Introduction to Operating Systems
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

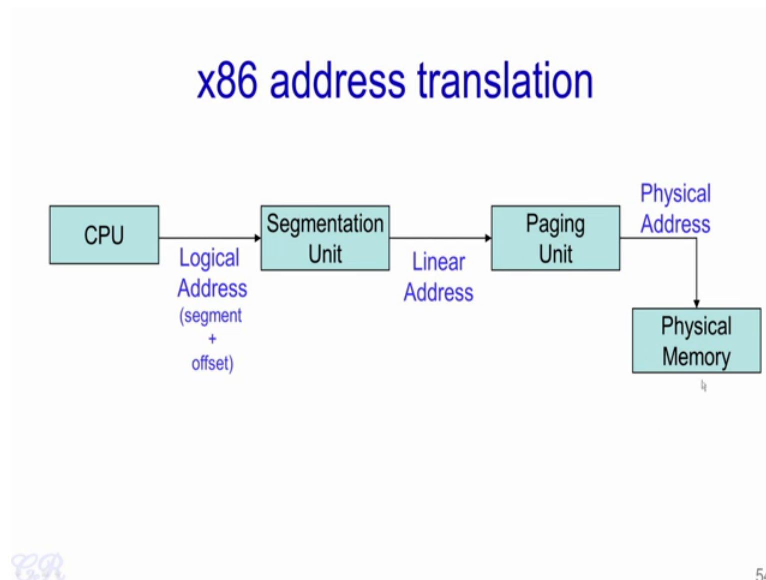
Week - 02
Lecture - 09
Memory Management in xv6

Hello. We had seen in previous videos about memory management schemes in Processors and about Virtual Memory and Segmentation, and we also had seen about how memory is managed in x86 systems.

(Refer Slide Time: 00:30)

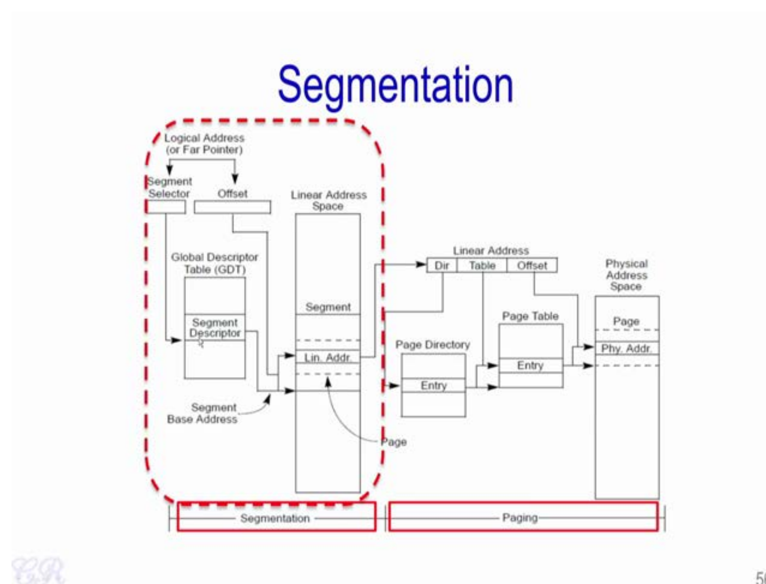
In this video we will look at Memory Management in xv6. So, xv6 is an operating system which is targeted for x86 platforms, therefore the video corresponding to memory management in x86 processors would be important. So, we would be referring lot of xv6 source code in this particular video, therefore for reference you could actually look at the xv6 source code booklet revision 8 which it is can be downloaded from this particular website.

(Refer Slide Time: 01:05)



Now, just to recall in an x86 system, there is two levels of memory translation. The CPU puts out a logical address which comprises of segmentation plus an offset. Then, there is a segmentation unit which converts the logical address into a linear address, and paging unit which converts a lead linear address to the physical address, and only then the physical memory or the RAM is actually accessed.

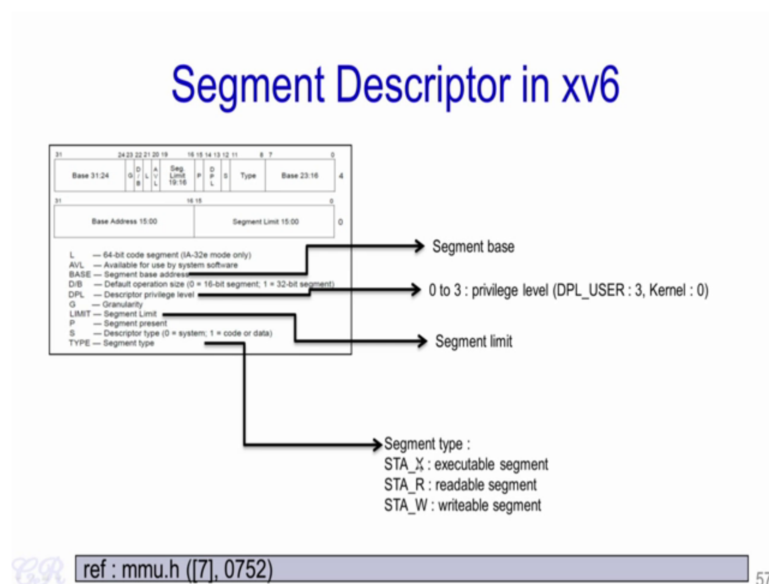
(Refer Slide Time: 01:36)



To get a full view of this memory management in xv6, we had seen this particular diagram. And in particular, we had seen about the segmentation unit which is this part over here followed by the paging unit. We had seen especially in the segmentation unit that there is a segment selector, which is a register such as the stack segment, code segment, the ds, es, fs segment and this particular segment would index into a table known as the GDT that is the global descriptor table.

And in this particular GDT is what is present the segment descriptor. The segment descriptor when combined with the offset will give you what is known as the linear address. Let us look more about how segmentation is handled in the xv6 operating system. So, first of all we will look more into the segment descriptor.

(Refer Slide Time: 02:38)



Each segment descriptor in an x86 system has 64 bits. So, these 64 bits can be viewed as 2 words of 32 bits each. The segment descriptor contains several attributes of the segment. Some of the important attributes are shown over here. So, one important attribute is the segment base or the base address of the segment. So, this base address comprises of 3 parts, it is of 32 bits and comprises of 3 parts, 16 bits over here bits 16 to 23 over here, and bits 24 to 21 present over here.

The segment limit contains the limits of the segment. So, this is of 20 bits and is present in 2 parts. We have the 16 lowest significant bits of the present over here while the 4 most significant bits present here.

Another important attribute in the segment descriptor is the privilege level; the privilege level is of 2 bits and can have values from 0 to 3. So, user processes which have the least privilege level given a value of 3, while operating system code which has the highest privilege level has a value of 0. Another attribute in the segment descriptor is the segment type. So, these could have 3 values STA X that is executable, readable segment as well as writeable segment. In addition to this, you could actually combine these attributes for a particular segment. For instance, you could have a segment, which is executable as well as readable. So, you could specify that the segment has a type X as well as R.

(Refer Slide Time: 04:38)

Segment Descriptor in xv6

```
// Normal segment
#define SEG(type, base, lim, dpl) (struct segdesc)    \
{ ((lim) >> 12) & 0xffff, (uint)(base) & 0xffff,      \
  ((uint)(base) >> 16) & 0xff, type, 1, dpl, 1,       \
  (uint)(lim) >> 28, 0, 0, 1, 1, (uint)(base) >> 24 }
```

SEG(STA_W, 0, 0xFFFFFFFF, DPL_USER)

```
// Segment Descriptor
struct segdesc {
  uint lim_15_0 : 16;  // Low bits of segment limit
  uint base_15_0 : 16; // Low bits of segment base address
  uint base_23_16 : 8; // Middle bits of segment base address
  uint type : 4;       // Segment type (see STS_ constants)
  uint s : 1;          // 0 = system, 1 = application
  uint dpl : 2;        // Descriptor Privilege Level
  uint p : 1;          // Present
  uint lim_19_16 : 4;  // High bits of segment limit
  uint avl : 1;        // Unused (available for software use)
  uint rsv1 : 1;       // Reserved
  uint db : 1;         // 0 = 16-bit segment, 1 = 32-bit segment
  uint g : 1;          // Granularity: limit scaled by 4K when set
  uint base_31_24 : 8; // High bits of segment base address
};
```

L — 64-bit code segment (IA-32e mode only)
AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level
G — Granularity
LIMIT — Segment Limit

ref : mmu.h ([7], 0752, 0769)

58

In xv6, this segment descriptor is represented by a structure in the operating system code. So, if you look at mmu dot h, you actually see this particular structure. So, as you can see all the attributes in the segment descriptor are represented in this structure. For instance, the base address which is of 32 bits and split into 3 parts is also represented by 3 parts over here, you have 16 bits over here followed by 8 bits and the most significant 8 bits

present over here. In addition to this, there is a macro in xv6 known as SEG which takes 4 parameters type base that is a base address or segment limit and the DPL. So, this macro is used to create segment descriptor, it is a helper macro which is use to create a segment descriptor.

Typical usage of this SEG macro is as follows. So, this particular usage of SEG creates segment which has a base address of 0, it has a limit of 2 power 32 minus 1 that is 0 x followed by 8 Fs, it is having a privilege level of user specified by DPL underscore user and it is of type w - that is the segment id writeable.

(Refer Slide Time: 06:00)

Segments in xv6

Segment	Base	Limit	Type	DPL
Kernel Code	0	4 GB	X, R	0
Kernel Data	0	4 GB	W	0
User Code	0	4 GB	X, R	3
User Data	0	4 GB	W	3

59

Xv6 does not make use of segmentation much it only creates 4 segments. These are the kernel code kernel data user code and user data. All these segments have a base address of 0, and have a limit of 4 GB. All the code segments that is use kernel code and the user code are of type executable and readable that is X, R over here as well as here the data that is the kernel data and the user data are of type writeable that is W, W. Now the kernel code and data have a DPL value of 0, which indicates the highest privilege level while the user code and data have a DPL value of 3 indicating the lowest privilege level. Next we will see how these 4 segments are created in xv6.

(Refer Slide Time: 06:54)

Loading the GDT

2308

```
struct segdesc gdt[NSEGS];
```

1724

```
c = &cpus[cpunum()];
c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);

lgdt(c->gdt, sizeof(c->gdt));
```

0512

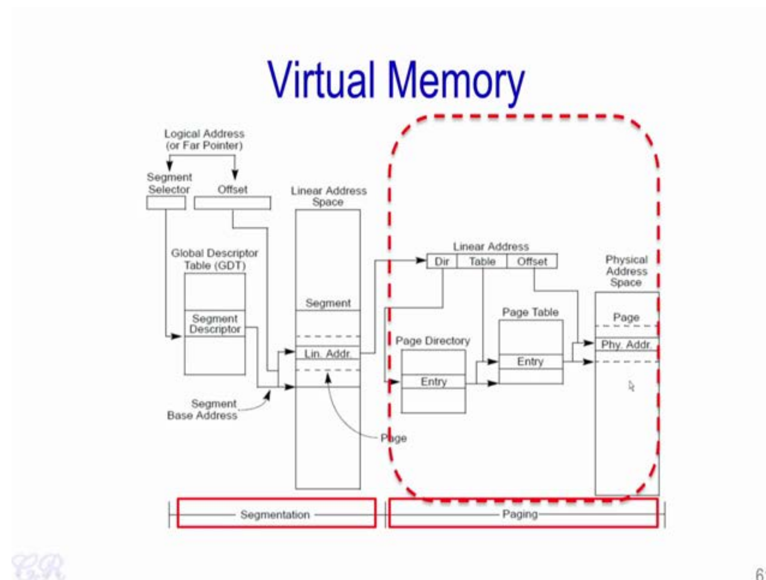
```
static inline void
lgdt(struct segdesc *p, int size)
{
  volatile ushort pd[3];

  pd[0] = size-1;
  pd[1] = (uint)p;
  pd[2] = (uint)p >> 16;

  asm volatile("lgdt (%0)" : : "r" (pd));
}
```

60

So, when it comes to segmentation it is required to have a GDT, which is present in memory. This particular declaration which is struct, segdesc gdt, which takes the number of segments, is an array comprising the GDT. If you look up line number 2 3 0 8 of the kernel code, you will see this declaration. So, this particular memory array or this particular array is used to store the GDT table. This particular array is filled by this these codes. So, we create the 4 segments as follows. So, we create the 4 segments has the kernel code segment, the kernel data segment, the user code segment and the user data segment. So, these segments are created according to the rules we seen over here

Finally, we need to have the GDT register pointing to this particular GDT table, so that is d1 by this invocation of this particular functions known as lgdt or load gdt. So, load gdt is the functions present in line about 5 and 2; and essentially it takes two parameters it takes pointer to the segment descriptor which essentially is the pointer to this GDT table and the size of the GDT table which we have passed over here size of c gdt that is the size of this GDT table, and essentially it just going to invoke this particular instruction call lgdt which fills the GDT pointer in the mmu with the address of the gdt.

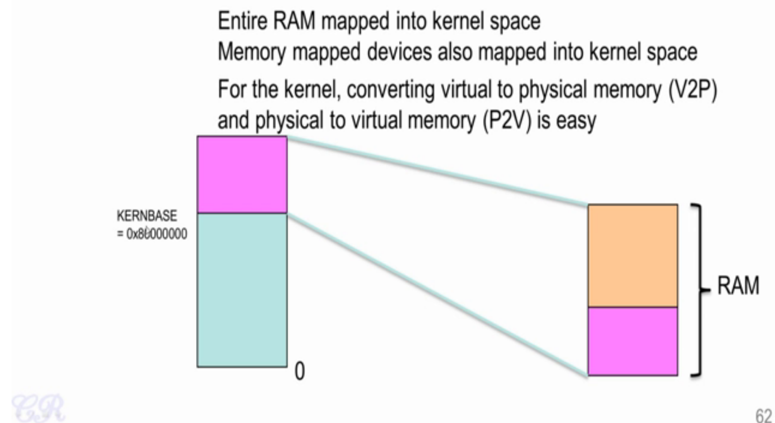
(Refer Slide Time: 08:38)



Next to the move on from segmentation to the paging unit that is how the virtual memory is managed in the xv6 operating systems. As we seen before in x 86 systems, virtual address sink is actually managed by two levels of page directories. As a result, the linear address or the virtual address is split into 3 parts; you have a directory entry which comprises of 10 bits, table entry comprising of 10 bits, and an offset comprising of 12 bits. The directory entry is use to index in a page directory table which is present in memory.

In x 86 systems, this page directory is pointed to by a register known as the CR 3 register. Now, the content of the page directory entry is then used to point to a page table. The second part of the linear address that is a table index is then used to offset into the page table to get the page corresponding page table entry then the contents of the page table is then added to the offset to get the corresponding physical address.

(Refer Slide Time: 09:58)



Now, we will see how this virtual addressing scheme is managed in xv6. As the x v 6 begins to boot the operating system kernel code and data gets copied into the lower regions of RAM as shown in this pink shaded region. As the OS continues to boot page directories and page tables are created such that the entire RAM gets mapped into the higher region of the logical address base that is the 0th location of the RAM gets mapped into what is defined as the kern base that is 0x80000000. Similarly, every address in this ram would have an identically the mapped address in the logical space.

Why such one to one mapping created between the logical address base for the kernel and the physical RAM. The reason is that, which such a map the kernel could easily convert from virtual from a virtual address to the corresponding physical address that is given any virtual address in the kernel space that is some over here, the operating system or the kernel could easily obtain the corresponding physical address using a macro known as the V2P or the virtual to physical memory conversions macro.

Similarly, in order to convert from a physical address to the corresponding virtual address would be very simple. So, any read any physical address in the RAM could be converted to the corresponding logical address or the corresponding virtual address by macro known as the physical to virtual memory macro is shorted as P2V. The important thing during in these two macros is this kern base.

(Refer Slide Time: 12:01)

V2P and P2V

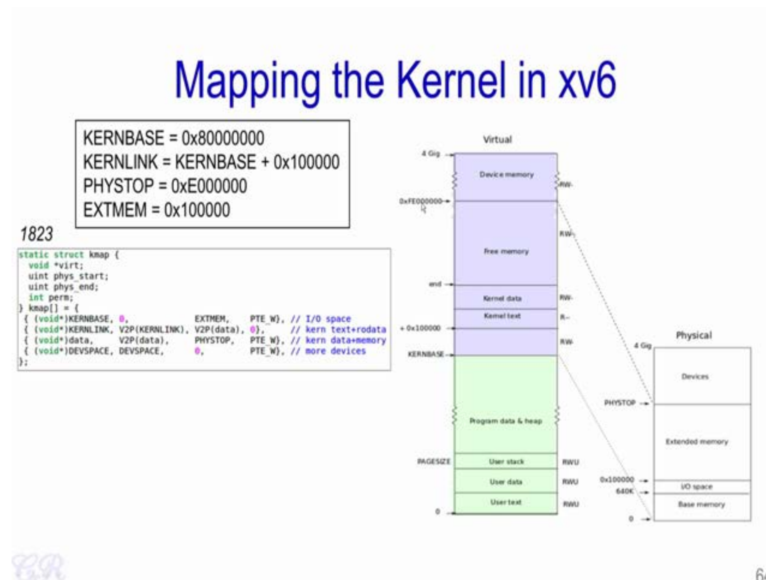
(0212)

```
#define V2P(a) (((uint) (a)) - KERNBASE)
#define P2V(a) (((void *) (a)) + KERNBASE)

#define V2P_WO(x) ((x) - KERNBASE)    // same as V2P, but without casts
#define P2V_WO(x) ((x) + KERNBASE)    // same as V2P, but without casts
```

63

Let us see these two macros in more detail. So, if you look at these two macros, this is present in the line number 212 in the kernel source code that the macro P2V simply takes and address a - which is a physical address, and converts it to the virtual address by adding a kern base. Similarly, the macro V2P takes a virtual address ‘a’, and converts it to physical address by subtracting the kern base. So, as you can see converting from virtual to physical and vice versa that is from physical to virtual becomes very simple from the operating system perspective.

(Refer Slide Time: 12:47)



Now let us see in more detail how the xv6 operating system creates the mapping between physical address and the logical address space. So, this particular figure here shows the physical RAM. So, as you can see the physical RAM is divided into several regions. The region from 0 to 640 k is known as the base memory; the region from 640 k to 1 MB that is 1 followed by five 0s is known as the IO space. While the region from 1 MB to phystop - p h y s stop is the extended memory. Now this phystop is defined as a macro, which signifies the maximum amount of RAM present in the system. So, it could be something like 2 GB, 4 GB or 16 GB and could vary from system to system. The region above this, which extends up to 4 G is used by memory mapped devices.

Now, in order to create mappings for this particular physical RAM, the operating system would create page directories and page tables. So, in order to create this mapping, the operating system code defines several macros. So, some of the important ones are shown over here. So, we have already seen about kern base which is defined to be equal to 0x8 followed by seven 0s. And there are other macros such as the kern link, phystop and EXTMEM.

Now the kern base determines the virtual address at which the kernels base starts. So, if you actually look at this logical address base, you will see that the kern base is defined at

this particular location. The region of memory below this location is user space and this is the region where user processes execute; while the region above this kern base that is from here up to the maximum of 4 gigs is where the operating system is present as well as where it manages memory and interacts with devices.

Now, another important macro is the kern link, which is the kern base plus 1 MB. So, as you see over here this particular point, which is kern base plus 1 MB denotes the kern link. So, this location is important because it is the location where the kernel code and data are present. So, starting at the kern link, there is the kernel text that is a kernel code; and after the kernel code is done, the locations comprise of the kernel that is the global data which is present in the xv6 operating system. The end of the kernel code and data is specified by this symbol end. The region from end to this particular location 0xfe followed by six 0s is the free memory, this particular free memory is use for various things in the operating system such as for the OS heap as well as for allocating pages for user processes.

Now in order to create a map of this kernel space into the physical RAM structure known as kmap is defined. The kmap structure which is present in the line number 1823 contains four elements. So, it has the virt - v i r t, which is pointer to the virtual address then it has the physical start address which indicates the physical address which gets mapped to the virtual address, and the physical end that is the end of that region, and of course, there is the permissions. So, xv6 defines four such regions. So, we have the IO space, the kernel text and read only data the kernel data and memory regions and the device space. So, these four regions can be actually mapped into this particular kernel space.

For example, the IO space which starts from the virtual address kern base and gets mapped to the physical address 0 up to text EXTMEM. So, this is the 1 MB regions starting from 0 to the end of the IO space that is the 1 MB muck. So, this is known as the IO space and is typically not used by the xv6 operating system.

The region kern link onwards is used to store the kern text plus rodata that is kernel code and read only data. So, this region gets mapped from the start of the extended memory in

the sense that kern link which is this particular location over here present by this line gets mapped into the extended memory. So, it is at the start of the extended memory that contains the kernel code and read only data in physical RAM. Then we have region known as the data region which gets mapped into the free memory so over here, and finally, you have the device bases which is above this particular location 0xFE000000.

(Refer Slide Time: 18:38)

Creating the Page Table Mapping for the kernel

- Enable paging
- Create/Fill page directory
- Create/Fill page tables
- Load CR3 register

Setting paging enable bit in CR0 register (1049)

66

Now let us see the sequence of steps that is involved in creating the mapping between the xv6 kernel spaces to the RAM. So, these are the four steps involved. First, there is enable paging. So, by default when the system is turned on paging is disabled. So, in order to turn on paging, particular bit known as the paging enable bit which is present in the CR0 register has to be set to 1. The CR0 register is a specific register in the x86 processor and in that register there is a bit call the paging enable bit which needs to be set to 1 to enable paging.

(Refer Slide Time: 19:25)

Creating the Page Table Mapping for the kernel

- Enable paging
- Create/Fill page directory — done in function walkpgdir (1754)
- Create/Fill page tables
- Load CR3 register

67

Another step is to create the page directories and the page tables. In order to create and fill the page directories the function walk page directory or walkpgdir is invoked by the operating system. So, if you look up the source code, you will see this walkpgdir function present in line number 1754.

(Refer Slide Time: 19:45)

walkpgdir (1754)

- Create a page table entry corresponding to a virtual address.
- If page table is not present, then allocate it.
- PDX(va) : page directory index
- PTE_ADDR(*pde) : page directory entry
- PTX(va) : page table entry

Linear Address
Dir | Table | Offset
Physical Address
Offset
Page Table
Flags
Page Directory
CR3

68

The walk page directory function creates the page table entry corresponding to a virtual address. So, essentially it is going to create an entry in this particular page directory. Secondly, if it finds that the corresponding page table entry in the page directory is not present then it creates in RAM page table for it. So, this page tab table as we seen will be of 1 page that is of 4 kilo bytes. So, we have seen that there are 1024 entries and each entry is of 32 bits. So, in all this page table will be of 4 KB that that is it will it will hold one page.

In a similar way, other page tables are created whenever required. So, in this particular function use a several macros such as the PDX macro which given the virtual address will extract the page directory in index that it is going to just give you this upper 10 bits of the linear address. Then we have the page table entry address which will give the page directory entry and the PTX macro which takes the virtual address and gives you the page table entry.

(Refer Slide Time: 21:06)

Creating the Page Table Mapping for the kernel

- Enable paging
- Create/Fill page directory
- Create/Fill page tables
- Load CR3 register

done in function mappages (1779)

69

After creating the page directory, the next step is to fill in the page tables. So, this is done by the map page which is present in line number 1779 of the xv6 source code.

(Refer Slide Time: 21:17)

mappages (1779)

- Fill page table entries mapping virtual addresses to physical addresses
- What are the contents?
  - Physical address
  - Permissions
  - Present bit

70

So, what the map pages does is that it is going to fill this particular page table with the mapping from the virtual address to physical address. So, these particular table entries contain as we have seen the physical address mapping permissions and also present bit. So, as we have seen that an entry in the page table is then used along with the offset to create the corresponding physical address.

(Refer Slide Time: 21:47)

Creating the Page Table Mapping for the kernel

- Enable paging
- Create/Fill page directory
- Create/Fill page tables
- Load CR3 register

Load the CR3 register to point to the page directory.

71

So, once we have created the page directories and the page tables, the final step is to load the CR3 register. The CR3 register is another register in x86 processor which contains the pointer to the page directory.

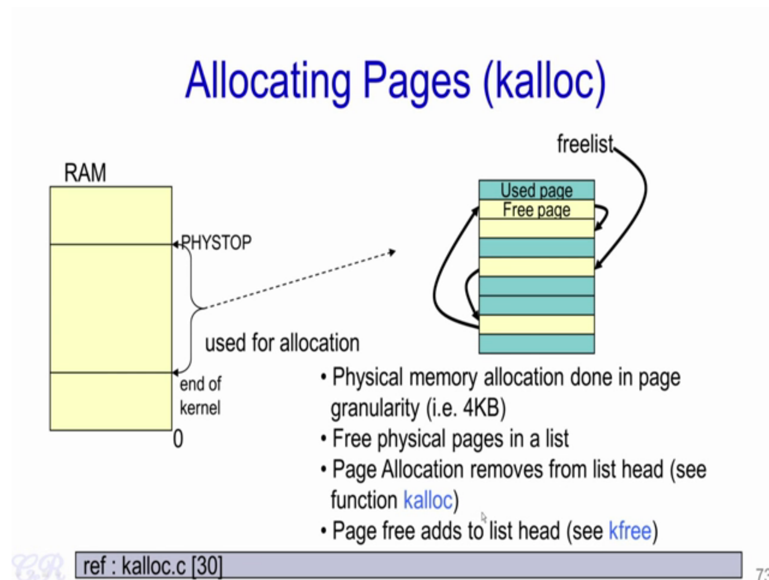
(Refer Slide Time: 22:05)

mappages (1779)

- Fill page table entries mapping virtual addresses to physical addresses
- What are the contents?
  - Physical address
  - Permissions
  - Present bit

In other sense we have this CR 3 register over here, which is present in the processor of the MMU and it points to the memory location of the page directory. Now that we have seen how the xv6 code enables paging creates page directories as well as page tables. Let us see how the xv6 operating system allocates memory for various purposes. So, these purposes could be from allocating pages for user processes or for the operating system its use itself.

(Refer Slide Time: 22:45)



We had seen that the xv6 code and read only data gets loaded in the 0th location of RAM, and extends upwards. At the end of the code and read only data up to phystop that is the physical end of the RAM is the free memory; this free memory is utilize by the OS for several purposes such as for allocating pages to user processes or for internal operating system book keeping requirements. So, what we will see now is how this free memory is managed by the operating system.

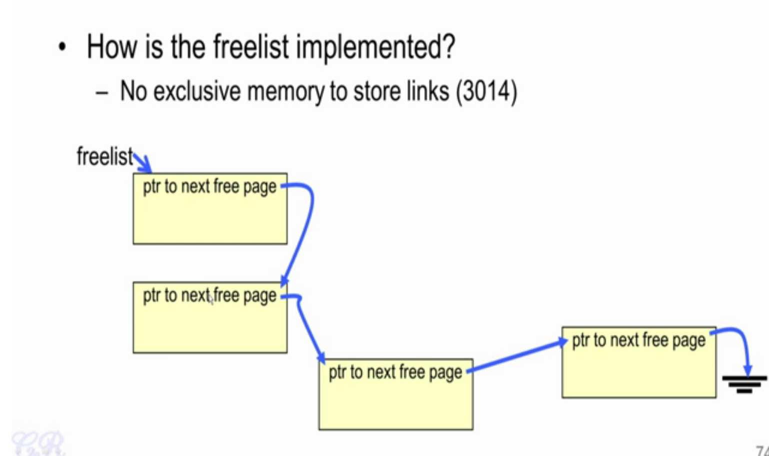
Essentially this free memory could be a large junk and it is split into pages of 4 kilo byte granularity thus we would have several pages present in this particular free memory region. Now, whenever required that is on demand page would be allocated to a particular calling function and used for it is requirements. So, after the usage the particular page is freed.

Now, the next thing that we need to think about is how the operating system or that is how the xv6 OS determines which page to be allocated and how the pages should be freed. In order to do this the xv6 code maintains link list of free pages. So, as you can see here, this particular figure has pages which are either blue or yellow. So, this entire region corresponds to the free memory region in the RAM, the blue region are the pages

which is utilized by either user process or by the operating system itself while the yellow pages are the once which are free.

Now, all the free pages are linked together using link list which is pointed to by pointer known as the free list. The free list points to the head of the link list and all the free pages are linked together with this list. Now in order to allocate page with kalloc function is utilized. The kalloc function would essentially remove a free page from the list and assign that page to be used. In order to free a page the kfree function is used; essentially the k free function would add the free page back into the list.

(Refer Slide Time: 25:15)



The link list is as shown over here where you have the free list pointer which points to the head of the list and then there are pointers to the consecutive free pages. Now, one thing which is different from the standard link list is that there is no exclusive memory to store the pointer to the next page. You note that each of these pages is of 4 kilo bytes and this 4 kilo byte free pages is not used for any other data thus the 0th location in this page contains the pointer to the next page.

In a similar way, the 0th location in this page would contain the pointer to the next page and so on. Thus we are creating this list. The free list points to the first or the head node

of the list and then the 0th location of that node points to the consecutive page and so on until we reach the end of the list. So, with that we come to the end of how xv6 manages memory. So, it is not the best management scheme that is possible, but it is a representative of what several operating systems actually do to manage memory.

Thank you.