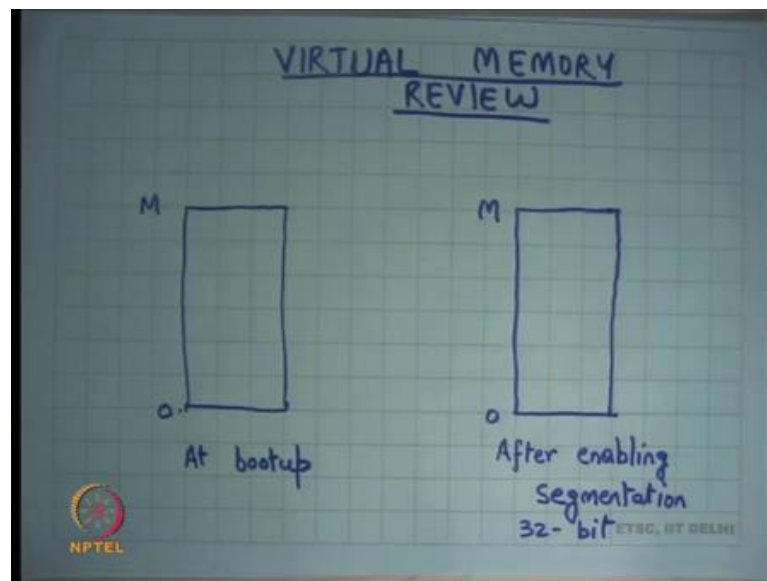


Operating Systems
Prof. Sorav Bansal
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture – 15
Setting up page tables for user processes

Welcome to Operating Systems lecture-15.

(Refer Slide Time: 00:30)



So, let us review the virtual memory subsystem as we have been looking at it. So, when you bootup the machine, we said that we start in the physical address space right. And so, if you use address x , then it just if x is less than M , then you had the physical memory at that address x right. You straight away hit the physical memory there is no translation involved. If you try to hit a physical memory address that is greater than the available memory let us say capital M is the amount of physical memory available in your system, then it will throw an error so that is not valid right.

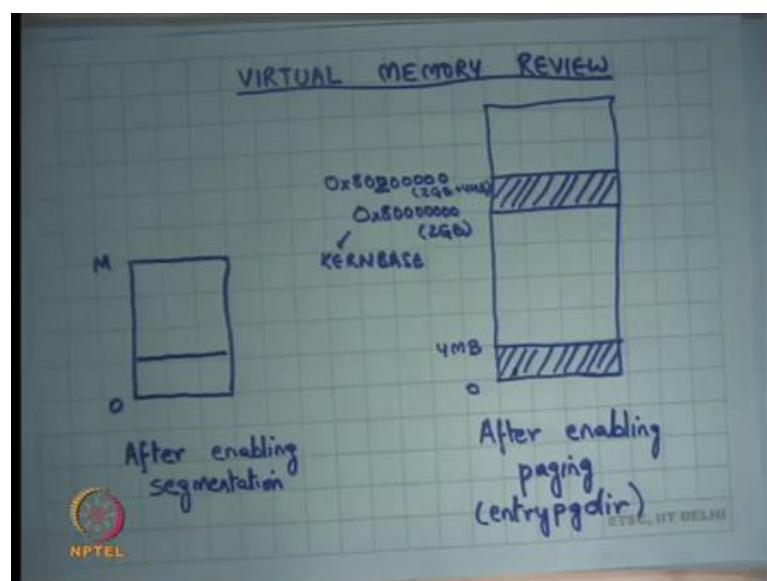
So, at bootup you have this simple system where there is almost no virtual memory subsystem and it is completely flat from 0 to M . Of course, recall that boot in 16-bit mode, so segmentation works in its primitive form, where you can add the segment register after multiplying it by 16 and all that. But for all practical purposes you know the kernel that we are studying sets all segment registers to 0, so you have a flat address space on 0 to M .

Then as soon as you enable segmentation in 32-bit mode, so you switch to you enable segmentation you initialize a global descriptor table and you switch 32-bit mode. The moment you do that you all your addresses are now getting translated through the segment hardware right. Each instruction has a default segment right. So, for example, if you are just making an access through some register, then it will the default segment is `ds` right. The instruction itself `eip` the instruction pointer itself always go through go through `cs` and all that.

So, in theory you could enable segmentation and you could use segmentation to segment your address space. But in the kernel that we are studying it just sets base and limit to 0, base to 0 and limit to $2^{32} - 1$. So, once again I have a completely flat address space right. So, if you would say I want to access address x , so let us say you fill in the value x into your `eip` register, it just goes into the physical memory at address x .

And as long as it is between 0 and M , you will see your valid byte; if it is greater than M , you will see an error right some kind of an exception will be thrown by the hardware. Similarly, `ESP` or any direct addressing, so these addresses can be generated in many ways through a register, through a direct addressing, through displaced addressing right all those things we have studied already right. In any case they all go through segmentation, and we saw that after enabling segmentation our address space is still the same 0 to M ok.

(Refer Slide Time: 03:12)



So, here is another figure. So, after enabling segmentation I had 0 to M, now the boot sector operates in this mode where only segmentation is enabled in 32-bit mode. And in that mode the boot sector, the boot sector was leaving in the first 512 bytes of the disk. So, the boot sector has code to load the kernel. The kernel lives from then sector number 1 till some value, and whatever that value is known to the boot sector the size of the kernel is known to the boot sector. And, so the boot sector loads that many sectors from the disk and puts them in memory.

While it is doing that, the address space is still 0 to M right. So, it picks up that kernel and sticks it somewhere here in this area right. We saw last time that it sticks the kernel at starting at address 1 MB right. So, it starts the it starts pasting the kernel starting at 1 MB and from there on it paste the kernel. And, recall that the size of the kernel was not very big either the size of the kernel was if I remember correctly definitely less than 1 MB right. So, you start you start the kernel from one M, you pick the kernel from that disk stick it at physical address 1 MB, and it will you know at most go till 2 MB right, all the kernel code and the kernel data.

Also recall that the kernel itself has been compiled with virtual addresses. So, all the symbols inside the kernel image have virtual addresses associated with them. So, if I say I want to branch to main, the address of main will be a virtual address right. It will be a virtual address in the sense that will be an address above this value 8 and seven 0s which is also called kern base. In our code, this called the kernel base right. So, all the symbols in our image have addresses above kern base, base or an above 2 GB.

So, all the symbols in that have addresses above 2 GB. So, anytime I dereference a symbol within my kernel code I am going to be trying to access a region above 2 GB. But in this address space, it is not possible right because assuming M is less than 2 GB, if I try to dereference a symbol in the kernel image, I will see an error, I will see an exception right, so not possible at this point.

So, the boot sector, so what the boot sector does it loads the kernel and it jumps to the first instruction of the kernel, and recall at the kernel was stored in elf format so the boot sector knows exactly where to start, and so the kernel the first instruction of the kernel get started. And the first instruction of the kernel leaves in this file called entry dot s

right. Now, the first few instructions of the kernel entry dot s are going to execute in this address space.

So, these instructions have to be very careful, they should not be dereferencing any kernel symbol, because all kernel symbols have addresses above 2 GB right. The moment it dereferences the kernel symbol, I will get an exception, not, it is not legal in this address space right. So, the first thing it does the entry dot s file does is it change changes this address space by enabling paging right. So, it enables paging and it uses a page directory called entry page directory right.

And we saw last time that this entry page directory implements an address space where the first 4 MB are mapped identically. So, if you access an address x within 0 to 4 MB, you will hit physical memory at address x right. On the other hand, it also maps this address kern base to kern base plus 4 MB to the same location 0 to 4 MB right. So, if you access address kern base plus x , and assuming x is less than 4 MB, then you will hit physical address x right, so that is how the entry page dir was configured.

And, so the entry dot s code just switches from this address space to this address space as soon as it enables paging and check sub CR3 point to entry page dir right. So, it points CR3 point to entry page dir enables paging and suddenly I am running in this address space. Recall that because the kernel itself all the instructions the entry dot s file itself was living in you know less than 4 MB space, so we set from 1 MB to 2 MB let say all the addresses you know the eip and ESP still remain valid, because the addresses from here are identical here 0 to 4 MB right. So, those remain valid.

You do not know it is not like you know I am standing on the ground and the ground has been taken away from it is not true because this as long as the kernel is less than 4 MB, the kernel safely mapped right, so the next instruction can execute alright. So, you enable paging, but ensure you ensured that the ground that I am standing on still stays as soon as I switch the address space right ok. But in this new table, there is an extra mapping and this extra this new this table is going to be used to switch from here to here right.

Recall that the kernel all the symbols in the kernel have values which are in this range. So, I should not be dereferencing any symbol from here. The only thing I am going to do is I am going to look, so in this there are some symbols likes there is the region that is allocated as stack right. So, we saw so let us look at the entry dot s file on sheet 10.

(Refer Slide Time: 08:39)

```
Aug 28 14:35 2012 xv6/entry.S Page 2

1050 orl    $(CRO_PG|CRO_WP), %eax
1051 movl   %eax, %cr0
1052
1053 # Set up the stack pointer.
1054 movl   $(stack + KSTACKSIZE), %esp
1055
1056 # Jump to main(), and switch to executing at
1057 # high addresses. The indirect call is needed because
1058 # the assembler produces a PC-relative instruction
1059 # for a direct jump.
1060 movl   $main, %eax
1061 jmp    *%eax
1062
1063 .comm stack, KSTACKSIZE
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
```

And we will see there is this instruction which is loading this value stack plus kstack size into esp. kstack size is a constant. Stack is the symbol that has been declared here right. So, kstack is the symbol in the kernel and recall I am saying that all symbols in the kernel have been compiled to have addresses in the kernel virtual address space which is above kern base right. So, the stack the value of stack will be some you know kern based plus something above kern base and so that plus kstack size, kstack size is let say you know 4096 bytes, so that variable is loaded into esp.

Recall that at this point the paging is already been enabled, so this address should be a valid address. I should be able to dereference esp, because I am operating in this address space right. In this address space esp will be pointing somewhere here. And so, when I dereference it, I will get right value right. So, I was able, so notice that the I have I have only started reading the kernel symbols after I enabled paging.

(Refer Slide Time: 10:05)

```
1072 .text
1073 .globl multiboot_header
1074 multiboot_header:
1075     #define magic 0x1badb002
1076     #define flags 0
1077     .long magic
1078     .long flags
1079     .long (~magic-flags)
1080
1081 # By convention, the _start symbol specifies the ELF entry point.
1082 # Since we haven't set up virtual memory yet, our entry point is
1083 # the physical address of 'entry'.
1084 .globl _start
1085 _start = VZP_W0(entry)
1086
1087 # Entering xv6 on boot processor, with paging off.
1088 .globl entry
1089 entry:
1090     # Turn on page size extension for 4Mbyte pages
1091     movl    %cr4, %eax
1092     orl     $(CR4_PSE), %eax
1093     movl    %eax, %cr4
1094     # Set page directory
1095     movl    $(VZP_W0(entrypgdir)), %eax
1096     movl    %eax, %cr3
1097     # Turn on paging.
1098     movl    %cr0, %eax
1099     NPTEL
```

Before I enabled paging, this code did not read any kernel symbols except that one kernel symbol that was read was entry page dir, but it was converted to its physical address before it was loaded into CR3. In any case CR3 is going to take a physical address. So, it is ok, and physical address is already mapped right. So, this piece of code in entry dot s is very carefully written. There is some amount of tricks involved in this kind of code.

And, this kind of tricks you can only see if you are actually looking at some kind of you know real code right. So, if you know if you find this interesting, you should probably go and single step and see exactly what is happening you know as soon as you turn on paging what addresses become valid, what remain invalid, what were valid earlier and have become invalid now, what were invalid earlier have become valid now etcetera, etcetera right. So, it is interesting ok.

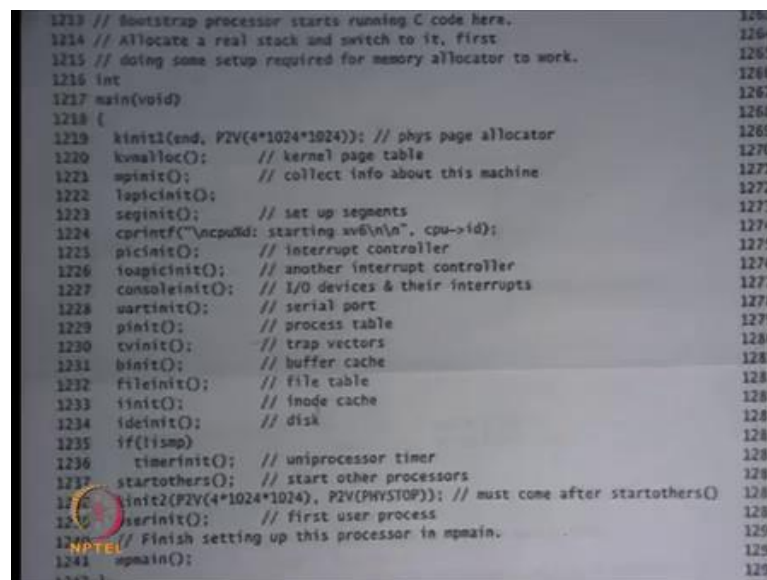
And then you basically, so at this point though you no need to any conversions from v to p right because stack having an address above 2 GB is a valid address, you already in the new address space. Similarly, main is a kernel symbol which will also have an address which is above kern base. And what you are going to do is you are going to move jump to main right. And the moment you do that, you have reloaded your eip with the kernel virtual address which is above kern base. So, you loaded esp with the kernel virtual

address here appropriately, and here you are going to load the eip with the kernel virtual address ok.

So, at this point, so far, my eip and esp were pointing somewhere here, I have reloaded esp to point here and it is standing on solid ground. And then I jump to main, and once again eip now points here. So, now both my esp and eip point here. And from now on I will basically be executing in this space completely because I have forgotten about all my addresses here; anything I dereference from the kernel will have addresses in this space.

So, from now on I just execute in the space right. So, from now on it is just normal plain c code that can execute you know just plain dereferencing should work right. But this kind of tricky code has to be written in assembly, because you know c does not understand this different address spaces and all that kind of stuff, it is very tricky the programmer has carefully done this very carefully done this alright. So, and so we said let us look at, so it is now going to jump to main.

(Refer Slide Time: 12:47)



```
1213 // bootstrap processor starts running C code here.
1214 // Allocate a real stack and switch to it, first
1215 // doing some setup required for memory allocator to work.
1216 int
1217 main(void)
1218 {
1219     kinit1(end, P2V(4*1024*1024)); // phys page allocator
1220     kmalloc(); // kernel page table
1221     mpinit(); // collect info about this machine
1222     lapicinit();
1223     seginit(); // set up segments
1224     cprintf("xv6\n\n", cpu->id);
1225     picinit(); // interrupt controller
1226     ioapicinit(); // another interrupt controller
1227     consoleinit(); // I/O devices & their interrupts
1228     uartinit(); // serial port
1229     pinit(); // process table
1230     tvinit(); // trap vectors
1231     binit(); // buffer cache
1232     fileinit(); // file table
1233     iinit(); // inode cache
1234     ideinit(); // disk
1235     if(tismp)
1236         timerinit(); // uniprocessor timer
1237     startothers(); // start other processors
1238     init2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
1239     userinit(); // first user process
1240 // Finish setting up this processor in mpmain.
1241     mpmain();
1242 }
```

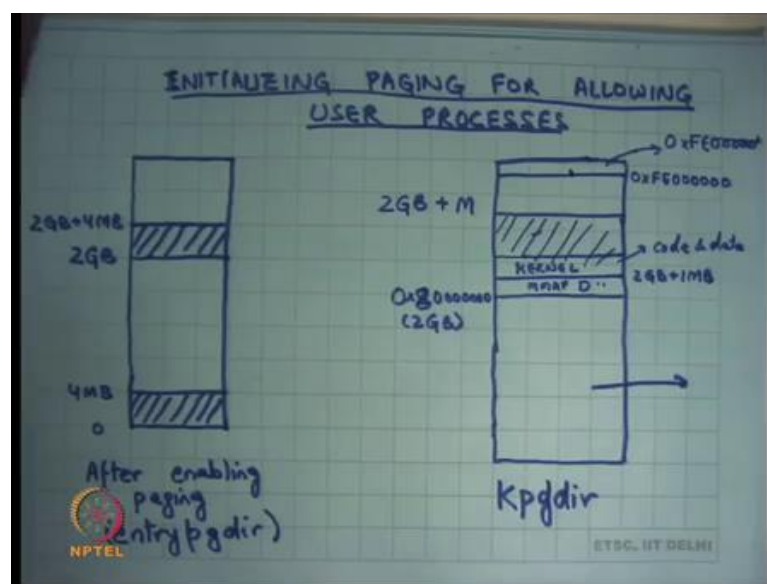
And, here is a code for main right. And main is this long function that you know that we do not necessarily need to go into full detail of. But the important thing to note here is that this is the first instruction that will get executed. And at this point the stack has been initialized correctly right with some size, stack size, kstack size. As long as these functions do not overflow the stack right, do not use too much stack space.

For example, they do not the function called depth is not very large in the above, they do not allocate local variable that are too large or too many local variables that that stack should be sufficient to implement all these to execute all these functions alright. So, the first function it calls this kinit1 one. And so, this is a physical page allocator alright. So, it is going to it going to initialize functions called k alloc and k free that allow you to take physical pages from the available address space. We are going to study this later. But let us just assume that this function works correctly.

And after this function has executed completely, the kernel can make calls to k alloc and k free right. K alloc is just like malloc for kernel. So, just you can allocate memory, and you can free memory except that k alloc only works at page (Refer Time: 14:10). So, malloc can take any size, but k alloc can only k alloc one k alloc call will allocate one page and give it to you ok. And then there is this function k v malloc which will initialize the kernel page table and that is what I am going to discuss today right.

So, we said that this is the page table after the entry page dir and what the kernel wants to do is that it wants to remove this paging this space and map this space completely into physical map address space right. So, right now only 0 to 4 MB region is mapped, you want to map entire 0 to M in this area right. And you want to remove that, so that they can be used for user pages right. Recall that was what the xv6 paging configuration was ok.

(Refer Slide Time: 15:00)



So, in this page kernel page dir, so let me call it K page dir. I want to switch from this address space to this address space, and so what I am going to do I am going to map right now it is just 2 GB plus 4 MB, I am going to map the entire physical memory from here right. So, I am going to say 2 GB plus M and this is entirely going to go to 0 to M in physical memory right, so that is what I want and how I going to do it we are going to look at the code later.

But let us just see what we want right, we want to map the entire physical memory here, also recall that the first 1 MB was reserved for memory map devices right. The first 1 MB of physical memory address space is actually not memory it is you know console and the other things. So, even that gets mapped, but that is just reserved for the memory map devices. Also recall that I loaded the kernel at 1 MB. So, so this is let us say M map let us say M map d memory map devices. Then there is some area which will be loaded for the kernel.

Recall that we had loaded the kernel starting at 1 MB of the physical address space. So, starting at 1 MB, you are going to have the kernel both code and data right. And all the other space I am going to call it free space right, and that is the space I am going to manage using k alloc and k free that is a space I will say this is space that you can use for your heap right or this is the this is the vacant space.

And this is the space that you can allocate for your data structures like the page directory right. So, where is the page directory can go to get allocated. Recall that the entry page directory was a global variable and so that was allocated in the data section of kernel. But all anything other than that, for example, per process page directories right, so all these things are going to be allocated from this extra space which is wherever the kernel ends and from there on till whatever capital M is the physical size of physical memory.

Also, this is the space allocate memory for data structures like process control box right. And most importantly this is the space from where you are going to allocate memory for the user processes themselves alright. So, what you are going to do is you are going to say k alloc going to get some page and you are going to create a mapping in this area for that. So, recall that is how it works basically allocate some pages here. Let us say a user says I want more page, or a user says I want to load a particular executable and that executable is larger than the current allocation of the process.

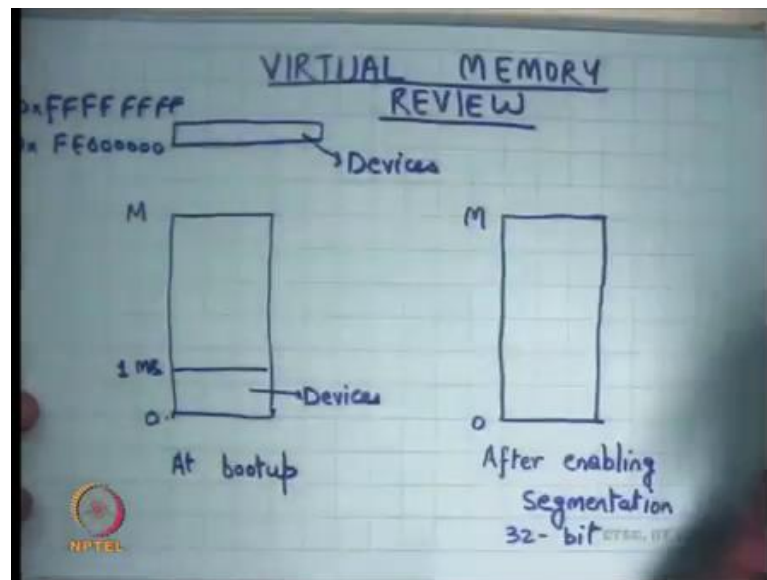
What is it going to do is, it is going to create mapping here it is going to allocate some space of here? Whatever space it allocates from here, it will have some backup in physical memory right some pointer into the physical memory. Some whatever pointer I get from allocation here I am going to convert it into its corresponding physical address, and then create a mapping from here to that physical address right, recall that the entire memory is been mapped here.

So, the entire memory has been mapped in my virtual address space, I allocate some area from that virtual address space, I get a page, I convert the address that I got to its physical address, and then I create a mapping in the user side of things to point to that right. So, these pages which are the mapped in the user side actually have two mappings in the page table: one on the kernel side and one on the user side right. So, if the two names for the same physical location, one name will be kern base plus something, and other name will be whatever the user want it to allocate it wherever it wanted to allocate it right, so.

So, I want to switch from this organization to this organization. Also point out that the top there is a slice of virtual address space on the top that starts at 0XFE and six zeros till 0XFF right. This slice of address space is actually used for memory map devices also. So, just like this area is used for memory map devices, this area is also used for memory map devices, and so this maps identically to the physical address space FE is same thing alright.

So, basically the physical address space in on a 32-bit X86 machine, the physical address 0XFE and six zeros is not actually pointing to memory, it is pointing to some memory map devices. And the kernel wants to retain that access. So, what it does is it just says the top virtual address space maps identically to its corresponding physical address space. So, if the kernel ever wants to access those devices, it can just access it using that address the same address right. So, basically, I am not going to use this top address space from FE000000 to FFFFFFFF for anything other than memory map devices right they map identically to physical address space right.

(Refer Slide Time: 20:59)



So, just to just to make this discussion more complete, at bootup the address space was not just this right. Actually at bootup the address space was 0 to 1 MB is devices, 1 MB to M is whatever your physical memory is, and also there is some chunk on the top FE e 2 3 4 5 6 to FF This area is can also be accessed. However, this will also point to devices right in the physical address space, so that was the original physical address space.

So, the amount of physical memory that you can have in your system is not really 2 to the power 32 minus 1, it is 2 to the power 32 minus 1 minus 1 MB minus whatever this is ok. And, so what the kernel wants to do is now retain this access and so it is going to map this identically to the corresponding physical address space right that is all. Anyways that is not really important, but when we are looking at code, it will help us in understanding what is going on ok.

(Refer Slide Time: 22:27)

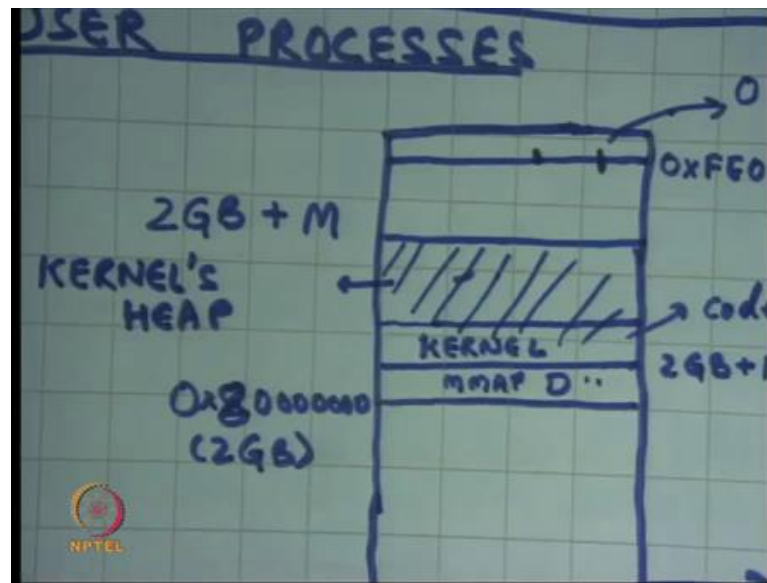
```
1753
1754 // Allocate one page table for the machine for the kernel address
1755 // space for scheduler processes.
1756 void
1757 kvmalloc(void)
1758 {
1759     kpgdir = setupkvm();
1760     switchkvm();
1761 }
1762
1763 // Switch h/w page table register to the kernel-only page table,
1764 // for when no process is running.
1765 void
1766 switchkvm(void)
1767 {
1768     lcr3(v2p(kpgdir)); // switch to the kernel page table
1769 }
1770
1771 // Switch TSS and h/w page table to correspond to process p.
1772 void
1773 switchvm(struct proc *p)
1774 {
1775     nptchcli();
1776     cpu->gdt[SEG_TSS] = SEG16(STS_T32A, &cpu->ts, sizeof(cpu->ts)-1);

```

So, let us look at what is kvmalloc doing on sheet 17. So, basically at the highest-level kvmalloc is going to initialize a new page table which is going to have this kind of mapping right, and it is going to switch to it. And it is going to forget about the old page table and that is it right. So, the kvmalloc function is just two lines. It calls the function called setupkvm; this is going to initialize a new page dir k page dir which will have this address spaces mapping.

And it is going to return a pointer k page dir to that particular page directory ok. So, it is going to initialize the page directory, where is it going to get the pages for to initialize for this page directory from its heap right from all this area that we discussed. This extra area: this is a kernels heap.

(Refer Slide Time: 23:24)



So, it is going to allocate a page directory from here, and it is going to initialize it, and it is going to return a pointer to the page directory ok. That that return pointer will be a virtual address in the kernel space right because everything in the kernel from now on is in the virtual address. When you allocate a page the return value that you get is also a virtual address. So, if you want to convert it to a physical address you need to subtract kernbase from it 2 GB from it right.

So, this function is going to allocate a page table and return a pointer to it this point is going to be a virtual address. And then I am going to call switch kvm. So, k page dir happens to be a global variable and. So, you just set up k page dir to this and switch kvm is just going to load cr3 with kpgdir except that it is going to call v 2 p on k page dir before it loads return cr3 perfect.

You allocated k page dir in your virtual address space initialized it, you got a virtual pointer you converted into physical pointer and loaded into cr3, recall that cr3 takes only physical pointers ok, so that is very simple. What we are going to look at next is setup kvm how exactly this page table is getting initialized alright. So, what is the structure of the page table.

(Refer Slide Time: 24:59)

```
1727 int perm;
1728 } kmap[] = {
1729 { (void*)KERNBASE, 0,          EXTHIGH, PTE_W}, // I/O space
1730 { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0}, // kern tex
1731 { (void*)data,      V2P(data),    PHYSTOP, PTE_W}, // kern dat
1732 { (void*)DEVSPACE, DEVSPACE,     0,      PTE_W}, // more dev
1733 };
1734
1735 // Set up kernel part of a page table.
1736 pde_t*
1737 setupkvm(void)
1738 {
1739     pde_t *pgdir;
1740     struct kmap *k;
1741
1742     if((pgdir = (pde_t*)kalloc()) == 0)
1743         return 0;
1744     memset(pgdir, 0, PGSIZE);
1745     if (p2v(PHYSTOP) > (void*)DEVSPACE)
1746         panic("PHYSTOP too high");
1747     for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
1748         if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
1749             (uint)k->phys_start, k->perm) < 0)
```

So, on the same page at line 1737, here is the code for setup kvm alright. And what it is doing is it first calls kalloc function to get a page from the kernel heap right. So, let us just assume that kalloc just allocates a page from the kernels heap and returns the pointer to it right, just like malloc, and puts it into page dir. Just in case the malloc failed or kalloc failed, then it just says oh I cannot proceed, it will return 0 and ultimately it will return 0 and tell the user that you know something has failed. Why could kalloc fail, give me some reasons?

Student: (Refer Time: 25:48).

You have run out of memory. So, let us say the physical memory was very small and so the head room that you have above the kernel space is very small. So, when you call the first kalloc, the first page itself got failed right.

Student: Sir.

Yes, question?

Student: Sir, how is the kalloc for (Refer Time: 26:03), right now we do not have a page directory or anything. So, from where will it allocate the page?

So, there was one function called k init that I skipped alright. So, I am going to discuss the later, but let us just assume that this address space has already been so this right now

we are working in this address space right 2 GB to 2 GB plus 4 MB. So, that is a great question. Assuming that there is some headroom above the kernel in this first 4 MB, I should be able to allocate a page right. So, it is not really limited by the size of physical memory at this point, it is actually limited by 4 MB this artificial limit that we used right, because you have only mapped the first 4 MB.

So, assuming that if the kernel size was so large if the kernel was you know let us say 3.9 MB, then my kalloc would have failed right irrespective of how much physical memory I have. So, if my kernel was indeed that large, I should have mapped more area here in my entry page dir, but my kernel is small right not that large let us say less than 1 MB. So, I have enough space.

So, when you to call kalloc, you can actually allocating space from here ok, because yeah you right at this point I am I am not switched I do not have a heap, my heap is actually very small. So, what you have done is he is just initialized whatever space is here to a heap and that is from that is where he is going to serve his request for kalloc alright ok.

So, so he is going to get a get a pointer in that kernbase to kernbase plus 4 MB space and that is going to be stored in page dir. The next thing it does is it zeros out the page dir. So, the pointer that he gets has a one-page size memory area that can be used right. So, page size is 4096 bytes, and recall that a page directory structure itself was one-page rights 4096 bytes. And so, what it does is just zeros out, so memset page dir zero-page size means zero out the entire page.

Student: (Refer Time: 28:08).

What does it mean to zero out the entire page directory, basically means no mapping exist right, because recall that for a mapping to exist the present bit in an entry should be set right? So, if you zero out the entire thing, none of the present bits are set, and so there are no mappings in this page directory initially ok. We can ignore this, this is just an error check, but then it is going to say for k is equal to k map k is less than this map pages right.

So, it is going to so k map is an array of mappings. It is going to look at this array, and it will call the map pages function to create mappings in the page table for that array.

What this array is k map is basically telling you what the regions are, that need to be mapped. For example, so it basically so static struct k map, it says this is the virtual address at which you need to map a region. This is the corresponding start physical address at which this region should be mapped. This is the end physical address. So, it also tells you the size of the region that needs to be mapped, and these are the permissions with which you should map it right.

So, for example, what we want to do is we want to say 0x you know this kernbase 2 GB value. So, 2 GB value to this 2 GB plus 1 MB should be mapped to 2 GB plus 1 MB with all privileges read-write execute. Then 2 GB plus 1 MB to whatever the kernel code is and read only data is that should be mapped to 1 MB to 1 MB plus whatever kernel codes kernel sizes in read mode. You do not want that the code should be writeable, just a just a precaution measure.

So, you just basically map it in read only mode. And then for everything else which is kernels data, and all the other memory that should be mapped in read-write mode in this area to the corresponding physical address right. And finally, this address 0XFE00 should be mapped identically to the same address in physical memory. So, those are the four sort of regions that you need to map right, four contiguous regions that you need to specify and that is what this array is telling you right.

(Refer Slide Time: 30:31)

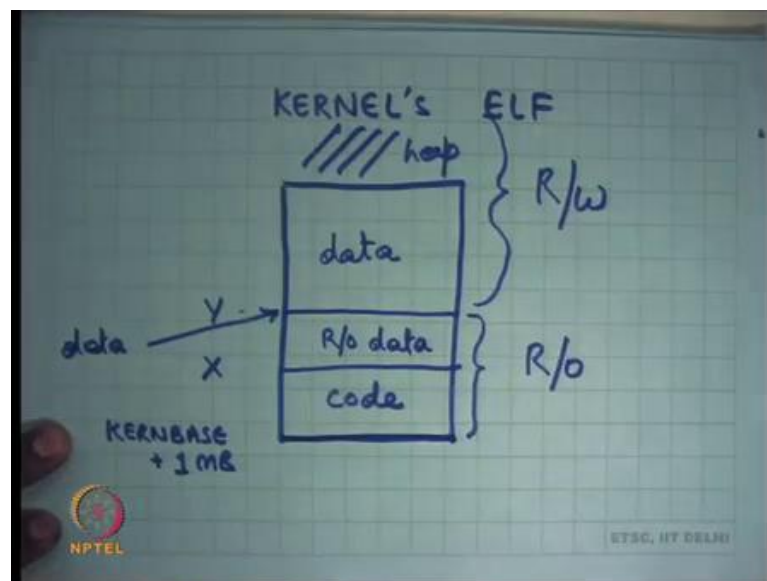
```
... // This table defines the kernel's mappings, which are present in
// every process's page table.
static struct kmap {
    void *virt;
    uint phys_start;
    uint phys_end;
    int perm;
} kmap[] = {
    { (void*)KERNBASE, 0,          EXTMEM,   PTE_W), // I/O space
    { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0), // kern text+rod
    { (void*)data,     V2P(data),    PHYSTOP,  PTE_W), // kern data+mem
    { (void*)DEVSPACE, DEVSPACE,    0,       PTE_W), // more devices
};

// Set up kernel part of a page table.
pde_t*
setupkvm(void)
{
    pde_t *pgdir;
    struct kmap *k;
    if ((pgdir = (pde_t*)kalloc()) == 0)
        return 0;
```


So, the first mapping is saying start mapping at kernbase which is 2 GB. Physical address 0 to physical address 1 MB EXTMEM is 1 MB with write permissions alright. It is just initializing the IO space saying kernbase to kernbase plus 1 MB map it to physical address 0 to 1 MB with write permissions, this is my IO this is my IO space memory map devices.

Then the next thing I have is starting at kernlink. So, kernlink is what, it is going to be the place at where you will start the kernel, it is 2 GB plus 1 MB ok, and V 2 P kernlink is 1 M b. So, recall that we have loaded the kern kernel starting at address 1 MB, the boot set at loaded the kernel starting at address 1 MB. So, we are going to load the kernel starting at 1 MB at address 2 GB plus 1 MB ok. The size the endpoint of this particular segment will be determined by where the data starts.

(Refer Slide Time: 31:49)



So, the kernel has been organized the kernels ELF has been organized in such a way that this is kern base plus 1 MB that is the start address. It will first have some code whatever the size of the code is and so somewhere the code will finish let us call that position X. Then there will be some area which will be called read only data ok. You can specify read only data. For example, in C, if you say const something then declared the global variable that gets allocated in the read only data right.

So, let us say you know data goes at Y, and then everything else is let us say data right. So, there are pointers in the kernel when you compile the kernel, there are symbols in the

kernel which tell you that here is the data, here is read only data and so on. So, this Y is pointing to so that is where the data point is pointing. So, what, what the loader is going to do is it is going to consider this as one segment and map it with read only permissions right. And it is going to map this and everything above it. So, above it is basically heap. After you load it above this is heap. So, everything above is going to be mapped with read-write privileges right ok. So, that is what that is what is happening here.

(Refer Slide Time: 33:25)

```

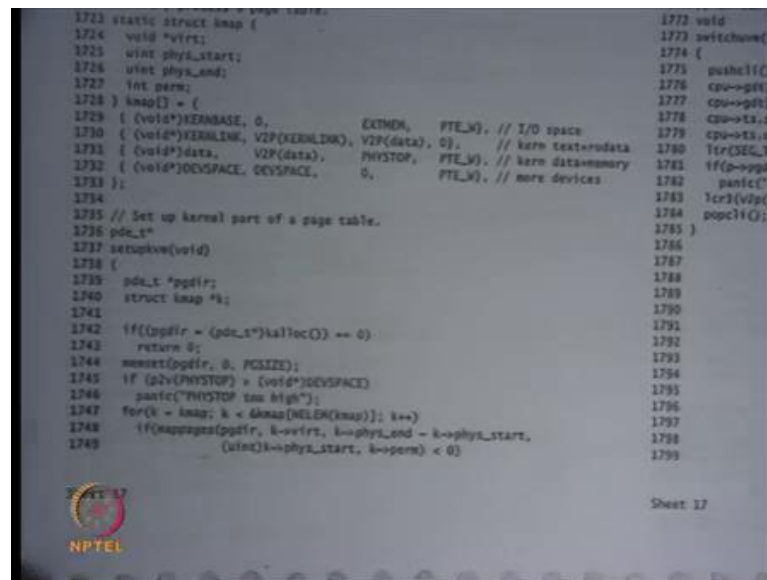
1713 // for the kernel's instructions and r/o data
1714 // data.KERNBASE+PHYSTOP: mapped to V2P(data)..PHYSTOP,
1715 // 0xf0000000..0: mapped direct (devices such as iocpts)
1716 //
1717 // The kernel allocates physical memory for its heap and for user memory
1718 // between V2P(end) and the end of physical memory (PHYSTOP)
1719 // (directly addressable from end..P2V(PHYSTOP)).
1720 //
1721 // This table defines the kernel's mappings, which are present in
1722 // every process's page table.
1723 static struct kmap {
1724     void *virt;
1725     uint phys_start;
1726     uint phys_end;
1727     int perm;
1728 } kmap[] = {
1729     { (void*)KERNBASE, 0,          EXTHIGH, PTE_W, // I/O space
1730     { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0}, // kern text+rodata
1731     { (void*)data,      V2P(data),    PHYSTOP, PTE_W, // kern data+memory
1732     { (void*)DEVSPACE, DEVSPACE,     0,       PTE_W, // more devices
1733 };
1734
1735 // Set up kernel part of a page table.
1736 pde_t*
1737 setupkern(void)
1738 {
1739     pde_t *pgdir;
1740     struct kmap *k;
1741
1742     pgdir = (pde_t*)kalloc() -- 0;
1743     return 0;
1744     memset(pgdir, 0, PGSIZE);
1745     if (p2v(PHYSTOP) > (void*)DEVSPACE)
1746         panic("PHYSTOP too high");
1747     for (k = kmap; k < kmap+NMAP; k++)

```

Kernlink V2P kernlink which is 1 MB to V2P data right; so, wherever data starts till that point you map it with zero permissions 0 mean read only permissions in this case alright. If it is writeable, then you say PTE W, if 0 that means, read only. And then starting at data map it from at V2P data all the way till phystop, phystop is the size of the physical memory let us say alright or the maximum size of physical memory that you will support.

All that all the way up to phystop, you map it with writeable permissions, so that is kernel data plus all the other memory that will be used as a kernels heap right. And then you map devspace to devspace which is just FE00 identically right. So, that is what you are going to do, you are going to read this array which has these mappings contiguous mappings in a nice readable way.

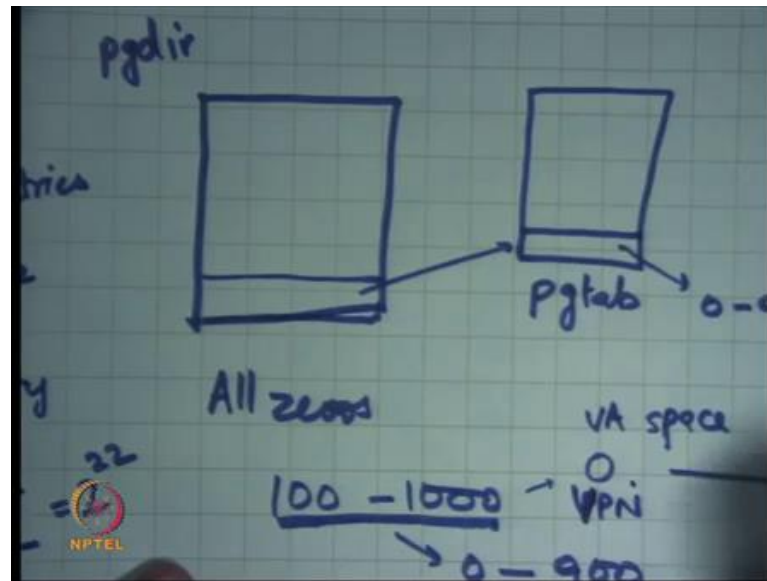
(Refer Slide Time: 34:27)



And for each of these mappings, you are going to call map pages right. So, what map pages is going to do is just going to create these mappings. Map pages takes an argument a virtual address, the corresponding size which is given by phys end minus phys start in the k map structure, the corresponding physical address at which it should be mapped and the permission with which it should be mapped alright.

So, it first takes the page directory in which it should create the mapping, it assumes that the page directory should already created, the virtual address, size, physical address permissions. And, if it succeeds and it will return a nonzero a positive value greater than equal to 0 value, otherwise return a negative value. What are some reasons for why it can fail? Recall that to map pages it may need to create allocate more pages right.

(Refer Slide Time: 35:31)



Right now, you are basically having a page dir. So, this is the page dir that is initially completely 0 doubt, all zeros. Now, let us say I want to say I want to map address 100 to 1000 to physical address 0 to 900. This type of the particular example right; so, I want to map virtual addresses 100 to thousand to physical addresses 0 to 900.

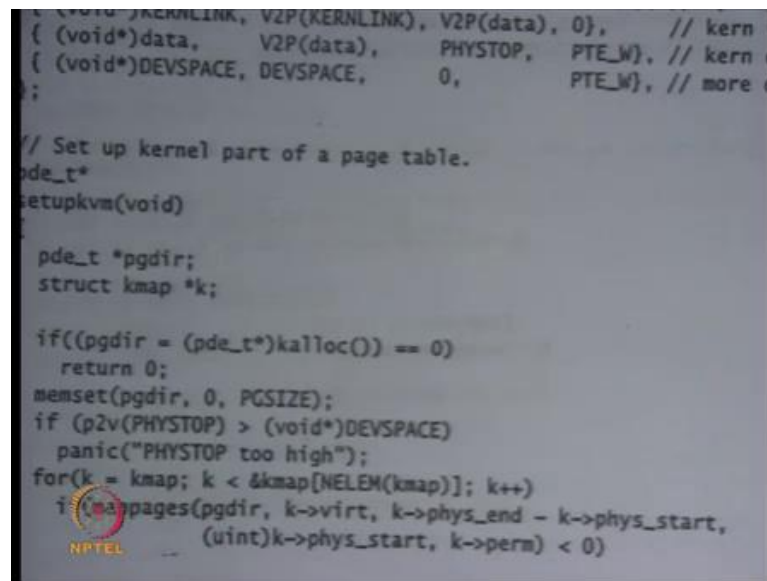
What do I need to do I will say address 100 is represented by entry number 0 right, each entry here represents a 4 MB region right because these are 2 to the power 10 entries mapping a space of 4 gigabytes right. So, each entry, 1 entry is mapping 2 to the power 32 divided by 2 to the power 10 which is 2 to the power 22 address space which is 4, 4 MB right.

And, recall that this 4 MB could be mapped using large pages or it could be mapped using another level of indirection using a page table right. So, in this case 100 is going to be this, the first entry within the first 4 MB. You are going to allocate another page table here; it is called as page table. And, in this you are going to say each entry going to allocate 4 kilo bytes.

So, here you going to create an entry saying map to 0 to 900 right. Clearly, I, I cannot just create this mapping, I need to say what is the corresponding page pages and so I can only do this at page (Refer Time: 37:22). So, instead of 0 to 100 to 1000, I will probably have to say map page 0 in VA space to map page 0 in PA space. So, this is just going to map to 0 in this case by the corresponding (Refer Time: 37:47).

So, to be able to do this, to be able to, so for this kind of mapping, I need to potentially allocate a new page to store this mapping right, and so this allocation could potentially fail. So, page directory is the top-level structure. Recall that a page table is a two-level structure. And so, whatever I want to map I may need to create allocate pages for the second level. And so, if that allocation fails, if I am running out of memory, I can probably I can probably fail so that is one reason for example, why map pages could fail ok.

(Refer Slide Time: 38:27)



```

{ (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0}, // kern
{ (void*)data, V2P(data), PHYSTOP, PTE_W}, // kern
{ (void*)DEVSPACE, DEVSPACE, 0, PTE_W}, // more
};

// Set up kernel part of a page table.
pde_t*
setupkvm(void)
{
    pde_t *pgdir;
    struct kmap *k;

    if((pgdir = (pde_t*)kalloc()) == 0)
        return 0;
    memset(pgdir, 0, PGSIZE);
    if (p2v(PHYSTOP) > (void*)DEVSPACE)
        panic("PHYSTOP too high");
    for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
        if (map_pages(pgdir, k->virt, k->phys_end - k->phys_start,
            (uint)k->phys_start, k->perm) < 0)

```

Yes. So, if I want to map let us say 0 to phystop in one go, I will divide it in two chunks and create entries in the page table, because recall that I cannot just the paging system does not allow me to map all at once. I have to you know divide it into either large pages or small pages, and then create appropriate entries in the page table to create that mapping ok.

Student: Already corrupt those pages.

So, the question is that let us say I want to map an entire region from data to phystop in the page table right. Question is how much space do I need to map this? The amount of space that I need to map this is basically whatever these spaces divided by

Student: 4 KB.

4 KB. If you are using small pages right, so that is not that much. So, let us say this space was, so the overhead is basically you know less than 1 percent, or less than 0.1 percent actually right. So, whatever this space was let say this space was x bytes, then the amount of space that your required to make that those x bytes is x divided by 4000 roughly right, so that is less than you know 0.1 percent of x .

Student: But when I map those in the page directory means that I have already also have a backup in my physical memory for those mapping.

Could you repeat?

Student: Like if I map it from my data to physical stop.

Ok.

Student: My in my page directory and it means that in my physical memory also I have allocated space for that particular memory only.

You have mapped that memory; you have mapped that memory. You are not, so there is a difference between mapping and allocation. You have just mapped that memory which means this name is going to refer to this location. You have not allocated that memory right. Allocation means you are going to you have basically committed to using it. You have just mapped it you are going to use it later on, ok. The memory that you have allocated is the kernels code, and kernels read only code and kernels data a kernel's read only data and kernels data.

All that has been already there are some contents into it which are useful which should not be overwritten. So, all that memories mapped, but that is a small piece. Everything above that you have mapped not necessarily used, you are going to use it later. And you are going to use it using `kalloc` and `kfree` functions, you are going to manage that using `kalloc` and `k free` functions.

Student: Sir, so we do not need to store that mapping means that mapping will require some space.

Which mapping?

Student: What should do physical?

We do not need to store that. So, can you say your question fully?

Student: I am saying that we need some space to store that mapping that this space number in virtual will point to this page number in physical

Sure.

Student: So, how do we do that?

So, how do we say that this page number in virtual space maps to this page number in physical space. We say that using a page table entry right. So, one entry in the page table basically says that this page maps to this page right. And so, an entry of 4 bytes gives you information about mapping for a region of 4 KB. So, it is you know point one percent overhead or space in that sense alright.

So, what map pages is going to do is, it is going to fill in the structure called page dir such that this area in virtual address space gets mapped to this area in physical address space ok. And it is going to do it using small pages alright. So, recall that I had said that you know one common optimization used in mainstream kernels is that you use large pages to map the entire kernel right that save space, but xv 6 does not do that I mean it is not so xv6 just to make thing simple uses small pages to even map the kernel address space. So, all these regions are going to be mapped using small pages right.

(Refer Slide Time: 42:53)

```
// (directly addressable from end..P2V(PHYSTOP)).  
// This table defines the kernel's mappings, which are present  
// every process's page table.  
static struct kmap {  
    void *virt;  
    uint phys_start;  
    uint phys_end;  
    int perm;  
} kmap[] = {  
    { (void*)KERNBASE, 0,          EXTHEN,   PTE_W}, // I/O s  
    { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0}, // kern  
    { (void*)data,     V2P(data),    PHYSTOP,  PTE_W}, // kern  
    { (void*)DEVSPACE, DEVSPACE,    0,        PTE_W}, // more  
};  
  
// Set up kernel part of a page table.  
pde_t*  
setu (void)  
{  
    pde_t *pgdir;
```

0xE0000000
224MB

In this case for example, if I wanted to be smart about it, I could have probably said look this area is 1 MB, so I am going to use small pages to map this area because you know 4 MB pages are too big to map. This area, this area from kernlink to V2P data, this is also very small I said it is less than 1 MB. So, even this is it does not make sense to use large pages. I am going to use small pages for this, but this area from V2P data to phystop and assuming my phystop is very large right like you know 100 MB or 200 MB.

Then this area is for potentially mapable using large pages right, but we are going to forget about it, we can say we are going to map all these areas using small pages right. And finally, this area is also small, so large pages may or may not help yeah, I mean perhaps large pages will help here too, but in any case, we are going to use small pages to map all this.

Student: Sir, phystop maps the end of (Refer Time: 43:46).

Phystop is just a constant defined in the kernel which basically says what is the maximum amount size of physical memory, that xv6 supports alright. So, phystop is just setup to hexadecimal E 3, 4, 5, 6. I think E and six 0s which is 224 MB right. So, it just irrespective of the size of the physical memory it maps data to phystop identically ok.

So, if the physical memory was larger than phystop those that area is not accessible right, above phystop area in the physical memory will not be accessible. If the physical memory was less than phystop let us say the physical memory was only 100 MB, then you have just unnecessarily mapped extra area, but that is I mean if the user ever accesses that area is going to get some error right, some exception ok.

Let us stop here.