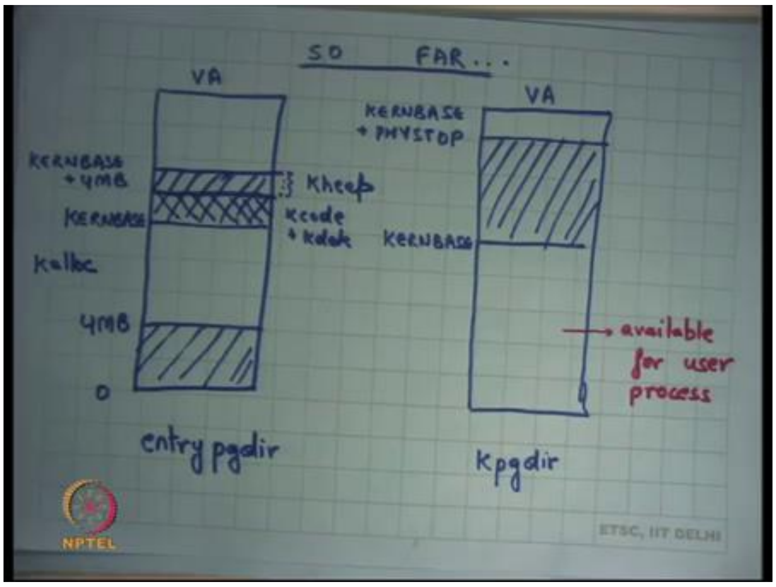
Operating Systems

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

Lecture – 16
Processes in action

Welcome to Operating Systems lecture 16.

(Refer Slide Time: 00:29)



So far, we have been looking at the virtual address space using paging right. And, we saw that when the operating system boots up, it boots up in the physical address space then it enables segmentation. Then it enables the 32-bit mode without paging in which case still it has the physical address space it can access the physical address space and at some point, it enables paging right.

And, the first page when it enables paging for the first time, it enables it using this page directory called entry page dir right and at the entry page dir you have an address space which is look something like this from 0 to 4 MB, you are mapped to physical address 0 to 4 MB. And, from KERNBASE to KERNBASE plus 4 MB which you know which is starting at the which is where the kernel is mapped. Let us call them these call these addresses the high addresses.

So, at the high addresses starting at the high first high address to the 4 MB high address, they are again map to the same. So, the both these regions are aliased to the same physical memory area right in the entry page directory. And, we saw that this is a nice way of doing things because this allows you a very smooth transition from a flat; from a address space which only had this much right and then so it the initialization code still runs in this space assuming that the kernel itself its within 4 MB. And, then at some point the kernel moves it stack and itself the instruction pointer in this space right.

And from then on it will want to always execute in the high addresses leaving the low addresses available for the user process right. So, eventually you will want to move from the entry page directory, which was just a temporary thing so, something like this where you have mapped the physical memory in the kernel address space. So, starting from KERNBASE to let us say the maximum physical address that that the kernel; suppose is phystop. So, you will map KERNBASE to KERNBASE plus PHYSTOP here; map to the physical address and the lower addresses will be available for user process right. So, the user can map its code here, map its data here and so on, yeah yes question.

Student: What happens to the entry page directory when the kernel page directory is created?

When the kernel page directory is created what happens to the entry page directory ok? One answer is that only in the entry page dir this remains, and you know; so question is does the entry page dir get deallocated right or does it keeps consuming the space that it was consuming earlier right. After all entry page dir is living in physical memory, it is consuming some space right.

Initially I had some use for it; so, I moved to entry page dir and I used it for something and then I have moved to k page dir and so what happens to the entry page dir? Do I de allocate that space and reuse that for something else.

Student: Sir, we could have used the entry page directory for the kernel page dir from; like we could have overwriting the phys entry page directory which is turning with the kernel page.

Here is the suggestion how about just overwriting entry page directory with the contents of kernel page directory and that; that would have saved some space alright. So, based on the codes we have seen so far for xv6; what do you think? How was entry page directory declared? It was a global variable right. It was a global variable which means it is taking some static space and it can never be deallocated right. Also it has a fixed size and its contents are sort of also you know initialized in some way, you can overwrite them and one way to overwrite them could have been that you just overwrite them with pointers right.

On the other hand, kp page directory was not a global variable in fact, it was allocated from the kernels heap. In fact, it was allocated from this region from KERNBASE to KERNBASE plus 4 MB. So, anything that you allocate from heap; the nice thing is you can de allocate it and you can start using it for something else. In the code that we have seen for xv6; entry page dir was not allocated of the heap right because by the time the entry page there was needed, the heap was not even initialized.

So, you use the entry page dir has some static array, global array. So, that much space is actually going to get wasted after entry page dir is not used right in the case of xv6 at least right. You may you if you really care about this amount of space which is you know 4 kilobytes, then you would maybe want to you know allocate entry page dir also of the heap and then deallocate it so you can use it for anything else you like later on or you could use it reuse it 4 kpgdir; all these are options.

You know one has to at some point make a tradeoff between coding simplicity and efficiency and so it is to just say that you know let us basically allocate this to a global variable; later I am not going to use it, but does not matter alright ok. So, this is the k page dir and yeah there is another question.

Student: But could not just we have changed the PTE underscored p flag to 0 and that was it could have been deallocated automatically.

The question is could not I have just changed the PTE underscored p flag to 0 and that would have deallocated it automatically. Well, I think there is some confusion between mapping and allocation right. So, mapping basically means that I have mapped, and I have created a mapping between a virtual address and a physical address that is what mapping means. Allocation means I have some bookkeeping to do, so I have said that this amount of space is being used already and this amount of space is not being used; so that is just bookkeeping right.

So, mapping and allocation are two different things so, but when I say I am going to free in a memory area; I am basically saying I want to deallocate it; un mapping it is not going to deallocate it right. When I want to deallocate it basically means that if I want to allocate something else; I can use reuse this space that is what deallocation means. So, that that space should be reusable that is what the allocation deallocation means right; mapping un mapping is something different alright; so, this is kpgdir alright.

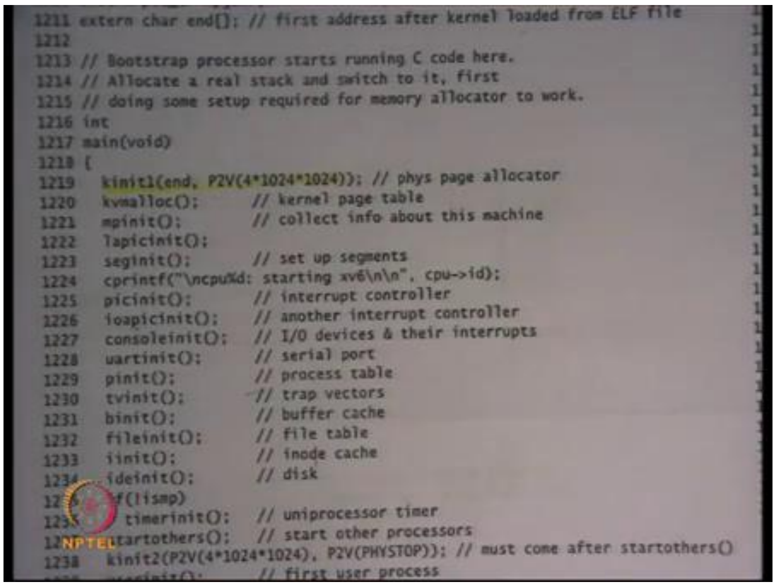
Now, I will point out a few things one interesting thing here recall that most of our programs assume that on the null address or 0 address is an invalid address right. If you ever dereference in a null address that basically means it is an error right; that is a convention right.

So, you cannot dereference the 0 address; another convention is when you say malloc and if it returns 0, basically means that the allocation failed. It basically means that the address 0 should really be invalid; if address 0 was valid then you know this kind of a mechanism will not work. So, typically the convention is you know initially at this point notice that 0 is a valid address actually. So, at this point if malloc; you know if this if this area was part of the heap and malloc return 0, it may actually not be an error it may actually be giving you an address which is 0 right, but at this point 0 is not mapped.

So, at this point you know; so as a convention typically, operating systems do not mapped the zeroth page; just as a convention right that makes your coding interface simpler because you just saying that one particular address called 0 is not going to be a valid address; that is a convention that is being used across the software stack right. If the operating system ever violates that convention, then you know a malloc could actually turn a 0 as a valid page and that would; that would violate my conventions right.

So, as a convention you basically say that at the 0, I am not going to map anything. So, even when the operating system is going to map the user pages; one simple convention that it follows as it is not going to map the zeroth page alright. So, and we were looking at xv6 and we said let us look at the code which basically does all this alright.

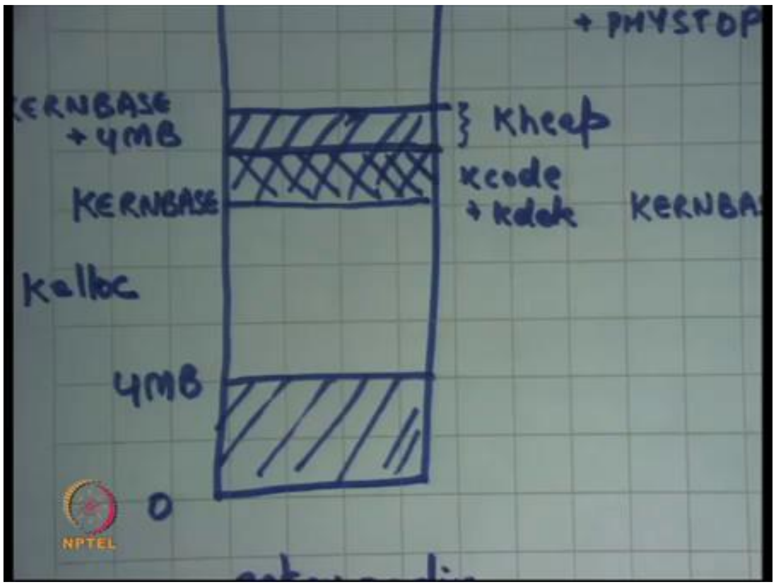
(Refer Slide Time: 08:58)



```
extern char end[]; // first address after kernel loaded from ELF file

// Bootstrap processor starts running C code here.
// Allocate a real stack and switch to it, first
// doing some setup required for memory 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
   f(!ismp)
    timerinit();   // uniprocessor timer
   tartothers();   // start other processors
  kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
  [illegible]      // first user process
```

So, let us look at let us look at the code which transitions from entry page dir to k page dir and we said that this function called kvm. So, firstly this function called kinit 1 initializes the physical page allocator alright.

(Refer Slide Time: 09:20)



So, what does it do? It basically says now let us say you know at this point I am executing in entry page dir and what I am going to do is I am going to inish and let us say some area of this 4 MB is used for; this is let us say used for the kernel code plus data k code plus k data right. And all this area above it is going to be used as kernels heap; let us call it k heap right. And this is where you are going to allocate; you are going to create you are going to use this space to allocate space for k page dir for example, alright.

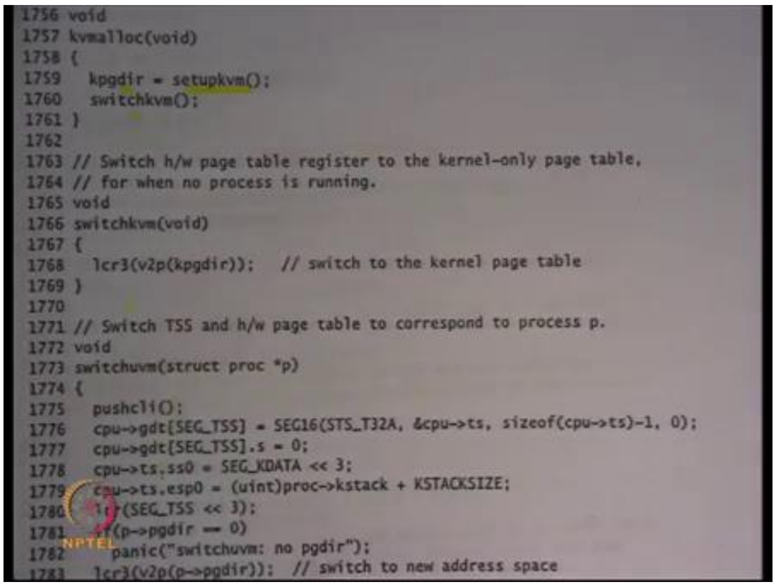
So, this is the area from where you are going to allocate pages for k page dir. So, before you can do that you need to initialize the space in some way and so you are going to initialize some data structure which is going to be which is which is going to be called the free list and you are going to add all these pages in this area to the free list ok. And so, once you do that; you can use these functions called you know k alloc.

So, once again this is just bookkeeping; so you have you have a you have an internal data structure called free list and k alloc is going to take a page from that free list and return it to you and k free is going to push it back to the field free list alright. So, this statement here this function called kinit 1 is just initializing the heap starting at the kernels end. So, end is basically wherever the you know it is initialized the; so, symbol end itself is initialized by the linker that you know in the in your compilation stage.

So, the end was a symbol that was given to you by the linker which basically marks the end of the code and data of the kernel. So, that is the; that is where the heap starts where the code end data end and till 4 MB right P 2 V of 4 MB. So, KERNBASE plus 4 MB that is what this means. So, end to 4 MB is where you are going to initialize the kernels heap and so what you going to do is kinit 1 is going to push all this to some free list. And, now you whenever you say k alloc is going to allocate a page from that area and you say k free is going to push back a page into that area available for us right.

And so, kvmalloc is going to initialize k page dir and inside kvmalloc; it is going to call k alloc to basically allocate these pages for the page table right alright.

(Refer Slide Time: 12:02)

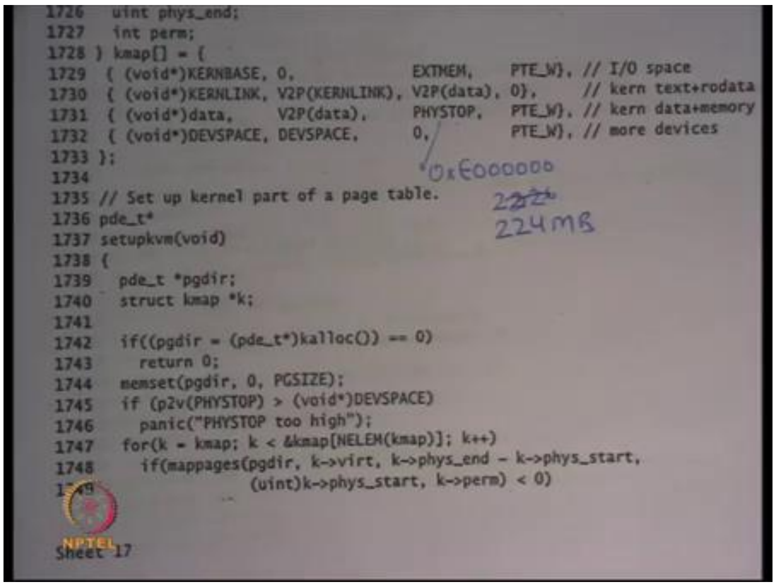


```
void
kvmalloc(void)
{
  kpgdir = setupkvm();
  switchkvm();
}

// Switch h/w page table register to the kernel-only page table,
// for when no process is running.
void
switchkvm(void)
{
  lcr3(v2p(kpgdir));   // switch to the kernel page table
}

// Switch TSS and h/w page table to correspond to process p.
void
switchuvm(struct proc *p)
{
  pushcli();
  cpu->gdt[SEG_TSS] = SEG16(STS_T32A, &cpu->ts, sizeof(cpu->ts)-1, 0);
  cpu->gdt[SEG_TSS].s = 0;
  cpu->ts.ss0 = SEG_KDATA << 3;
  cpu->ts.esp0 = (uint)proc->kstack + KSTACKSIZE;
  ltr(SEG_TSS << 3);
  if(p->pgdir == 0)
    panic("switchuvm: no pgdir");
  lcr3(v2p(p->pgdir));  // switch to new address space
```

So, now let us look at kvmalloc in that sheet 16; we were looking at it last time but let us look at it again sheet 17 ok. So, kv malloc does nothing it for it allocates a page table in the virtual address space and assigns the pointed of it or the page directory to k page dir.

And, then it calls switch kvm which switch kvm does nothing, but loads the physical address corresponding to the k page dir which is just whatever k page dir is minus kern base and puts it into cr 3 right that is what that is what kvmalloc is going to do; it is going to allocate a page directory and the switch to it.

(Refer Slide Time: 12:41)



```
  uint phys_end;
  int perm;
} kmap[] = {
 { (void*)KERNBASE, 0,             EXTMEM,    PTE_W}, // I/O space
 { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0},     // kern text+rodata
 { (void*)data,     V2P(data),     PHYSTOP,   PTE_W}, // kern data+memory
 { (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;
  memset(pgdir, 0, PGSIZE);
  if (p2v(PHYSTOP) > (void*)DEVSPACE)
    panic("PHYSTOP too high");
  for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
    if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
                (uint)k->phys_start, k->perm) < 0)

Sheet 17
```

Let us look at what happens when you allocate a page directly. So, let us look at setup kvm. So, this is the code for set of kvm; it basically allocates a page of the heap right recall that k alloc which going to give me a page from between end and 4 MB right end and V 2 p 4 MB is going to give me a page from there.

The address of the page will be a virtual address I am only dealing in virtual addresses at this point. If for some reason the k alloc failed you just say you know there is a failure; I am running out of space and this can happen if your kernels code plus data was actually very large it was close to 4 MB let us say right.

Then you memset page dir to 0 which means you say that none of the entries in the kernel page directory at present and then you say you know you do some sanity check that PHYSTOP should be greater than DEVSPACE let us ignore this. And then it iterates over this array called k map which has which contains information about the regions of kernel which need to be mapped and then calls map pages on those regions right.

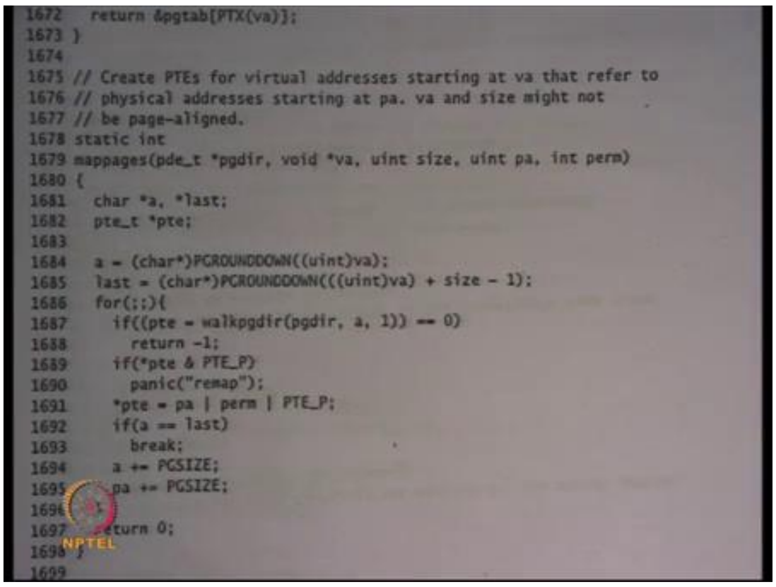
Last time we looked at the different regions that were mapped; we said you know KERNBASE. So, the first at first feel of this is the virtual address at which you need to map; the second field is the physical address. So, the virtual to physical address mapping is KERNBASE gets mapped to 0; EXTMEM is the size. So, EXTMEM is basically representing 0 to 1 MB space which has all the devices in it and all that.

So, that is mapped with permissions writable then KERNLINK is mapped to V2P KERNLINK to. So, this is kernels text and read only data which is mapped with read only privileges. Then from data to V2P data, data to PHYSTOP, you map it as writable privileges. So, all the write writable data and the kernels heap is in this segment from data to PHYSTOP; we right data to PHYSTOP data to data plus PHYSTOP basically what data do V P 2 V PHYSTOP; PHYSTOP is expressed as a physical thing it is a size; so you basically have it have to right.

And finally, there is this extra thing for DEVSPACE which is basically to keep the upper map memory mapped devices for yeah. So, we can safely ignore this right and so what this loop is going to do is going to look at each of these entries. So, it is going to iterate over the entries of k map and for each entry, it is going to create mappings in the page table right.

So, map page is going to create a mapping in page directory starting and virtual address virt of size phys and minus phys start right; phys and virt is the third field and phys start was the second field starting at physical address phys start with permissions perm right that is what map page is going to do. And we already know what map page is going to do its going to fill in. So, we have initialized a 0; a completely 0, 0 out page dir I am going to fill in these entries some of those entries are going to be non-zero from now on right. Also, I am going to use only small pages to do this.

(Refer Slide Time: 15:46)



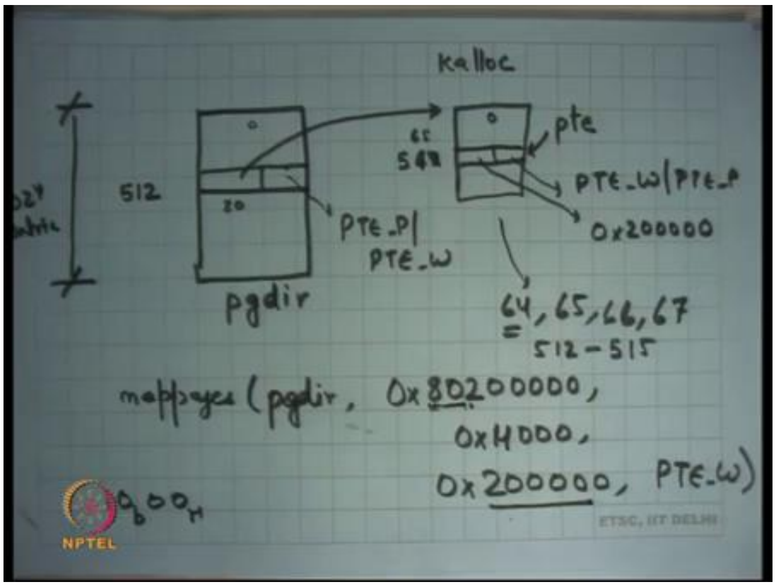
```
  return &pgtab[PTX(va)];
}

// Create PTEs for virtual addresses starting at va that refer to
// physical addresses starting at pa. va and size might not
// be page-aligned.
static int
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
  char *a, *last;
  pte_t *pte;

  a = (char*)PGROUNDDOWN((uint)va);
  last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
  for(;;){
    if((pte = walkpgdir(pgdir, a, 1)) == 0)
      return -1;
    if(*pte & PTE_P)
      panic("remap");
    *pte = pa | perm | PTE_P;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}
```

So, this is map pages. So, I am going do use only small pages to map this address space and so not only do I have to fill up the page directory; I have also have to allocate in the second level page tables right and so and point the page directory to the second level page tables and fill those page tables appropriately also right that is what I am going to do.

So, this is simple map pages takes a page directory, a virtual address, size, physical address and permissions and it creates these mappings. Internally, it calls the function; so, it basically says where do I have to start? I start at you know you will round whatever the virtual address is you round it down to the nearest page boundary and that is where you start. And this is where you end you say VA plus size minus 1 and you round it down to its page boundary and that is where you end. And these are the pages that you need mean to map; starting from a to last right. And then you basically say let us walk the page directory with this address a which means you look at you; you say look at a, you look at the top 10 bits of a; index using that value into the page directory and so on right in the next take 10 bits into the page table and so on right.

And based on that you get a page table entry right and the and then you basically set up the page table entry to point to the physical address that you want to say with their right permissions and the present flag on alright and then and you and then you keep doing this right.

(Refer Slide Time: 17:13)



So, basically if I want to draw this what is happening is, I started with a page directory which was completely 0 right; all everything is zeroed out. And then I said let us say map pages, page dir let us say I wanted to map 0 x 8 0 2 right to physical address and size let us say right; let us say size of hexadecimals 4000 and starting at physical address and let us say permissions are writable right.

If I wanted to do this; what I m going to do is I am going to say 8020000 this is the virtual address; let us look at the top 10 bits of this address. So, the top 10 bits are going to be you know the first two hexadecimal characters make up 8 bits and the 2 bits of the third one make it you know the top 10 bits and so that is that basically comes up to this end the 512th entry let us say right; it has 1024 entries and all. And this particular address is basically saying that the 512th entry needs to be mapped.

At this entry I am going to see it is a 0 right; so, nothing has been mapped here. So, what I am going to do is I am going to say k alloc and I am going to create a new page. I am going to allocate a new page right from the heap that you have seen before same heap we are going to use k alloc and I am going to point the top 20 bits to the physical address of this page right and I am going to set the permissions here to present and writable right. So, allocated a page a point I converted its into its physical address put it in its 20 bits and also zeroed it out. So, it is also initially 0 completely right.

Then I am going to look at the next 20 bits of this address and so the next 20 bits are going to be let us say.

Student: Next 10.

Next 10 bits sorry I look at the next 10 bits which are what 0 1 0 0. So, 1 0 in binary and let us say 0 0 in hexadecimal which is right; so that that will be probably be it and so that is 60 into 4 16 into 4 let us let us say this is 64th entry maybe I am wrong, but let us just assume that is the 64th entry here.

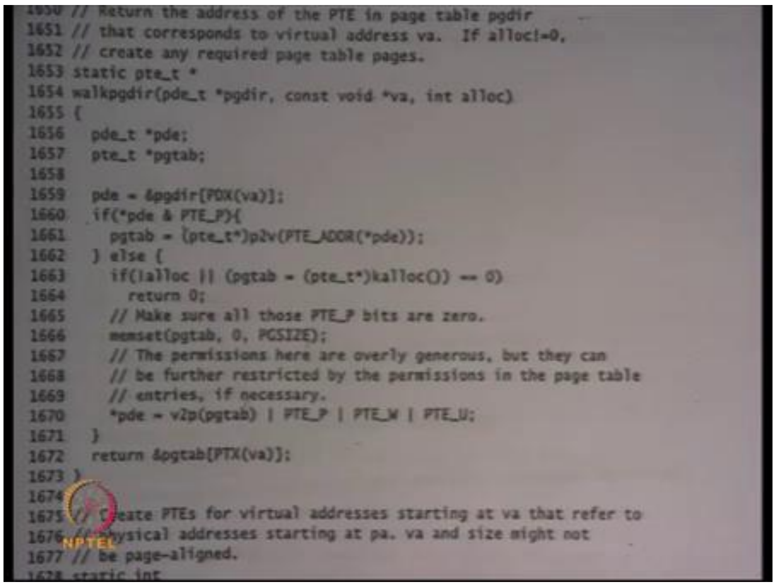
So, everything else is going to be 0 with a 64th entry; the pointed to the 64th entry is will be called the page table entry pointer. And I am going to set that to point to what? The top 20 bits are going to point to this address right 0x200000 and the flags are going to say PTE W and PTE P right and that is it. So, you created a mapping as desired from 80200, but you also have to do it for all these pages. So, how many pages there these are? These are to the part 12; these are 4 pages right. So, you have to do it for entry number 64, 65, 66 and 67. So, you have to create 4 entries in the PTE right.

So, you allocated one-page table page and you created 4 entries in the page table page to create this mapping. So, in this case the entries 64, 65, 66 and 67; I am going to be created I am assuming that 64 is the correct number I have not done the calculation, but wherever it starts ok. Somebody says its 512 good. So, if this is 512 then it is going to be 512 to 515 that will get created in this ok. So, that is what map pages and is going to do for you.

And we can now look at the code and convince ourselves that it is doing exactly what we wanted. So, walk page dir returns a pointer to the page table entry that needs to be filled up and you just fill up the page table entry with the physical address.

So, in other words walk page dir is going to return a pointed to this entry and you just fill it up with this value. So, walk page dir is going to create return a pointer to this particular entry and you are going to fill it up with the physical address and the flags right. So, that is what it is doing walk page dir returns a pointer to PTE and you just fill it up star PTE is equal to pa and permissions and present flag and you keep doing this right.

(Refer Slide Time: 23:02)



```
// Return the address of the PTE in page table pgdir
// that corresponds to virtual address va.  If alloc!=0,
// create any required page table pages.
static pte_t *
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
  pde_t *pde;
  pte_t *pgtab;

  pde = &pgdir[PDX(va)];
  if(*pde & PTE_P){
    pgtab = (pte_t*)p2v(PTE_ADDR(*pde));
  } else {
    if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
      return 0;
    // Make sure all those PTE_P bits are zero.
    memset(pgtab, 0, PGSIZE);
    // The permissions here are overly generous, but they can
    // be further restricted by the permissions in the page table
    // entries, if necessary.
    *pde = v2p(pgtab) | PTE_P | PTE_W | PTE_U;
  }
  return &pgtab[PTX(va)];
}

// Create PTEs for virtual addresses starting at va that refer to
// physical addresses starting at pa. va and size might not
// be page-aligned.
static int
```

Walk page dir what is it doing? It is just doing exactly what we did on paper right which is just walk the page table. So, it indexes the page directory using the top 10 bits of the virtual address, looks at that entry if that entry is already present then it gets a page table address and it returns the PTE address by in looking at the next 10; entries bits of the virtual address.

If it is not present already then it allocates it then it uses k alloc to allocate a page for the page table itself and initialize it to 0 and returns the corresponding entry.

So, this is called lazy allocation of the page table; you did not up allocate the page table entirely upfront, you allocated it lazily as in when you were mapping the virtual addresses you started allocating the page table pages. And that is the reason a two-page two level page table helps right that is the whole reason why a two level page table helps because you do not have to allocate the entire page table up front. What is the size of a full-page table?

Student: 4 MB; 4 MB.

4 MB because they are going to 4 kilobyte 4 k page tables 4 k second level page tables and each page table will be 4k in it in size right and so actually it is going to be more than 4 4 MB it s going to be 4 k into 4 k that 16 MB right. So, the sorry there going to be there going to be 1 k page second level page tables each of them 4 k; so that is the only 4 MB of second level page tables plus 4 kb of the page directory.

So, 4 MB plus 4 kb, but you do not need to allocate upfront you do it lazily depending on what virtual address is get map. Assuming that the size of the virtual address is that get that get mapped as small then you are going to basically save some space alright; so that is fine ok.

So, this walk page dir function is basically doing in software what the hardware would have done on every memory access right. The hardware does the same thing actually when you say a virtual address you looks at the top 10 bits, indexes the page directory then index is the page table and so on except that there is one extra complication that you if the page table is not present here you are also allocating it lazily right.

So, walk page dir is just a software counterpart about the hardware would do to walk the page table at runtime ok.

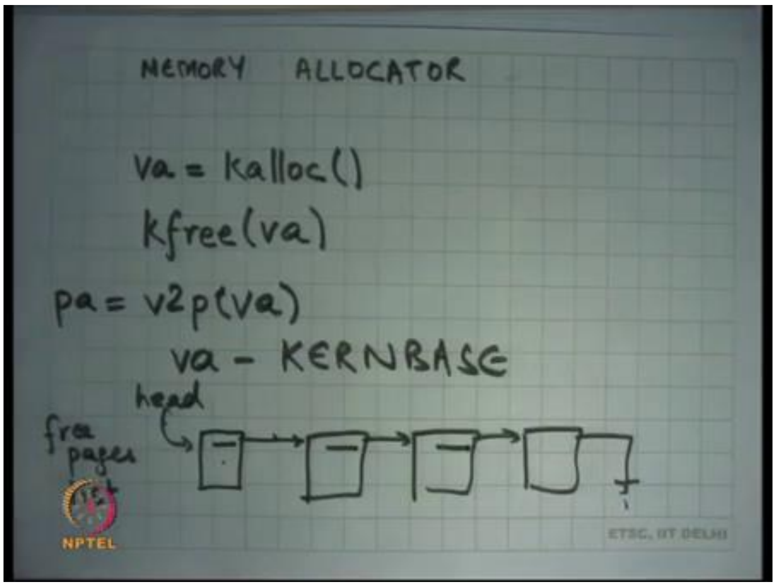
Student: Sir, walk page status walking just the first level not the second one.

It is walking both levels.

Student: Sir when it is not going (Refer Time: 25:27) entry.

See it is actually. So, it is de referencing the page directory to get the page table and then it is returning the appropriate point offset inside the page table as a page table entry right. So, it is it’s not; it is not dereferencing the page table entry, but it is getting the address of the page table entry and then sending it back as a return value and so the caller as supposed to dereference the page table entry. So, it is walking both levels alright good.

(Refer Slide Time: 26:06)



So, now let us look at how the memory allocator works right. So, how does the memory allocator work? Recall that we have been using this function called k alloc and what it returns with to me is the virtual address right.

And then I have I there is another function that I am using that is called k free which takes a virtual address and freeze it right. In this case, now this is similar to malloc and free that you may have used in your programs; there only difference is that k alloc does not take a size, it just allocates one page right. Also the other thing it has is it does not it always returns an address that is page aligned; it does not just allocate a page anywhere, it allocates an page at an address that is page aligned right.

So, whatever is the return value of k alloc; the last 12 bits of that address will always be 0. Why does it why do you think a kernel needs an allocator that always returns a page aligned address?

Student: For allocating this.

For yeah, so for a lot of the things that a kernel needs to do; it needs to allocate pages right and then create mappings for them in the page table right. For example, it wants involved pages for the page table themselves the page table recall had to be page aligned right. Similarly, if it wants to create pages for the gives the process it will need to create mappings for it in the page table. So, for a lot of things you need page aligned addresses and so you know it is easier to basically just have an alligator that always release gives you page aligned addresses right ok.

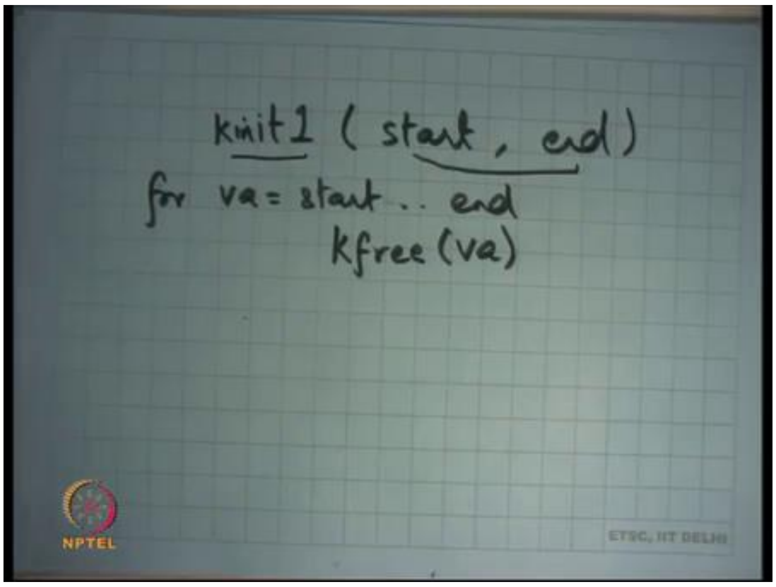
So, these functions always give you virtual addresses and if you ever wanted to convert them to physical address; all you need to do is say v 2 p of v a is going to give you the physical address right and v 2 p is nothing, but v a minus KERNBASE; that is a nice thing about the organization that we have I just sort of follow them from that.

So, how does, how this typically be implemented? Well basically the internally the kern maintains a list of free pages, just a linked list of free pages right. And then it is; it is pointed to by some head pointer and when you say alloc it just takes the first entry from here and updates head to the next point here and just returns that address right and when you say free it just adds it to him. Also notice that the next pointer itself is stored within the pages themselves right; you do not need extra space to store the next pointer because these pages are not getting used anyways. So, you can just use the space within them to store the next pointer itself ok.

So, this is you know this is roughly how malloc and free also work except that malloc and free need to be more complicated because they need to allocate variable sizes of data. So, you need to worry about fragmentation issues, but if you if you are just doing the fixed size allocator, all you need to do is just maintain a list and use the next pointer itself is going to be stored in the list ok. And so what is going to happen is that initially k when you said kinit 1 adds all these pages to the free list basically what it does is it just calls k free on all the add pages present in the space right.

So, just keeps calling k free on all the pages in this space, recall that this is the kernel heap and so all these pages need to be added to the free list. So, one way to of adding all these pages to the free list is just called keep calling k free on all these pages right and so all are them get added to the free list and now you and now your allocator and deallocator work using k alloc and k free.

(Refer Slide Time: 30:33)



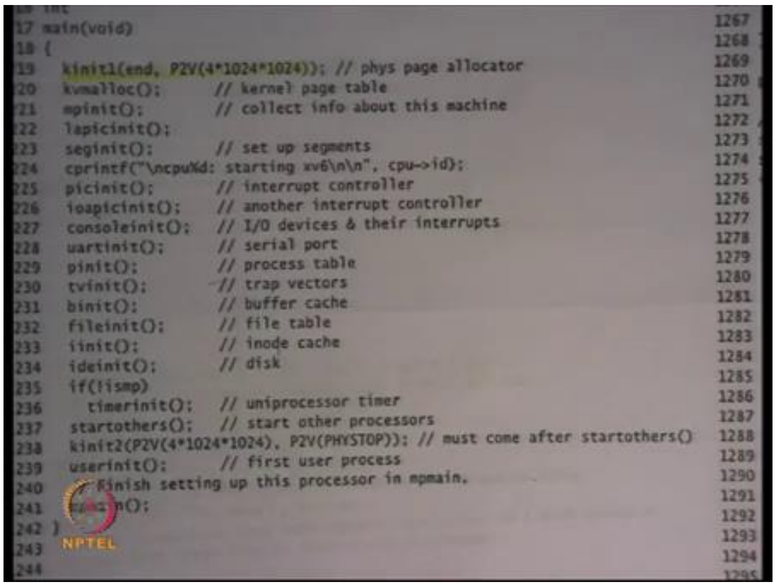
kinit1 (start , end)

```
for va = start .. end
        kfree (va)
```

So, basically k init; the function k init 1 simply just you know takes a start and an end address and just you know iterates over this address and each pay that it sees in that address you just adds it to the free list using k free. So, it just says k free v a for and let us say for v a from start dot end right. Now of course, I am simplifying some things here start and end need to be page lined addresses and all that right. So, if you look at the look at the implementation of k init 1, you are going to roughly find something like this right.

Let us look at k; let us look at the calls to k init 1. So,; so k init 1 free adds all the pages starting at end till P 2 V of 4 MB and adds them to the free list right that is what k init 1 is going to do and at this point you have some free list so that you can satisfy some k allocators and. So, kvmalloc and all the functions here are going to use the space which is within the 4 MB. So, notice that all these functions are going to be using only the heap which is lower than 4 MB right, any allocation all the space above 4 MB in the physical address space is not being used at all at this point right.

(Refer Slide Time: 32:06)



```
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();
}
```

Then there is another function called k init 2 that finally, makes available the other physical memory also and what it does is it just says free all the memory starting at 4 MB P 2 V of 4; 4 MB which is you know KERNBASE plus 4 MB to P 2 V of PHYSTOP.

So, add all these pages from starting at 4 MB till PHYSTOP also to the free list and at this point you know your k alloc will also start taking pages from space above 4 MB ok. Why is there a must come after start others? Well there is a diff; so, k init 2 relies on some you know initialization of locking subsystems etcetera. So, k init 2 requires; so because xv6 is a multiprocessor operating system; the same memory allocator serves requests to all the CPUs and so there needs to be some locking inside the allocator right because if two peoples simultaneously access it there should there should be some issues.

So, we are going to look at it in more detail, but for now let us just assume that k init 2 requires to initialize the locking subsystem and to be able to do that it needs that other processor should have started alright and after it has allocated all this memory; so it at this; so why PHYSTOP? Is it ok to say PHYSTOP? What if; what will happen if the physical memory, so PHYSTOP is just a constant right in the in xv6 code and PHYSTOP just is a constant which is around 240 MB right. What if the amount of physical memory on the machine was less than PHYSTOP? Is it to do this?

So, how many say no ok? 5 people. How many say yes? 0 others are undecided ok. So, well it is not right because recall that the free list is actually get there, there is data that is being written in the free list also right the next pointers and so you need to dereference those values to be able to write and anything on there that and if there is no physical memory there then dereferencing about generator generate an error an exception right.

So, in a way xv6 is lamely assuming that the amount of physical memory will always be greater than you know PHYSTOP. On the other hand, if the amount of physical memory was greater than PHYSTOP; is that a problem? No, that is no problem because you just using less than the amount of available memory right. Recall that you could I mean has a difference between mapping and allocation.

Even though if the amount of physical memory was less than PHYSTOP, I said it was to map addresses right because I just created a mapping; as long as I do not dereferences those mappings it is ok, but once I start using them that is the problem right. And I can only use those places if I add them to the heap right that is only that is only way, I can sort of start using them and then de referencing them and all that.

So, it was its not to use a memory less than PHYSTOP if you are because of the statement here for xv6 yes.

Student: Because if we consult incorrect address then make a hardware (Refer Time: 35:30) as just calculate modulo the after size and get (Refer Time: 35:36)

So, question is if you generate an indirect in incorrect address incorrect physical address right. So, they are two different things one is the program can generate an incorrect virtual address which means that the address was not mapped in the fulfill virtual page right. So, there is a certain type of exception that is gets thrown that that time, but if the page is actually mapped and the physical address is actually computed, but when you actually go on the bus and ask the memory for that physical address; the memory says oh I do not have that physical address. So, what happens right. so, it is an exception that gets generated ok.

The alternative that was suggested by you is why can kern the address get wrapped or anything of that sort no I mean wrapping is not possible assuming that you are only dealing with 32 bit addresses and this is 32 bit bus there is no wrapping that can happen really alright.

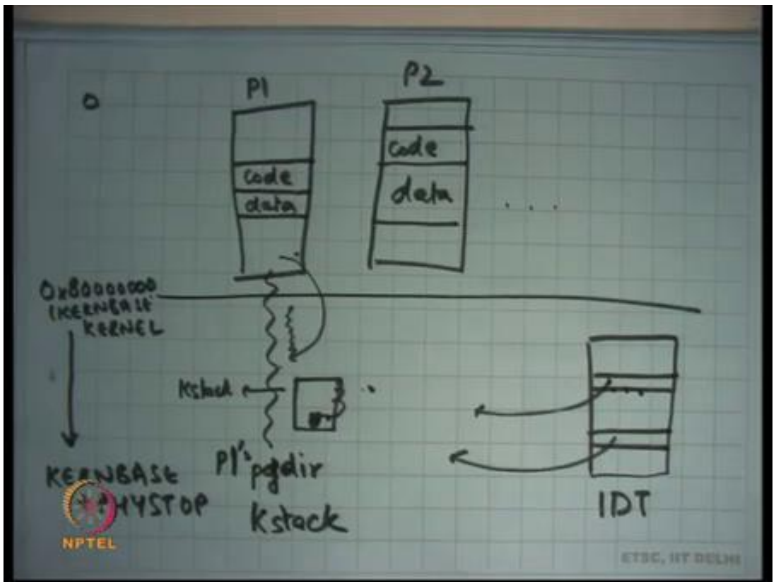
And after that I am going to call user init which is going to initialize my first process and after that my processes are going to run just like we have talked about in the Unix world, where they are the first process is going to get started that is the init process. And, the init process has some you know hard coded code in it which says let us say for the first process. And, let us say the first process is init process folks the shell process and the shell process basically takes commands from the standard input and depending on what the command is it may 4 more processes and so on ok.

So, that is how it typically works that is how it typically works in Linux also there is one init process that starts up and it then reads some files and depending on that it starts up some processes that should be started up at the beginning, some of those processes are going to be let us say the login process the x window system or whatever right.

So, notice that the first process is unique because that is the process that is created by the kernel, all other processes are not created by the kernel they are forked by already existing processes that is the only way to create new processes right. This is the first process that gets created by the kernel and that is what the user and it is going to do ok. So, I am going to look at user in it in the next lecture, but before we do that let us review how this how the system is going to run from there on right.

So, we have initialized everything, we have mapped entire physical address space if started the first process; the first process is going to make system calls and I have also initialized my device of system. So, they are going to be timer interrupts and all that kind of time right and so let us look at you know a sample execution of what is going to happen what is going to keep happening actually right.

(Refer Slide Time: 38:15)



So, so firstly, let us say this is the kernel and this is process P 1 which is the first process P 2 let us say it forked another process and so on right I am I am drawing it inverted. So, let us say this is address 0x8 right or KERNBASE right and so these addresses from here to here which is let us say KERNBASE plus PHYSTOP will be magnified the kernel right and everything from 0 to KERNBASE is the user side of things.

So, a kernel will have let us say its own code somewhere here and we will have some data each process right and similarly he may have its code here and this data here and they will have some heaps and all that hm.

And also there will be something called an Interrupt Descriptor Table we have seen this before IDT that is going to have pointer to which space right; they are going to have pointers to handlers and all pointers will be in the kernel space right. So, let us say P 1 wants to make a system call it uses a software instead of instruction; the IDT gets consulted the system called handler gets called the, but the system.

So, in the context of the same process in the same page directory; so, while this process is executing a certain pay directory has been loaded into the hardware; when it makes a system call you switch from user to kernel when you use the same page directory. So, in other words you are basically using the executing in the context of P 1 at this time. We are executing in the kernel, which is sheared across all processes video always exists, but at the time of the system call itself you are executing in the context of the process that made the system call. Because why because you are using the same page directory you have not change the page directory right; we just change the instruction pointer.

Also, so you know you are using P1 page directory and you are using P 1's kernel stack. Assuming it is a process model each process has a different kernel stack. So, you are running one P1's k stack right and how does the hardware load the P 1 k stack? Using the task state segment that must have been set by the OS maybe we know that alright.

So, it is going to execute in P1's k stack it is when it executes in P 1 k stack, the first few things it is going to do is its going to save all the registers of the executing process in the k stack itself right. So, let us say this is the case stack; the first few registers are saved by the hardware namely c s, e i p, s s, e s p, e flags right. The first five registers are saved by the hardware and the next few registers you can save in software which is which is how xv 6 will do it and this is how most operating systems will do it. It will just say push this is just to push that, register though this is a have not be modified. So, you know the values are still preserved and you just keep pushing in them on the k stack.

Once you have pushed all those registers you have saved the state of the user process, we are going to execute some logic on behalf of the process which may involve some privileged operations. And then you are going to say let us return back right and returning is going to be just pop off those values that you had saved on entry one by one, some of those values are going to be popped off in software. And, the last five values are going to be popped off in hardware by using the i rate instruction right and you are going to execute back in the process mode, user mode.

How does the process give arguments for the system call? One way that it just sets up its resistor values to indicate the arguments right it. In fact, sets up the resistor value so also indicate the system call number and how does the kernel give a return value to the system call?

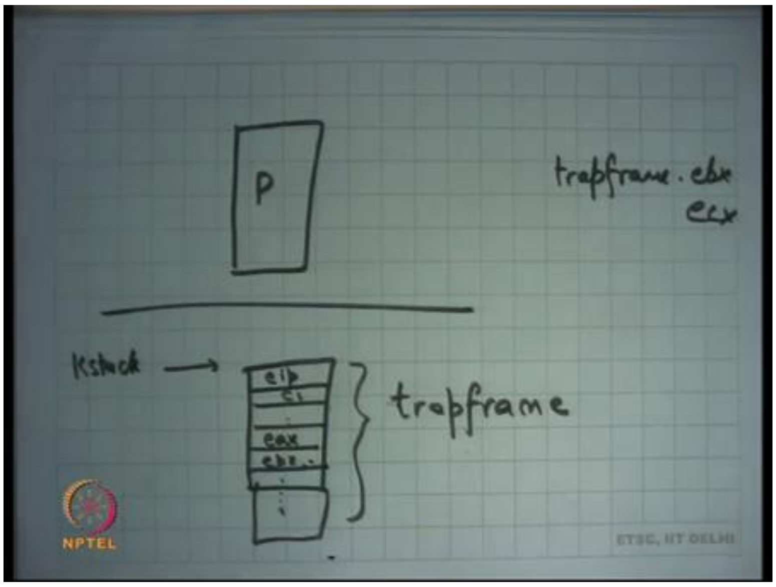
Student: Using the eax register.

Using the eax register so, but what does it need to do here?

Student: Modify.

Modify the saved value in the stack. So, whatever the state value in the stack of the; at the place where eax is living, you modify that. So, at the time of return it is going to get popped off into the eax register and the user is going to see the new eax value right. It does it is not good to just change your own eax because you have own eax is going to get lost when you return to P a. You need to change the eax that is saved in the stack at the right offset and so when you are going to turn back that eax is going to get visible to the user alright.

(Refer Slide Time: 43:41)



So, this structure; so let us say this is a process P when it executes here it starts at k stack right it saves eip, cs and so on and it also saves other resistors like eax and software ebx and so on right. This structure which lives on the stack is called the trap frame; while I am executing in the kernel, I can look at the trap frame to look at my arguments and also return a return value right.

So, the first thing a kernel does is create at the entry the first thing the handler will do its going to create this trap frame. The first five entry of the trap frame have already been created by the hardware, the next whatever number of entries depending on the number of registers in the architecture; you going to create them in software by hand and now you going to execute the logic.

So, for example if you wanted to look at your arguments you are going to look at the trap frame and look at the value stored and so let us say you know trap frame dot ebx create a contains argument number 1 to the system call right ecx contains argument number 2 edx contains. So, it is not the real register that is containing it is the save register which is in the trap frame that contains the arguments of the system call and that is where you also write your value to give a return value to the system call ok.

Student: Sir, what is the upper limit on the arguments directly (Refer Time: 45:18).

What is the upper limit on the number of arguments that can be passed to a system call? That is defined by the OS designer right and clearly the number of arguments cannot be unbounded because you know you have a finite amount of kernel stack. Typically, you know on kernel on Linux you would have you know at most 5, 6 arguments if you wanted more arguments then you will use that use pointer chasing to do that alright. So, it is possible that one of these arguments is actually a pointer that points into some structure which lives in this address space ok.

And so, the kernel can look at this pointer, dereference it and get more arguments from there right. And, your lab 2 is actually going to expose you to this kind of argument passing from the user to the kernel for system causing is going to implement this argument passing ok. So, on a system call to control move from here to here; the trap frame gets set up, the kernel executes on behalf of the process in the same address space at the process and then at some point it returns and the return is just unwinding of the stack at some point you are going to unwind the trap frame.

And, unwinding of the trap frame means you just load the registers where the trap frames values and then you call irate and you are back in the user space. The code the logic can read the trap frame and right to the trap frame for arguments and return values. Same thing happens if there was an external interrupt; let us say there was a timer interrupt that goes up right.

So, if there is a timer interrupt that goes off, that fires then the execution moves from user space to kernel space the some call the trap frame gets saved, some logic gets executed. Let us say the logic that gets executed is the logic or scheduler that at this point decides that you do not want to let this process continue running and you want to switch to another process; in which case you will switch the address space, switch the kernel stack and unwind it from there ok.

We are going to discuss this more next time.