  # address gdt
```

The null descriptor, a descriptor which points to 0, to a segment from 0 till 2 to the power 32 minus 1 with execute and read privileges right, and then another segment once again from 0 to 2 to the power 32 minus 1, but with right privileges. And the intent is that I am going to use the segment descriptor number 1 for my code segment and I am going to use the segment descriptor number 2 for the other segments right D S data segment, S S tax segment, E S the segment use those strings ok.

(Refer Slide Time: 04:55)



```
# Complete transition to 32-bit protected mode by using long jmp
# to reload %cs and %eip.  The segment descriptors are set up with
# translation, so that the mapping is still the identity mapping.
ljmp    $(SEG_KCODE<<3), $start32

.code32  # Tell assembler to generate 32-bit code now.
start32:
  # Set up the protected-mode data segment registers
  movw    $(SEG_KDATA<<3), %ax    # Our data segment selector
  movw    %ax, %ds                # -> DS: Data Segment
  movw    %ax, %es                # -> ES: Extra Segment
  movw    %ax, %ss                # -> SS: Stack Segment
  movw    $0, %ax                 # Zero segments not ready for use
  movw    %ax, %fs                # -> FS
  movw    %ax, %gs                # -> GS

  # Set up the stack pointer and call into C.
  movl    $start, %esp
  call    bootmain

  # If bootmain returns (it shouldn't), trigger a Bochs
  # breakpoint if running under Bochs, then loop.
  movw    $0x8a00, %ax            # 0x8a00 -> port 0x8a00
```

And so, I initialize the GDT, I execute some instructions to set up a CR 0, control register 0 to switch to protected mode. And finally, I call an l jump instruction which loads the C S with SEG_KCODE which is nothing but 1; which means you are loading the first SEG, the number 1 shifted by 3 bits on the left to CS. So, that CS now points 2 descriptor number 1 in the GDT, right. So, CS gets loaded with descriptor number 1, eip gets loaded with the address of start 32. And, start 32 is nothing but a function or a symbol that is just defined next so whatever is the address of this, that gets loaded in eip.

So, what you are going to do is, you are going to start executing this code in using the C S register and notice that because the CSS the segment which CS points 2 has base 0 you know start 32 can be used just like that; you know you do not have to worry about start 32 minus base or anything of that is how you know it is all start addressing. But what it also does is it tells the processor that from now on you are going to be executing in 32-bit mode right, and you are going to be executing in privileged mode. And so, all these instructions are compiled in 32-bit mode and that is what this directed dot code 32 is telling the assembler that, compile these instructions in 32 bit mode or assemble these instructions in 32 bit mode, right.

So, as you saw the same instruction has different representation in 16-bit mode and a different representation in 32-bit mode. So, these instructions have to be assembled in 32-bit mode and that is what this directive is telling the assembler, alright ok. And then what does they do, it basically looks for the first thing it needs to do, before it makes any memory access is to initialize it segments right; because all memory accesses go through some segments register right. Recall that the default segment register for all memory accesses the data segment DS, the default segment register for all stack accesses through ESP and EBP is SS.

And also, there are some instructions called string instructions for which the default addresses, default segment is ES, alright. FS and GS are two other segments that are not the default segment for any other instruction, but you can explicitly specify in the instruction that I want to use FS, right. So, what the programmer is doing here is that; he is loading SEG K data, which is descriptor number 2 in privileged mode, last two bits are still 0 in d s, e s and s s.

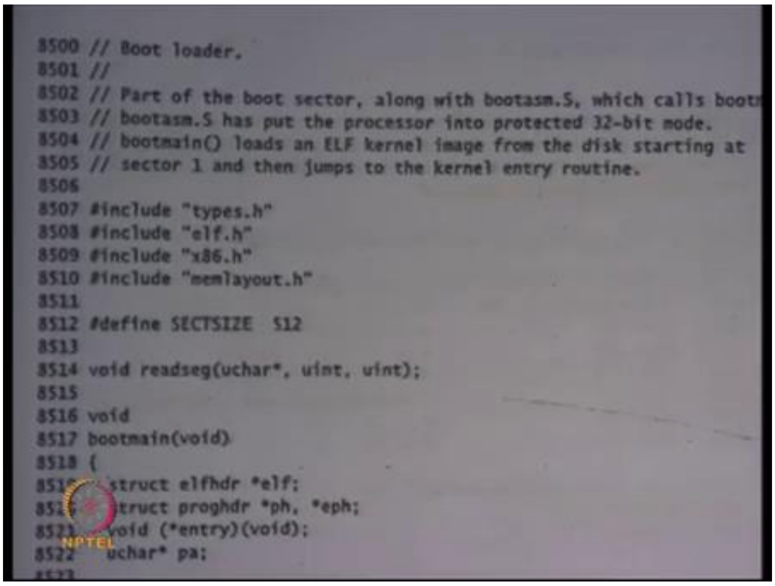
And then it is settings a x to 0 and then loading 0 into fs and gs. What is going on here? It is loading the null segment into fs and gs. So, the programmer does not intend to use fs and gs at all, in his program. So, there will be no instruction in my kernel which will ever use fs and gs, so it might does just initialize it to the null segment. So, any access through the null, through these segment register will create a, generate a trap; but that should not really never happen because my kernel will never be executing through FS and GS alright ok.

The next thing it does is it initializes the stack, we also saw how it initializes stack; it initializes a stack just below the code, just below the were the code was load loaded. So, it is code, the stack is going to go grow downwards starting from 7 C 00. So, the all that space is basically junk because we are there is no code stored there and there is no other data structure that is stored there. So, it can be used for stack.

And right now, I just need the stack for calling my C function; the C function may have some local variables that will also get allocated on this stack, right. Recall that the C function will can be compiled by GCC, as long as the caller of the C function conforms to the calling conventions which we have already seen, it is right. And so, here it is indeed conforming to the calling conventions; firstly, there are no arguments right, secondly, I never expect this function to return, right. So, I do not actually. So, what boot main is going to do is, just going to assume that so, it has no arguments, so it is just going to start allocating on the stack and this function is never going to really return.

If it for any reason returns, which means it is a bug in your operating system, then there is some debugging code which allows you to see what happened ok. So, we can ignore that. So, now, let us look at the boot main function which is defined in C. And once again the assembly program is a caller of the C program right; and the C program is compiled using a specific compiler let us say GCC. And so, as long as the callers obeys the calling conventions it is right, and it is in this case ok.

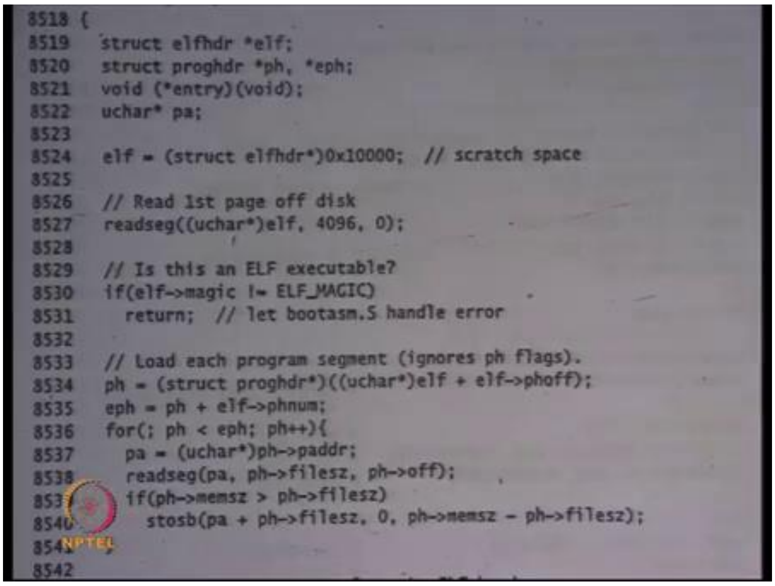
(Refer Slide Time: 09:33)



```
// Boot loader.
//
// Part of the boot sector, along with bootasm.S, which calls boot
// bootasm.S has put the processor into protected 32-bit mode.
// bootmain() loads an ELF kernel image from the disk starting at
// sector 1 and then jumps to the kernel entry routine.

#include "types.h"
#include "elf.h"
#include "x86.h"
#include "memlayout.h"

#define SECTSIZE  512

void readseg(uchar*, uint, uint);

void
bootmain(void)
{
  struct elfhdr *elf;
  struct proghdr *ph, *eph;
  void (*entry)(void);
  uchar* pa;
```

So, here is so, next sheet, sheet 85 is where you see the C code of the boot sector. Once again, the C code is living in the same 512 bytes of the boot sector, right ok. So, here is the function boot main, takes no arguments, returns nothing; actually, it never returns alright. And here are some local variables, they are going to be allocated on stack right, we understand that each stack is going to get decremented.

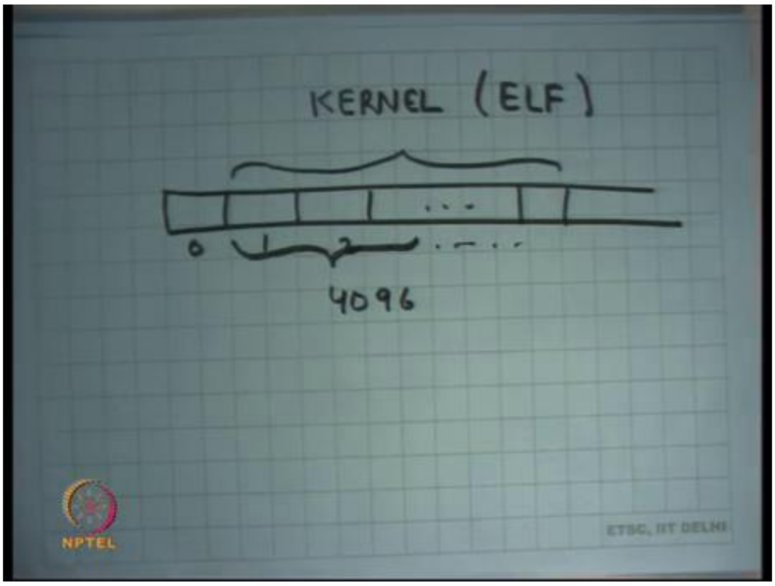
(Refer Slide Time: 10:03)



```
{
  struct elfhdr *elf;
  struct proghdr *ph, *eph;
  void (*entry)(void);
  uchar* pa;

  elf = (struct elfhdr*)0x10000;  // scratch space

  // Read 1st page off disk
  readseg((uchar*)elf, 4096, 0);

  // Is this an ELF executable?
  if(elf->magic != ELF_MAGIC)
    return;  // let bootasm.S handle error

  // Load each program segment (ignores ph flags).
  ph = (struct proghdr*)((uchar*)elf + elf->phoff);
  eph = ph + elf->phnum;
  for(; ph < eph; ph++){
    pa = (uchar*)ph->paddr;
    readseg(pa, ph->filesz, ph->off);
    if(ph->memsz > ph->filesz)
      stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
```

The next thing it does is, it initializes some area at this address 10000, 10000 stands for 64 kb right. So, at 64 kilobytes memory address, it says you know I am going to use this as some scratch space to do some computation and so it initializes that area. And then it reads the first page of disk, right.

(Refer Slide Time: 10:41)



So, in this case, it is assuming that. So, let us say this is the disk logically speaking, and this was the sector number 0 which is the boot sector. What it is assuming is that; the kernel ELF is laid out starting at sector 1, alright. So, this is sector 1, 2 to some value.

Student: Sir.

All right and this is basically holding all the bytes of kernel in ELF format, alright. We also said that the ELF format is an executable format which allows a loader to figure out where I should load these this the code, where I should load the data, what should be the first instruction should that should be executed and all that right. So, it is assuming that starting from this sector 1 and the first 4096 bytes.

Student: Contain.

Alright, contain what would be expected in the first few bytes of the ELF, the header, the header of the ELF file, right. So, that is what he is doing here. So, this particular command here is going to read 4096 bytes starting at sector number 1 into this area which is just some area that is not being used for any other purpose and type cast as an ELF header. So, read those bytes and treat those bytes as the header of the ELF file, alright. So, that is what this type casting means, alright. So, I read those bytes and then use and treat them as the ELF header. And then I when I treat type casted as an ELF header, then I check that you know some fields of this ELF header are correct. So, for example, there is a number called magic in the ELF header, with just checks whether it is indeed is an ELF file or not.

So that particular byte should be set to a certain value to indicate that it is indeed an ELF file. If that bit byte is not said to that particular value, then it is not an or it is a malformed ELF file or something. So, I you know loader should complete a border right there. So, that is what he is checking here, that the magic field is as I expected of the ELF header right, if not then.

Student: Sir, is not a creating a it has a character as a character array is let us been cast to character point.

Alright. So, it is reading 4096 characters that is why it is cast as a character array; but ELF itsELF has a type ELF header right. So, when you are going to. So, once you have. So, you are typecasting an ELF header to a character array and reading bytes as though they were characters; and now you are going to operate on this ELF variable as though it was a ELF header right, that is what you are doing.

So that is what, I mean the reason I can dereference dot magic is because the ELF header has a field called magic. So, I can just dereference it, right. And so, I check whether it is you know, whether it is correctly formed, it should be. And now the ELF header has a field called physical header offset. So, it basically says at what offset does the physical header lives, right.

And from that I get a pointer to the physical header or, so sorry, program header yeah not physical header the program header, right. So, the program header is obtained by adding this offset; offset is specified in the number as a number of bytes. So, you first typecast it as a pointer to bytes or a pointer to characters and then add this value; so, that many characters forward and then you typecast it back to a struct prog header.

So, you know typecast into a structure which represents a program header and the return value is basically the program header that you get, right. The program header itself is actually arranged as an array of program header. So, it is not, it is a program header is not a single program header, but basically in array of program headers you know so, a program header basically specifies the segment.

So, for example, here is the code segment, here is the data segment, you know here is the stack segment or whatever. And so, each segment is going to have some data, some values saying that no this is the offset at which this particular segment lives; this is the address at which this segment should be loaded right, and that is all and this is the size of the segment, alright

And so, the loader can look at that at program header and figured out this is the offset, and this is the address at which it needs to be loaded and this is the size. So, it just reads those many bytes from the disk or from the file and puts it in that address and that is what the job of the program header is, right. So, the program header has this information that where the segment should get loaded, where it lives in the file and what is a size.

And the loader is going to read this information to set it up, about based on what it says, right. So, that is what the boot loader is going to do, right. So, this is nothing but a loader except that it is a loader for the kernel right. In, when we saw when we talked about loading in the context of a process that was loading a process; so, the kernel loads a process, the boot loader loads a kernel, right.

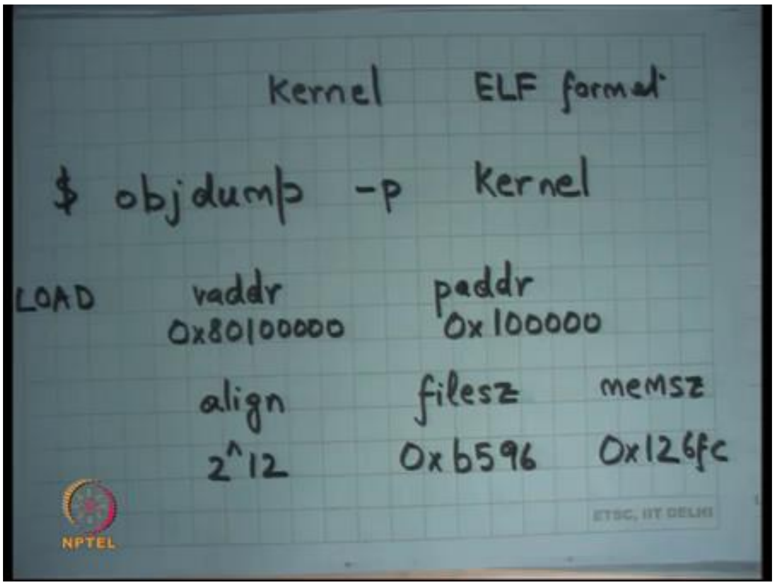
So, that is a so, in this case and both of them and because operations are similar, you in the processes has use, process file uses the ELF format and the kernel file also uses the ELF format in this case right; and typically that is how it is done. So, you are just going to iterate over this array of program headers. So, till you get to you know so, there is another field which says how many program headers there are p h n u m. So, that is the end. So, this is end program header, ph say this is a start program header and you are going to just iterate till you reach the end program header and each time you are going to get the address at which it needs to be loaded; in this case it says paddr which is physical address.

It is going to read the segment off from the file at this offset, right. So, offset says what is the offset in the file from which this program, this segment needs to be loaded and it also knows how much how many bytes. So, file size says how many sizes. So, what is the size on file, what is the offset in the file and you are going to read that segment into this area called p a alright; and p a is given by paddr the physical address, right. So, we just read called read seg and I am going to read those one bytes into p a and p a is given by paddr so, you know basically doing what you, what a loader would do.

And finally, as an optimization ELF header allows you to specify the size on file and size in memory separately, right. So, the program header can have a different value for file size and different value for mem size. So, the idea is that a segment need not have all it is contains in the file, a segment may need to be you know let us say one megabytes large; but only 500 bytes of that one megabyte need to be initialized with some values, all other bytes can be initialized with 0, right.

So, here is that is what he is checking, if mem size is greater than file size for that particular segment then initialize the rest of the bytes with 0, right. So, s t o s b means store byte, starting from file size for these many bytes mem size minus file size, the byte 0. So, store 0 for all the bytes starting from file size till mem size, ok. So, that is what it is doing. And so, it so, it is basically executing the loader logic you know; similar loading loader logic will be inside the kernel to load an executable, here you are using the loader logic to load the kernel inside the boot loader, alright.

(Refer Slide Time: 19:13)



Kernel ELF format

```
$ objdump -p kernel
```

LOAD vaddr 0x80100000 paddr 0x100000

align 2^12 filesz 0xb596 memsz 0x126fc

NPTEL
ETSC, IIT DELHI

So, to understand this better let us look at. So, the kernel is an executable in ELF format right and there is there are some Unix utilities to be able to look inside an ELF file, alright. And so, one of the Unix utilities is o b j dump, alright. So, o b j dump and you can specify kernel as an argument, will print the contents of the ELF file in some human readable string format. So, you can actually you know till parse the file and print it and some human readable format, so that is what o b j dump does.

And so, let us look at what objdump of the kernel file? So, the kernel was compiled by a GCC and linked by some of the make files and all that; but let us look at the finished product and if you execute this command objdump -p kernel, it basically prints out all the program headers in the ELF file, right. So, you can you know on your command from just do o b j dumped dash p kernel and it will print out the program headers.

And I am going to you know, I did this before the lecture and I noted down what were the contents of the kernel file, so that you know it will help our discussion. And so, here is what I saw, you know it said load which basically says it is a loadable segment it said vaddr; so which said what is the virtual address at which this segment should be loaded.

And the value of this was 0x80 100000, right and then they were the paddr which whose value was 100000, okay. There was something called alignment, which says 2 to the power 12 ok, it sets file size which said an hexadecimal b 5 9 6; I mean these values are not important, but mem size 0x126fc, alright ok.

So, this is what this program segment was. So, this segment says that, you know it has. So, ELF allows you to have two different addresses; one word it calls vaddr and other it calls paddr, and the loader is free to choose whichever it likes ok. And we want to see how it, which one is chosen when; it says at what should be the alignment. So, it says you know this particular program should be aligned at 2 to the power 12 boundary; it is a page size alignment.

The size on file is this value is the four-digit hexadecimal number and the size and memory is this value which is the five-digit hexadecimal number. So, what the loader is supposed to do is as load, supposed to load these many bytes from the read these many bytes on the disk and put them in memory and all the other bytes which is this minus this set them to 0, alright ok.

So here is an example of exactly what it does. Also notice that, carefully the developer has chosen vaddr as or and paddr as some as values which are greater than equal to 1 and five zeros which is 1 MB right, why?

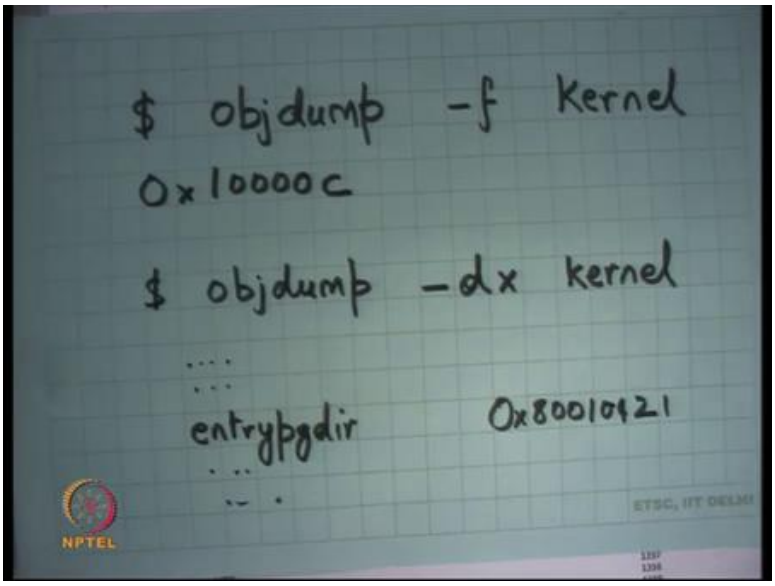
Student: Values

Why is this value not 0?

Because the first 1 MB or physical address space has memory map devices right, like the console and other things. So, there is some clutter there and you want to avoid over writing all that right; you do not want, you want real memory, you do not want a writing, you do not want things to be printed out on console. So, the real memory is definitely one, above 1 MB you will have real memory, right. So, the programmer chose to choose 1 MB in this case.

If he had chosen, could he have chosen a lesser value? Well he would have you to be very careful, if he is doing that so, that is not a great idea. Could he have to choose a greater value, for example, could he have chosen 2 MB? Yes, I mean it is completely valid to choose 2 MB; he is just choosing 1 MB. If you choose 2 MB or wasting more space in the bottom right, I mean why do you need to do that you just sort of keep it at 1 MB, alright ok. Also, if you want to look at the start address, you can use this command called o b j dump dash f kernel, ok.

(Refer Slide Time: 24:31)

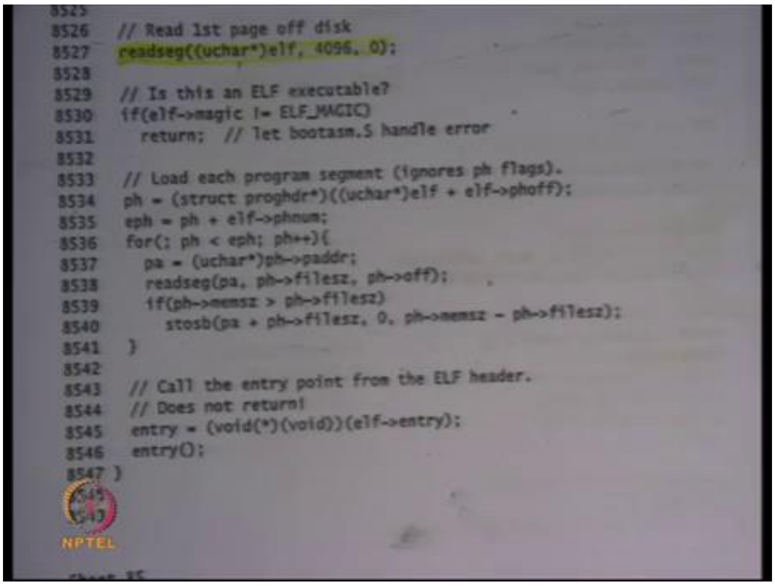


```
$ objdump -f kernel
0x10000c
$ objdump -dx kernel
  ....
  ...
  entrypgdir      0x80010421
  ...
  ...
```

And what is going to you know. So, what I saw was basically this address 10000 and a c ok. So, this is basically saying where is my, what is the first instruction that you should execute. Notice that the first instruction is not necessarily the first byte of the program. If the first byte of the program is that 1 MB; the first, the starting point is 1 MB plus 12, alright 1000 C, alright.

So, with this understanding of how the kernel is, what the kernel has; let us go and look at the loader code.

(Refer Slide Time: 25:17)



```
  // Read 1st page off disk
  readseg((uchar*)elf, 4096, 0);

  // Is this an ELF executable?
  if(elf->magic != ELF_MAGIC)
    return;  // let bootasm.S handle error

  // Load each program segment (ignores ph flags).
  ph = (struct proghdr*)((uchar*)elf + elf->phoff);
  eph = ph + elf->phnum;
  for(; ph < eph; ph++){
    pa = (uchar*)ph->paddr;
    readseg(pa, ph->filesz, ph->off);
    if(ph->memsz > ph->filesz)
      stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
  }

  // Call the entry point from the ELF header.
  // Does not return!
  entry = (void(*)(void))(elf->entry);
  entry();
}
```

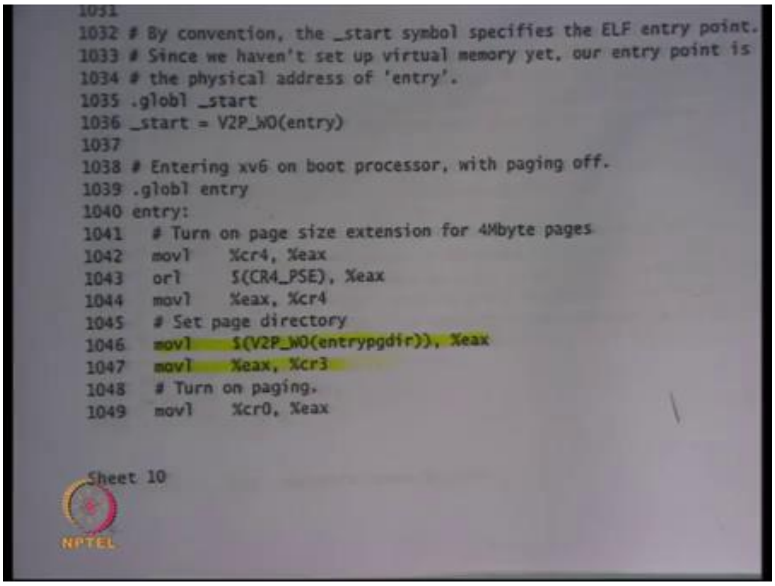
So, here it is going to load the kernel from the disk to address 1 MB in physical address, right. So, notice that he is not using the vaddr field of the program header; he is using the paddr field in the program header. The loader is free to choose whatever he likes right; but he is using the paddr field here, because the vaddr is not even set up right now, right. 2 GB and above memory is not even mapped, it is only the bottom memory that is mapped; we are currently executing in physical address space, paging has not been enabled at, alright.

And so, firstly, xv6, looking at this you can be sure that xv6 will never be able to boot in any machine which has less than 1 MB space, right. Because you know, it is trying to dereference a location which is at 1 MB. In fact, it should have at least 1 MB plus whatever the mem size and the program header says; it does a minimum that xv6 needs, actually it needs a little more, alright. So, so it just. So, it is basically writing it to paddr because paddr make sense or it can just dereference paddr and it can write it there, right. So, the tool chain basically set up paddr such that, the boot loader will basically you read use the paddr values to load it the right place.

And finally, it just calls, it just it figures a. So, the entry field in the ELF is basically that we are where the program should start execution, right and that is that 1000 C. And so, it basically reads that value from ELF dot entry, once again it is in the header; and I just treat it as a function, I typecast it to a function and I just call that function ok. So, at this point I am basically switching from the boot sector executable or to the kernel executable.

So, the kernel executable has been compiled to start at a 100 C and that is actually the code at entry dot s on sheet 10. So, let us look at sheet 10.

(Refer Slide Time: 27:33)



```
# By convention, the _start symbol specifies the ELF entry point.
# Since we haven't set up virtual memory yet, our entry point is
# the physical address of 'entry'.
.globl _start
_start = V2P_WO(entry)

# Entering xv6 on boot processor, with paging off.
.globl entry
entry:
  # Turn on page size extension for 4Mbyte pages
  movl    %cr4, %eax
  orl     $(CR4_PSE), %eax
  movl    %eax, %cr4
  # Set page directory
  movl    $(V2P_WO(entrypgdir)), %eax
  movl    %eax, %cr3
  # Turn on paging.
  movl    %cr0, %eax
```

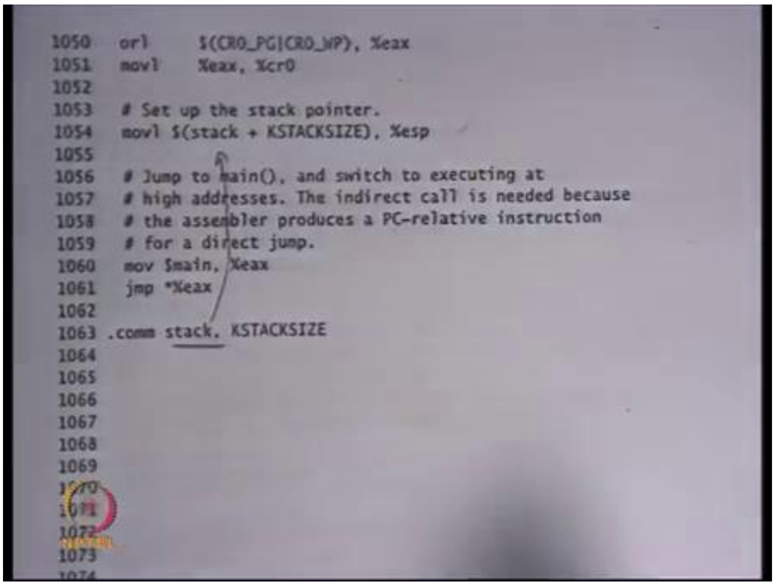
Sheet 10

So, once again the first few instructions of the kernel are implemented in assembly and very soon it is going to jump into C code, right. So, now the kernel has forgotten about the boot sector completely, another kernel will initialize itself. The kernel assuming that there is the physical address space that is mapped, so segments have been set up; but the first thing the kernel is going to do is enable paging.

So, let us see how it enables paging; firstly, it executes some instructions to enable to allow large pages. So, recall that large pages help in you know, reducing the number of pages that you required. So, it first enables large pages, you know you can safely ignore these instructions just to know that they are there. And finally, here is the most important instruction it loads, it uses a macro called V 2 P, which basically converts a virtual address to a physical address. Entry page dir is just a, it is going to be compiled as, the value of entry page dir will be a virtual address in the compiled kernel. So, it will be some value above 2 GB, right.

V 2 P what it does is, it just subtracts 2 GB from whatever the value is, to get the physical address. Recall that the physical address is just a, the virtual address in the kernel minus 2 GB, right. So, V 2 P is just computing the physical address of entry page dir, moving it into this register eax and then moving that register to c r 3, so loading a page directory into c r 3, alright.

(Refer Slide Time: 29:17)

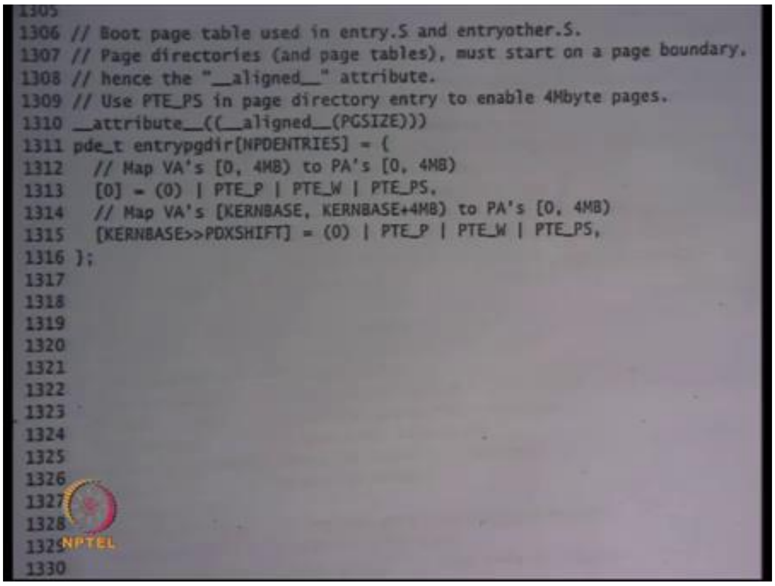


```
  orl     $(CR0_PG|CR0_WP), %eax
  movl    %eax, %cr0

  # Set up the stack pointer.
  movl $(stack + KSTACKSIZE), %esp

  # Jump to main(), and switch to executing at
  # high addresses. The indirect call is needed because
  # the assembler produces a PC-relative instruction
  # for a direct jump.
  mov $main, %eax
  jmp *%eax

.comm stack, KSTACKSIZE
```

And the next thing it is going to do is, it is going to enable paging. So, move cr0 eax, you know some instructions like you know enable paging. So, it is setting some flags or them or-ing some flags and then moving it to c r 0, it basically enables paging, right.

So, once again these instructions are not important, but just know that these are enabling paging. So, let us understand exactly what the contents of the entry page dir are, right. So, that is going to so, as soon as paging is enabled your address space has changed, right. So, the same address now means something else potentially, right. So, let us understand what the entry page dir is and that is sheet 13.

(Refer Slide Time: 29:59)



```
// Boot page table used in entry.S and entryother.S.
// Page directories (and page tables), must start on a page boundary,
// hence the "__aligned__" attribute.
// Use PTE_PS in page directory entry to enable 4Mbyte pages.
__attribute__((__aligned__(PGSIZE)))
pde_t entrypgdir[NPDENTRIES] = {
  // Map VA's [0, 4MB) to PA's [0, 4MB)
  [0] = (0) | PTE_P | PTE_W | PTE_PS,
  // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
  [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
};
```

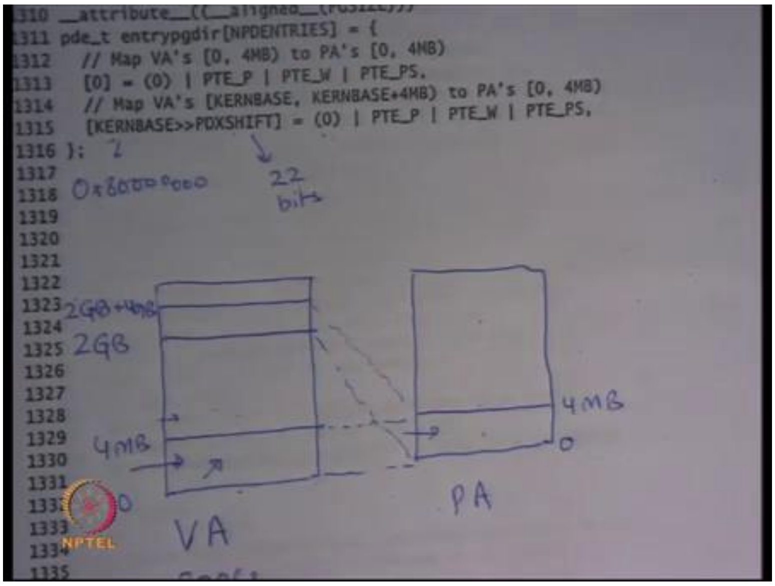
So, entry page dir is declared as a global variable in the kernel space, right. And so, that is fine right; it can entry page dir could have been a global variable, it could have been a heap variable whatever it does not matter. What matters it should have an address and it should have an address in the virtual address space of the kernel and what that instruction was doing was converting the virtual address into a physical address and loading into c r 3.

What is also important is that for the time that this page directory needs to be used, this address stays as it is right; nobody is overwriting this address or writing some other things on this address. So, global variable serves as a nice place to, I mean global variable will have that space allocated to it for the lifetime of the execution. So, that is in order reasonably good place to store this thing. Entry page dir is an array of size NPD entries; NPD entries should be how many, how?

Student: 2 to the power 10.

2 to the power 10, right so, 2 to the power 10 entries in the page directory. The first entry that is 0th entry is pointing to physical address 0, these are the top twenty bits of physical address with the flags present, writeable and page size. Page size means, it is of large page, it is a 4 MB page right.

(Refer Slide Time: 31:23)



```
__attribute__((__aligned__(PGSIZE)))
pde_t entrypgdir[NPDENTRIES] = {
  // Map VA's [0, 4MB) to PA's [0, 4MB)
  [0] = (0) | PTE_P | PTE_W | PTE_PS,
  // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
  [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
};
```

So, what this is doing is let us say. So, this, so let us say this is, let us say this is my VA space and let us say this is my PA space then 0 to 4 MB is being mapped to physical address 0, alright. So, it has an identity mapping here, 0 to 4 MB; that is what the first entry is doing, it is mapping 0 to 4 MB in the virtual address space to 0 to 4 MB in the physical address space, right. The 0th entry is also pointing to a physical address 0, right. And the 0th entry of the PTE is mapping 0 to 4 MB, right for large pages, clear.

The next thing it does it says, look at this entry KERN base shifted by PDX shift. So, what is KERN base? Kern base is 2 GB. So, that is the base of the kernel one two three four five six seven, so that is 2 GB right. And PDX shift is 22 bits, right, so you shift and address 22 bits to get the PDX number, the page directory number, so that is 22 bits, right. So, it is basically saying, look at the look at address number 2 GB, look at the address 2 GB in the virtual address space. And so, this area let us say this was 2 GB to so, it just one entry and this entry also has present writeable and page size bit set. So, it is 2 GB to 2 GB plus.

Student: 4 MB

4 MB, right and where is it mapping it? It is mapping it to 0, once again. So, physical address is 0, right. I hope you are being able to read this right, it is basically saying this is the value of this particular entry. The value is basically, the top 20 bits of this value

integrate the physical address, and these indicate the flags, right. So, in this case the physical address has again 0. So, what is it doing? It is mapping this.

Student: At the same place.

At the same place, ok. So, that is how it has set up the virtual address space. So, the moment it enabled paging, the virtual address space has changed. Before it enabled paging; if you for example, said you know let us say you opened this in GDB, and you said show me the contents of this address. And after you enable paging you said show me the content of this address, you would get the same value right; because then there is an identity mapping, right.

So, the paging did not change the address range, the contents of the address range from 0 to 4 MB right; but it did change the contents of addresses which are above 4 MB, right. Earlier when there was no paging, I could have accessed the address 8 MB; now when I have paging, I cannot address access the address 8 MB, right. Because the 8 MB is not mapped right; but 4 MB is mapped, and it is mapped identically. So, 0 to 4 MB is identical, as it was before and after.

Similarly, earlier if I had said I want to access address 2 GB plus 10 let us say, it would have given me I cannot access that address, was the physical memory is not that large. But now if I say access address 2 GB plus 10, then I will get an answer and the answer will be the value of the byte at physical memory address 10, ok. So, the moment you turn on paging the address space has changed; but the programmer is clever, he ensures that right now all the code and data and stack are living in 0 to 4 MB.

And so, as soon as you turn on paging, these still remain, right. So, it is not like the carpet has you know been pulled under my feet. The addresses that I am using currently are still valid, as soon as I turn on paging and that is the reason, I basically have two mappings from a kernel and user.

What the kernel is going to do next is? Recall that the address space layout of processes that the kernel uses 2 GB and above, and the process uses, the user space uses 0 to 2 GB. So, ideally this mapping should not be there, right. So, this page table is only a temporary page table alright, just to initialize. So, because and you needed the temporary page table to make sure that as soon as you turned on paging, you know it is not like I cannot execute anymore, right; because all your stack and code pointers are still in this area.

So, what the kernels going to do is; first initialize this, then he is going to jump from here to here and once he is in this area, then he is going to remove this mapping and now he can use that for processes, ok. So, there is a temporary page table which has both the areas mapped, 0 and 2 GB. And during that it just switches from physical addresses to virtual addresses and now it can use the lower addresses for use the processes, alright it is clear yeah question.

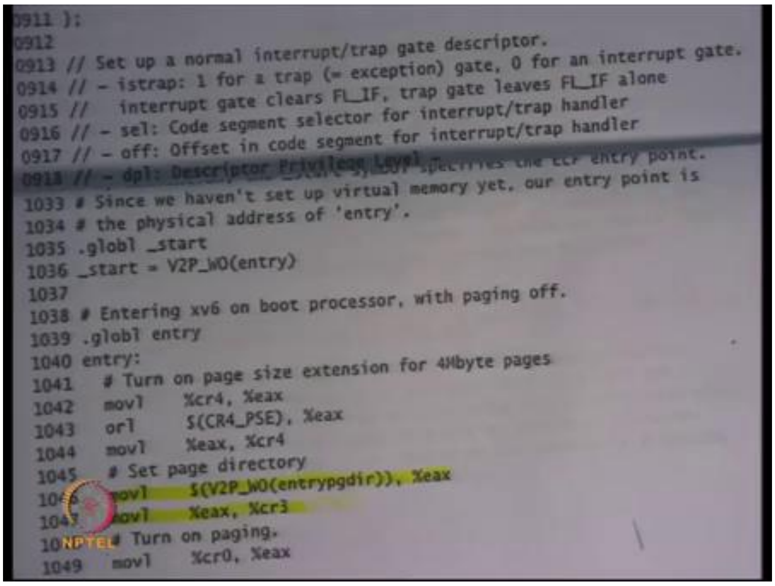
Student: Sir why do not we doing that V 2 P in that in the GB space?

Why are we doing V 2 P of entry page dir?

Student: Because so, there was no entry page directory system.

So, why are we doing V 2 P of entry page dir?

(Refer Slide Time: 37:21)



```
};

// Set up a normal interrupt/trap gate descriptor.
// - istrap: 1 for a trap (= exception) gate, 0 for an interrupt gate.
// - istrap: 1 for a trap (= exception) gate, 0 for an interrupt gate.
//   interrupt gate clears FL_IF, trap gate leaves FL_IF alone
// - sel: Code segment selector for interrupt/trap handler
// - off: Offset in code segment for interrupt/trap handler
// - dpl: Descriptor Privilege Level ... specifies the ELF entry point.
# Since we haven't set up virtual memory yet, our entry point is
# the physical address of 'entry'.
.globl _start
_start = V2P_WO(entry)

# Entering xv6 on boot processor, with paging off.
.globl entry
entry:
  # Turn on page size extension for 4Mbyte pages
  movl    %cr4, %eax
  orl     $(CR4_PSE), %eax
  movl    %eax, %cr4
  # Set page directory
  movl    $(V2P_WO(entrypgdir)), %eax
  movl    %eax, %cr3
  # Turn on paging.
  movl    %cr0, %eax
```

Student: Yeah.

Can somebody answer that? What is the value of entry page dir?

So, the kernel has been compiled such that all the addresses are in 2 GB are in the range 2 GB and above, alright. So, the kernel itsELF has all the addresses in 2 GB and above.

And so, entry, the value of entry page dir will also be a value that is 2 GB and above, that is a virtual address. So, you need to convert a virtual address to a physical address before you loaded into the c r 3 register, right. So, entry page dir value, entry page dir is at a global variable, right. And what is the address of a global variable? It is a virtual address and it has been compiled to have an address which is 2 GB and above, right.

Because most of the time the kernel will run or actually almost all the time the kernel will run in virtual address space, which is 8 GB and above. So, the kernel has been compiled to assume that it lives in 8 GB and above address space, the virtual address space right. And so, all these symbols in the kernel have virtual addresses, which will be 2 GB and above. What you need to load into c r 3 is a physical address; recall that c r 3 takes only physical address, right. So, I need to convert the virtual address into a physical address and that just means subtracting it by 2 GB, because kernel has a one to one mapping and then loading into c r 3.

Student: Sir, but if paging was not enabled how are we having the virtual addresses earlier?

If paging is not enabled how are we having virtual addresses earlier? See we do not have, I mean the kernel has been compiled to assume that it will run in a virtual address space, at and it will start at address 2 GB and above.

Right. So, that is and that is the reason entry page dir has a value. So, that is a compiled value, right that is a compiled time generated value. At compilation time you said entry page dir will have a certain value and those values have to be 2 GB and above. Because entire kernel all the code and everything all those instructions have an address 2 GB and above. So, you know just to make this clear, if you ever do let us say o b j dump dash d x let us say right, kernel. You will be able to see all the symbols, alright. And one of those symbols will be entry page dir and it is value will be something you know, which is above 2 GB 8 0 0 let us say 1 0 1 2 1 ok.

So, the kernel has been compiled to believe that it is going to be executing in the virtual address space, right. And so that is why. So, when you say entry page dir you are going to get a virtual address and you need to convert into a physical address before you actually load into c r 3, ok.

Student: Sir after the paging has been enabled then we will not we have to use V 2 P

After the paging has been enabled, we will not we need to use V 2 P.

How many say yes?

How many say no? Ok, 6 or 7; you still need to use V 2 P; because c r 3 has to have a physical address, right. cr3 needs to have a physical address. So, if you are loading c r 3, only a physical address can be loaded into c r 3; you still need to have V 2 P, ok. So, I have initialized paging and once again what I am going to do is, I am going to initialize my stack, just like the boot sector initialize the stack before jumping into C codes. I want to jump into C code, and I am going to initialize the stack; the stack is declared as this variable of size kstack size, right. So, we allocate some value of stack, it is 4 kilobytes and you move the top of that value into e s p, so that the stack is grouped downwards.

So, you move into stack e s p and now you jump into the main function of the kernel which is, will be written in C, right. The way you jump into the main function of kernel in this cases, you move main into eax and you jump you dereference that using star eax. Notice that he is using an indirect jump to jump to the main kernel, you could have, could you have used a direct jump like jump dollar main straightaway. In general, you can use direct jump, but in this case you cannot and why etcetera let us just defer that discussion for later. Let us just say that it is required that you use an indirect jump to do this and I am going to discuss why you cannot use the direct jump here.

Student: Why are there stars before eax?

Why is this star before eax? Basically, saying that dereference eax and whatever value you get set e i p to that, that is what the semantics are. So, so this is just a syntax AT&T syntax of the x86 instructions; it just jump star to some address, alright. So, what happened here in the assembly code of the kernel, it just enabled paging, it enabled paging such that, the address is it is currently running in are still valid; but new addresses have become valid 2 GB and above and now it is jumping to the kernel, the main the C code.

The main, the variable main itself is a variable that is compiled using the C compiler and the value of the main variable itself will be a virtual address, ok. Just like entry page they had a virtual address which means 2 GB and above; the variable main will also have a virtual address which is 2 GB and above. So, what am I actually doing here is that, from right now and what was the value of e i p right now? E i p was somewhere between 0 and 4 MB right now, right; recall that we never jumped anywhere between to 2 GB, right. We even the start value that we had in the kernel was 10000C.

So, as I just jumped to the physical address 1000C. Now I have initialized a virtual address and here is the first time I am going to jump to the virtual address, right. The variable main contains a virtual address which is above 2 GB, just like all other variables in the kernel. And so, here is where I am jumping from physical to virtual addresses; and here is where I can explain why you know you need an indirect branch. A direct branch a uses P C relative addressing, alright; so it basically says, if the target is x and I am at y, then the bytes that get actually written on those instruction are y minus x, right.

And so, it works if you are working in the same address space; but if you want to switch address spaces, that PC relative addressing does not work, ok. So, that is a short answer, you know if you are interested you can read more about it. But an indirect branch is basically you know, it is basically setting up eax to that value, whatever 8 is you know 8 0 0 something and then saying jump star eax; which means nothing but just replace the current e i p with this value, which is some address 2 GB and above.

And because that address has been mapped in the address table, I will be standing on solid ground; no, I will not fall. Also notice that I the page table only mapped 0 to 4 MB, which basically means, that the address of main should be less than 4 MB, right. And that is actually true, because if you look at this sheet that I had, the entire kernel size is 1 2 6 5 f c which is a 5 digit hexadecimal number and 4 MB is you know 6 digit or 7 digit number. So, it is ok, right.

So, the kernel itself has is you know, the developer is aware of the fact that the kernel is less than 4 MB and that is why it has only setup the first and the only one entry in the page directory, in the temporary page directory entry, right. If it knew that the kernel is bigger, it would have probably had to initialize more entries in the page directory, right ok. So, I have initialized the stack; what is the value of stack? Is it a virtual address or a physical address?

Student: Physical.

Student: Virtual.

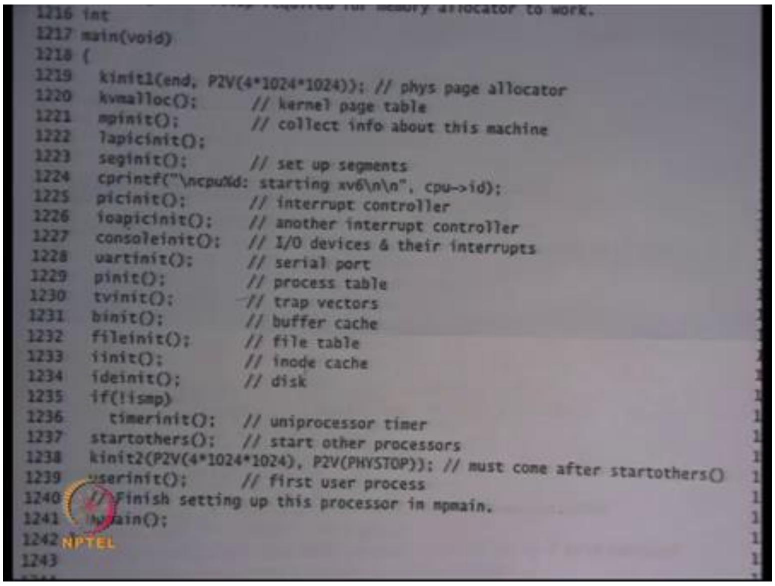
Virtual

Student: Virtual.

The entire kernel has been complied with virtual addresses. So, all the variables in the kernel have only have virtual addresses, alright. So, any time you have you refer to a pointer it is virtual address. So, esp itself it is now having a virtual address; the moment you call jump star eax, eip also has a virtual address. So, now, you will be executing completely in virtual address space ok. And now you are going to start initializing your data structures and your page table, so that you can now run other user processes, alright.

So, I am not going to go into the. So, it is enough for today, but let us just look at the main function very quickly. So, here is the main function.

(Refer Slide Time: 47:37)



```
                                   allocator to work.
int
main(void)
{
  kinit1(end, P2V(4*1024*1024)); // phys page allocator
  kvmalloc();      // kernel page table
  mpinit();        // collect info about this machine
  lapicinit();
  seginit();       // set up segments
  cprintf("\ncpu%d: starting xv6\n\n", cpu->id);
  picinit();       // interrupt controller
  ioapicinit();    // another interrupt controller
  consoleinit();   // I/O devices & their interrupts
  uartinit();      // serial port
  pinit();         // process table
  tvinit();        // trap vectors
  binit();         // buffer cache
  fileinit();      // file table
  iinit();         // inode cache
  ideinit();       // disk
  if(!ismp)
    timerinit();   // uniprocessor timer
  startothers();   // start other processors
  kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
  userinit();      // first user process
  // Finish setting up this processor in mpmain.
  mpmain();
```

So, this is what gets called from the assembly code. So, stack has been set up, I just jump through the main function, alright. And what the main function does? It makes a lot of other function calls to initialize a lot of things, right. So, for example, it is initializing the physical page allocator here, initializing the kernel page table, it is a multiprocessor machine, it is a multiprocessor operating system so, it also initializes other operate, other processor in the system.

So, the way it works is only one processer boots up and then it initializes all the other processes. So, all that is done here, it initializes the IO devices, interrupt subsystems; at some point in this initialization it will reenable interrupts. Recall that so far, we have been running with interrupts disabled, cli was executed right in the beginning. So, at some point in this initialization, it will reenable interrupts after it has set up the interrupt handlers and finally, it will call you know, it will call user in it which will create the first user process.

And that user process will probably you know call fork or something to create more user process, which will eventually call the shell for example. And now the you can type command on the shell and potentially fork even more user processes and so on. So, that is how the user process going to get created. And you know and then you just let all the processes runs simultaneously, right. So, multiprocessor; so, all processes can now execute, and they can see what the processes are, that need to be run and run them run concurrently, ok. But tomorrow I am going or the next lecture I am going to look at k v malloc in detail, alright.

So, we have seen that the kernel initialized a temporary page table to switch from physical to virtual address space. Now I am completely in virtual address space, so I can remove the mapping from the physical address space and so, that is what kv malloc is going to do. And now it will arrange it in such a way that each process has a separate page table and you can actually map user pages into that address space, ok.

Thank you.