**Computer Organization and Architecture**
**Prof. V. Kamakoti**
**Department of Computer Science and Engineering**
**Indian Institute of Technology, Madras**

**Lecture - 31**
**Paging, Cache**

So, just very quickly; so all of you know how paging is enabled.

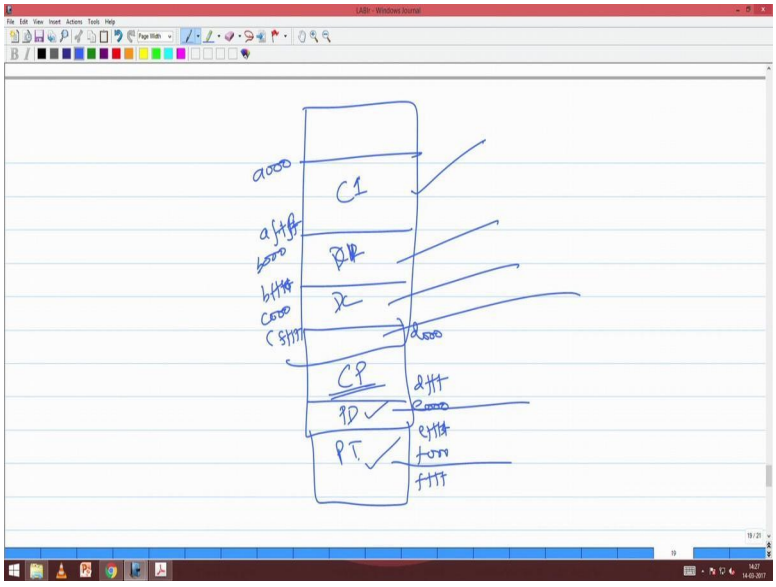
Student: (Refer Time: 00:24).

Right, and then what are the precautions to be taken before enabling paging, I thought in the class.

Student: (Refer Time: 00:38).

The page that enables paging should be identically mapped, then only the next instruction to be executed.

(Refer Slide Time: 00:46)



So, there is a page a000 to afff: three f's. And then starting from b000 there are this bfff, c000 to cfff. So, these three are pages in the. So, let me call it as data page 1, data page 2. Then there is something from d000 to dfff, then e000 to some efff, then f000 to ffff. So, where are the page tables you have stated? Page table is from f000 to ffff and e000 to efff

is the page directive. And this is what you call as something like a control page which has earlier segmentation and other things.

So, this page should be identically mapped, this page also, this page also, this page also, this one, and this one, all are identically mapped. Then every other page the odd page should be mapped onto this one b000 to c000 and the bfff and even page should be mapped on to cfff. That is how you set up your page directory and one page. So, let us say this is one general thing that you can, then things like you can access a very big address but see that it is going to access only a somewhere in b000 or c000. Then jump to a very large off set, but it will jump to the next instruction. Things like that you can tr. And once you do this, you will also see some changes in the page table. That page is accessed, that page became dirty, all these things you can do.

The next interesting thing is you actually make one page not available and try to access that address, see that it goes to a page fault handler. And what will the page fault handler do? It will go and fetch the page do lot of things that we will not do that operating system will teach you. The page fault handler can just go and make that bit 1 and restart that instruction so that again you can restart and start executing that again.

So, you can create a page fault and make that page 1 and then come back. So, these are all simple exercise that you can basically carry over and get an idea of how this paging works very very simple exercises. You apply your mind half an hour you can finish this not a great deal at. And the interesting thing after that there is set of problems that we have been cooking right, so every semester we are getting some new set of problems. So, those problems what we says you can once you have this infrastructure solving these problems are going to be very straight forward, correct.

So, this is what is needed out of you, very very simple. You get these basic building blocks get an understanding of a then whatever problem is uploaded in the model just go and do that. And that is going to very straight forward. See I am telling you while doing this you will understand entire thing about paging; tomorrow the operating system goes they talk about anything about paging you will understand what it is. How will the page fault handler work, how will this exception will generated, all these things will be known.

So, you have to write a page fault handler. What will the page fault handler do? It should find out where which page there is a fault and then it has to go and make that one. So, when a page fault happens there are some control register actually updated saying where that page fault which page, you can get some ideas.

Just go and look at this Intel manual 3.

(Refer Slide Time: 04:36)

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Address of page directory[1] | | | | | Ignored | | | | PCD | PWT | Ignored | | | | CR3 |
| Bits 31:22 of address of 4MB page frame | | Reserved (must be 0) | Bits 39:32 of address[2] | PAT | Ignored | G | 1 | D | A | PCD | PWT | U/S | R/W | 1 | PDE: 4MB page |
| Address of page table | | | | | Ignored | | 0 | Ign | A | PCD | PWT | U/S | R/W | 1 | PDE: page table |
| Ignored | | | | | | | | | | | | | | 0 | PDE: not present |
| Address of 4KB page frame | | | | | Ignored | G | PAT | D | A | PCD | PWT | U/S | R/W | 1 | PTE: 4KB page |
| Ignored | | | | | | | | | | | | | | 0 | PTE: not present |

Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

NOTES:

1. CR3 has 64 bits on processors supporting the Intel-64 architecture. These bits are ignored with 32-bit paging.
2. This example illustrates a processor in which MAXPHYADDR is 36. If this value is larger or smaller, the number of bits reserved in positions 20:13 of a PDE mapping a 4-MByte page will change.

You will get all this ideas. So, you have to find out where went wrong and go and make that bit one and then restart it will work. So, in terms of this please look at; see in a page directory entry right the first bit is a valid bit. If we have 1 that means, all these things make sense, if you have 0 nothing means sense. So, this is a page directory entry and this is a page table entry. So, you could also have 4 MB pages. In large data bases they will, now we are starting of 4 KB. We can have the page size to be 4 MB also. So, we will not bother about 4 MB here, we will just do 4 KB because that is trying to understand.

So, these two whatever page this is 114 page number of this volume 3 essentially talks about how the page directory entry looks like, and this also tells you how the page table entry will. So, this is what is pointed to by sorry; CR 3.

(Refer Slide Time: 05:37)

Table 4-3. Use of CR3 with 32-Bit Paging

| Bit Position(s) | Contents |
|---|---|
| 2:0 | Ignored |
| 3 (PWT) | Page-level write-through; indirectly determines the memory type used to access the page directory during linear-address translation (see Section 4.9) |
| 4 (PCD) | Page-level cache disable; indirectly determines the memory type used to access the page directory during linear-address translation (see Section 4.9) |
| 11:5 | Ignored |
| 31:12 | Physical address of the 4-KByte aligned page directory used for linear-address translation |
| 63:32 | Ignored (these bits exist only on processors supporting the Intel-64 architecture) |

This is what the pointed to by CR 3: the start of this page directory and this will point though this. So, every page table its address will be there in the PDA entry. When the first bit is 1 as you see here I am just rotating here, when the first when the first bit is 1 then it essentially means that this entry is valid otherwise it is invalid. And what are these things let us go one by one, very quickly read slash write what is R slash W.

(Refer Slide Time: 06:19)

PAGING

Table 4-4. Format of a 32-Bit Page-Directory Entry that Maps a 4-MByte Page

| Bit Position(s) | Contents |
|---|---|
| 0 (P) | Present; must be 1 to map a 4-MByte page |
| 1 (R/W) | Read/write; if 0, writes may not be allowed to the 4-MByte page referenced by this entry (see Section 4.6) |
| 2 (U/S) | User/supervisor; if 0, user-mode accesses are not allowed to the 4-MByte page referenced by this entry (see Section 4.6) |
| 3 (PWT) | Page-level write-through; indirectly determines the memory type used to access the 4-MByte page referenced by this entry (see Section 4.9) |
| 4 (PCD) | Page-level cache disable; indirectly determines the memory type used to access the 4-MByte page referenced by this entry (see Section 4.9) |
| 5 (A) | Accessed; indicates whether software has accessed the 4-MByte page referenced by this entry (see Section 4.8) |
| 6 (D) | Dirty; indicates whether software has written to the 4-MByte page referenced by this entry (see Section 4.8) |
| 7 (PS) | Page size; must be 1 (otherwise, this entry references a page table; see Table 4-5) |
| 8 (G) | Global; if CR4.PGE = 1, determines whether the translation is global (see Section 4.10); ignored otherwise |

Read write means, the first bit is 0th bit which is called bit 0 is the present bit. So, first bit is read write, if 0 means writes are not allowed, 1 means write will be. Anyway a page

can be read for sure, but I may not allow you to write. So, for example, where do you want in the absence of segmentation, suppose I have an architecture where there is no segmentation; where do you want no writes.

Student: Code Segment.

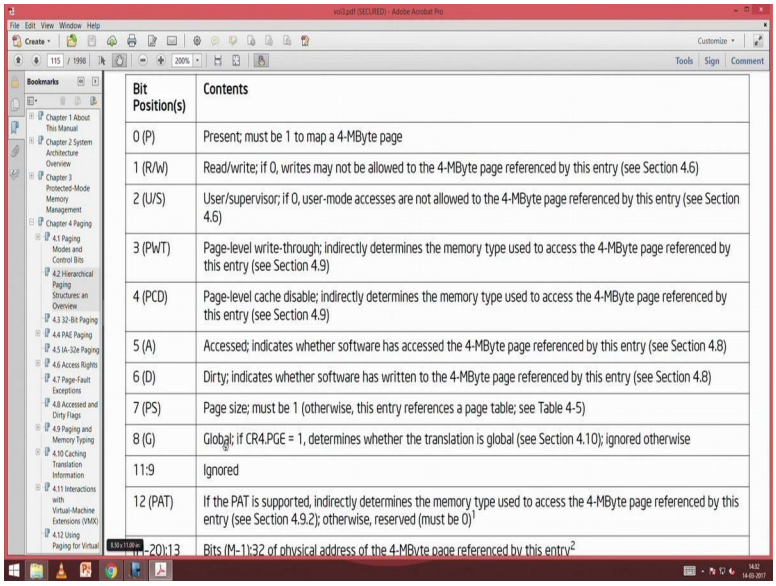
Code segment, correct; code segment I do not want to touch or I could have some parameters segments which I want to preserve, I do not want to program to come in corrupt it. So, if I make certain segments as read only then lot of security issue can be handled.

The second bit is user slash; if 0 means user mode access or not allowed otherwise it will be allowed. User mode means three supervisor means 0 1 and 2 right, or 0 is supervisor 1 2 and 3 are user I do not know, you just check Section 4.6, we will go there but no, there is little it will takes time to back. And Section 4.6 I will tell you: I think user is 1 2 3 sorry; user is 3 and supervisor is 0 1 2 ok.

This page level cache, level write I will just check later. Just forget about 3 and 4 we will deal it when we do the cache. Fifth is access: after a loaded did I go and read or write into it that is this. Sixth is the our famous dirty bit. Now then there is something called 7; 7 is page size page size must be 1 otherwise this entry references a page table; forget this also you just make it 1.

I will we will discuss about that later.

(Refer Slide Time: 08:20)



| Bit Position(s) | Contents |
|---|---|
| 0 (P) | Present; must be 1 to map a 4-MByte page |
| 1 (R/W) | Read/write; if 0, writes may not be allowed to the 4-MByte page referenced by this entry (see Section 4.6) |
| 2 (U/S) | User/supervisor; if 0, user-mode accesses are not allowed to the 4-MByte page referenced by this entry (see Section 4.6) |
| 3 (PWT) | Page-level write-through; indirectly determines the memory type used to access the 4-MByte page referenced by this entry (see Section 4.9) |
| 4 (PCD) | Page-level cache disable; indirectly determines the memory type used to access the 4-MByte page referenced by this entry (see Section 4.9) |
| 5 (A) | Accessed; indicates whether software has accessed the 4-MByte page referenced by this entry (see Section 4.8) |
| 6 (D) | Dirty; indicates whether software has written to the 4-MByte page referenced by this entry (see Section 4.8) |
| 7 (PS) | Page size; must be 1 (otherwise, this entry references a page table; see Table 4-5) |
| 8 (G) | Global; if CR4.PGE = 1, determines whether the translation is global (see Section 4.10); ignored otherwise |
| 11:9 | Ignored |
| 12 (PAT) | If the PAT is supported, indirectly determines the memory type used to access the 4-MByte page referenced by this entry (see Section 4.9.2); otherwise, reserved (must be 0)[1] |
| (M–1):13 | Bits (M–1):32 of physical address of the 4-MByte page referenced by this entry[2] |

Eight is global: if CR4 dot PG is 1 determines whether the translation is global otherwise ignore, this is also you ignore as because we are not going for 4 MB pages. And there are some 9 10 11 3 bits which is ignored, which you can use. Then there is page address translation again you can forget this bit m minus 20 30 32 of physical address of 4 megabit, so I forget that. So, these are the bits you should know for sure; in this page directory entry and page table entry fine.

So, you set up this page table and start working on it, this is you see this.

(Refer Slide Time: 09:21)

### Table 4-5. Format of a 32-Bit Page-Directory Entry that References a Page Table

| Bit Position(s) | Contents |
|---|---|
| 0 (P) | Present; must be 1 to reference a page table |
| 1 (R/W) | Read/write; if 0, writes may not be allowed to the 4-MByte region controlled by this entry (see Section 4.6) |
| 2 (U/S) | User/supervisor; if 0, user-mode accesses are not allowed to the 4-MByte region controlled by this entry (see Section 4.6) |
| 3 (PWT) | Page-level write-through; indirectly determines the memory type used to access the page table referenced by this entry (see Section 4.9) |
| 4 (PCD) | Page-level cache disable; indirectly determines the memory type used to access the page table referenced by this entry (see Section 4.9) |
| 5 (A) | Accessed; indicates whether this entry has been used for linear-address translation (see Section 4.8) |
| 6 | Ignored |
| 7 (PS) | If CR4.PSE = 1, must be 0 (otherwise, this entry maps a 4-MByte page; see Table 4-4); otherwise, ignored |
| 11:8 | Ignored |
| 31:12 | Physical address of 4-KByte aligned page table referenced by this entry |

This is what you see, table 4.5 is what you should see for 32 bit what are your use, that was I think that was for a 64 bit right that is a 4 megabit. So, we are doing four kilobit page sorry this is for you should know. And this is for a 32 bit entry. So, there are lots more of these paging things that will come under advanced computer architecture, because we need to know lot more about operating system to appreciate things.

So, we will stop at this stage to give you a blueprint operating system. Once you start studying operating system there you may need so many other things, you come back to argue and learn lot of this. There is no point in learning so much of Intel also; Intel is not our cousin or something. So, we will learn something about thing to just understand what is how paging works foundation we need, we will not focus too much on Intel the other processors.

As I told you 99 percent of the computing device in the same now uses on. So, Intel is only 1 percent because all computing devices are mobile devices today. So, we will not basically break our head so much. But nevertheless we should understand how it works and this is a very nice system, but nevertheless you should really want to bring their secure system I still believe, why I am still teaching this because down the line if I really want to build up for a security then it is very very important that we; I think Intel will succeed that. This level of double security, three tire security; you have a security at segmentation, security at paging and then four levels that and then I have LDT GDT type of security, then from that then paging.

So, see dead codes: what are dead codes? Dead codes are those which you cannot actually when you execute the program right you can never get that part of the code to execute. These are all potential you know worms or whatever malicious code, because in some specific input pattern that fellow will start execute. So, we are really do not know by just looking at a program or by even executing a program we cannot do a security analysis, where there could be some even function of the code and some combinations of the hidden part of the code, some composition like function one then calls function two then function three then the behavior will be different. We call function one then function three and then function two the behavior will be different. It is like it composited in different way and you get different behaviors. So, these are some very interesting things that would happen.

So security wise, I think this level three level of protection will be estimated. And then there is I do not know what you will be learning in operating system right. One of the important concept that we need to learn is about capability based operating system. What is a capability based operating system? A capability based operating system is one where for every object in an operating systems you have certain access rights, and its relation with every other object is very well defined.

For example, processors and objects; I take one process say some demon- http demon right you know what an http demon right it, basically runs the http right. So, if I take the mail demon also http demon have some access. So, there are several access resources and I have CPU, and I have memory, I have peripherals, I have ports, I have net cards, there are several things. And then within part of your system itself there are some specific parts of a code like file system management, process management etcetera.

So, this process what is this relation with ever other process, I need to have that values it should be very well defined. If it is not well defined then that is where the laps is starts. So, this fellow should not supposed to go; the http demon has demon is has not going to support go and touch some part of your super, why should it go and access your super block of your disk; there is something called super block you learn it up in OS course. Why should I go and access, right?

Suppose, I say cannot access that rule is simply implemented then your security of your operating system can be much more effective. But that type of capability based OS we are not in the position to build. But what I believe is that if you go as regress as Intel x 8 6 plus this we can build a really capability based device. Why do you enforce the policies and trust on the hardware gone and implement it.

So, I put this polices, I will put this rules, who will implement I am very happy if the hardware does this work wherein the software, because hardware is immune. I have designed the other account I change the transistor and software is put it. So, that is one reason why I still believe that x v 6 will we sort of the hardware when you want to start building those type of fancy or those type of really secured operating system. Doubts?

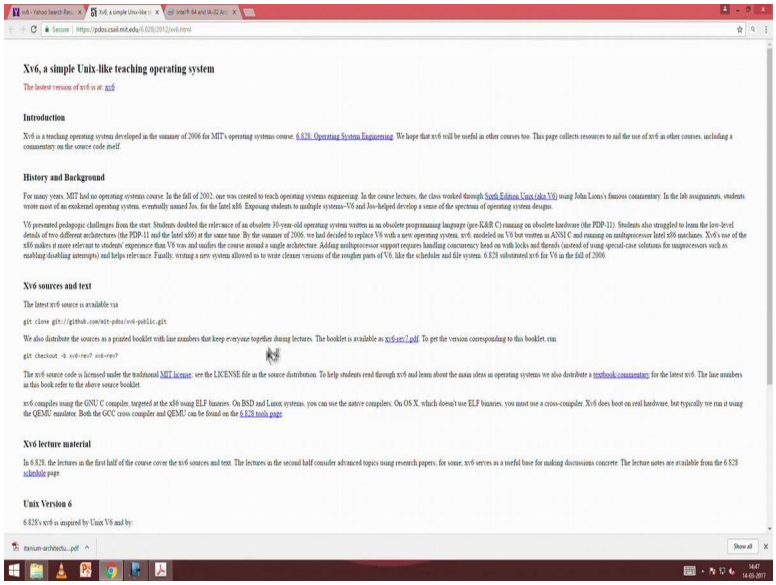
Student: Sir.

Yeah.

Student: Sir, (Refer Time: 15:20).

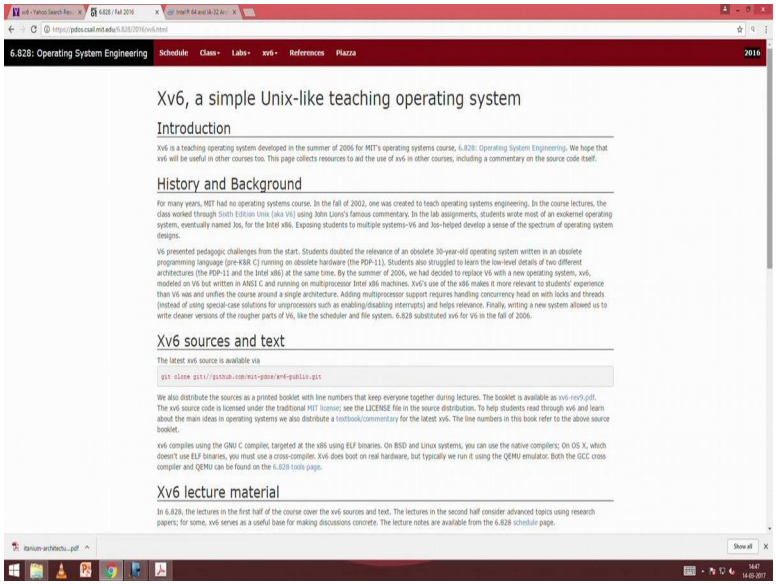
Linux uses x v 6, actually one brief question of Linux is even there 8000 of course which you can see. We ask to receive too many Linux code then you go mad x v 6 go and look at x v 6 is the whole thing.

(Refer Slide Time: 15:37)




So, there is database you can open it on see beautifully they have done.

(Refer Slide Time: 15:46)



So, this is very well written index code, you will have first 10 pages of 12 pages is. See lot of empty spaces and here it starts: that memory layout. Can you see LGDT?

(Refer Slide Time: 16:13)

```
0503   asm volatile("cld; rep stosl" :
0504               "=D" (addr), "=c" (cnt) :
0505               "0" (addr), "1" (cnt), "a" (data) :
0506               "memory", "cc");
0507 }
0508
0509 struct segdesc;
0510
0511 static inline void
0512 lgdt(struct segdesc *p, int size)
0513 {
0514   volatile ushort pd[3];
0515
0516   pd[0] = size-1;
0517   pd[1] = (uint)p;
0518   pd[2] = (uint)p >> 16;
0519
0520   asm volatile("lgdt (%0)" : : "r" (pd));
0521 }
0522
0523 struct gatedesc;
0524
0525 static inline void
0526 lidt(struct gatedesc *p, int size)
0527 {
0528   volatile ushort pd[3];
0529
0530   pd[0] = size-1;
0531   pd[1] = (uint)p;
0532   pd[2] = (uint)p >> 16;
0533
0534   asm volatile("lidt (%0)" : : "r" (pd));
0535 }
0536
0537 static inline void
0538 ltr(ushort sel)
0539 {
0540   asm volatile("ltr %0" : : "r" (sel));
0541 }
```

```
0553   asm volatile("movw %0, %%gs" : : "r" (v));
0554 }
0555
0556 static inline void
0557 cli(void)
0558 {
0559   asm volatile("cli");
0560 }
0561
0562 static inline void
0563 sti(void)
0564 {
0565   asm volatile("sti");
0566 }
0567
0568 static inline uint
0569 xchg(volatile uint *addr, uint newval)
0570 {
0571   uint result;
0572
0573   // The + in "+m" denotes a read-modify-write operand.
0574   asm volatile("lock; xchgl %0, %1" :
0575               "+m" (*addr), "=a" (result) :
0576               "1" (newval) :
0577               "cc");
0578   return result;
0579 }
0580
0581 static inline uint
0582 rcr2(void)
0583 {
0584   uint val;
0585   asm volatile("movl %%cr2,%0" : "=r" (val));
0586   return val;
0587 }
0588
0589 static inline void
0590 lcr3(uint val)
0591 {
```

So, LGDT, so a s system is assembling so they are using LGDT. LIDT, LTR road task resistor, clear interrupt, STI move instructions, GS register all these things. That means, what Linux uses lot of your assembly instruction, I am just proving. Just do not think that just delete see CR 3 everything is.

(Refer Slide Time: 16:53)

```
#ifndef __ASSEMBLER__
// 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
};

// 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 }
#define SEG16(type, base, lim, dpl) (struct segdesc)  \
{ (lim) & 0xffff, (uint)(base) & 0xffff,              \
  ((uint)(base) >> 16) & 0xff, type, 1, dpl, 1,       \
  (uint)(lim) >> 16, 0, 0, 1, 0, (uint)(base) >> 24 }
#endif

#define DPL_USER    0x3     // User DPL

// Application segment type bits
#define STA_X       0x8     // Executable segment
#define STA_E       0x4     // Expand down (non-executable segments)
#define STA_C       0x4     // Conforming code segment (executable only)
```

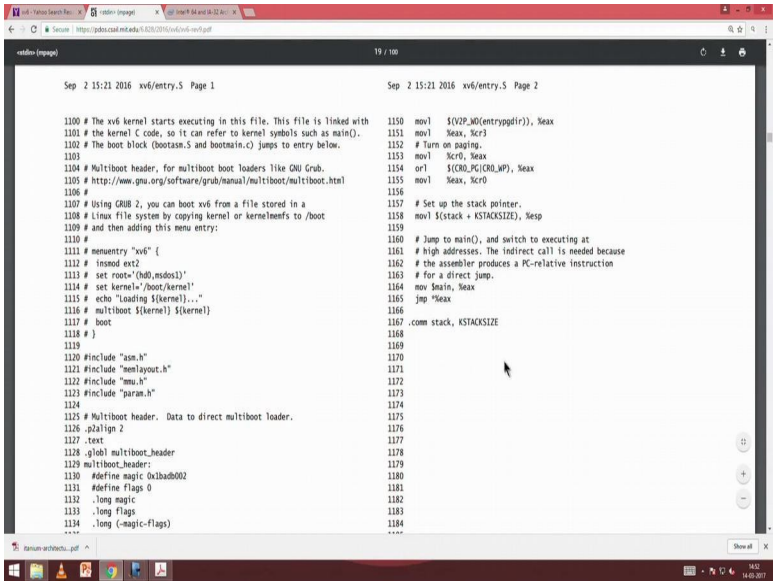


```
#define STS_IG32    0xE     // 32-bit Interrupt Gate
#define STS_TG32    0xF     // 32-bit Trap Gate

// A virtual address 'la' has a three-part structure as follows:
//
// +--------10------+-------10-------+---------12----------+
// | Page Directory |   Page Table   | Offset within Page  |
// |      Index     |     Index      |                     |
// +----------------+----------------+---------------------+
//  \--- PDX(va) --/ \--- PTX(va) --/

// page directory index
#define PDX(va)         (((uint)(va) >> PDXSHIFT) & 0x3FF)

// page table index
#define PTX(va)         (((uint)(va) >> PTXSHIFT) & 0x3FF)

// construct virtual address from indexes and offset
#define PGADDR(d, t, o) ((uint)((d) << PDXSHIFT | (t) << PTXSHIFT | (o)))

// Page directory and page table constants.
#define NPDENTRIES      1024    // # directory entries per page directory
#define NPTENTRIES      1024    // # PTEs per page table
#define PGSIZE          4096    // bytes mapped by a page

#define PGSHIFT         12      // log2(PGSIZE)
#define PTXSHIFT        12      // offset of PTX in a linear address
#define PDXSHIFT        22      // offset of PDX in a linear address

#define PGROUNDUP(sz)  (((sz)+PGSIZE-1) & ~(PGSIZE-1))
#define PGROUNDDOWN(a) (((a)) & ~(PGSIZE-1))

// Page table/directory entry flags.
#define PTE_P           0x001   // Present
#define PTE_W           0x002   // Writeable
```

So, if you want actually learn see this. See this is segment descriptor limit, waves, base then type and somewhere so limit 19 to 60 this is your segmentation. These are all completely used. So, if you can actually go through this. Now it is all go to, this is your context of your process task state. So, see you have entries for all your registers right eax, ecx, ede, vx and then your csss ds; I told you right CR 3 that is the entry fist you are also see in your task state. So, for every task [FL], so for every task you can have your page directory base. I told you in the morning right. So, this is that. So, this is context of your process and it goes on.

So, the entire course, this is the first line of code boot asm dot s. So, the first two will be some entry here, so it goes.

(Refer Slide Time: 19:02)




So, this is how you enable paging here see: movl cr 0 eax or l mov as eax cr 0 right. So, this is the point where you are enabling paging, turn on paging. So, whatever you did now operating system also does. So, I am just giving you a one to one mapping of what is happening, then it to jumps to eax.

So, what do you did exactly in your lab right for enabling paging that is there in line number 1153 to 115. If you understand x v 6 then you can do and directly play with the

corn. There are some 10 exercises here you understand all the 10 exercises you can go and play directly with the corn. See clash lab: lab 1 to lab 7. After this right, after your current course on the operating system goes best thing is to take x v 6 understand it

completely and the take this compile this into assembly and boot your. Now you are booting with that USB right, boot with x v 6 and see how it is working. And then go and change them scheduling and see how the counter works. So, that will be the (Refer Time: 19:59). You know one full semester you should do fully operating system, (Refer Time:

20:03) complier networking and all these things.

But then if you do that you will become real OS data; that is something. Because you really got a machine to boot scratch and come up. So, go and read this x v 6. This is real operating system.

Student: Sir.

Yes, you are only using trap gate right so for.

Student: (Refer Time: 20:34).

(Refer Time: 20:45).

Student: (Refer Time: 20:46).

You should have used only trap gate, you have used task gate because the template.

Student: (Refer Time: 20:51).

Why, what do you mean by implement the trap gate? Just you have to go and make that IDT entry as whatever. You have to just change the bit type there that is all then it becomes a trap gate.

Student: (Refer Time: 21:08).

Hidden work on, send the code that does not work. See the task gate will be thought in assignment number 5 and I will also explained why task gate is necessary in the context of in interrupt service routine. That I will explained that is something called double fault and for that we need to have a task gate. But for all practical purpose fort your intra service routine that you have written as a part of your third assignment; task gate is trap gate is enough; trap or interrupt gates are enough. Task gate you should not use means you can use, but.

Student: (Refer Time: 21:55).

Because your interrupt gate, the privilege level of your privilege service routine.

Student: (Refer Time: 22:11).

You set it to three, and what was the code segment privilege level? So, the interrupter descriptor they will work point into you fourth segment right. What was the privilege level of the code segment?

Student: Zero.

So, how will it work?

Student: (Refer Time: 22:35).

So, it will not work, right. So, we will teach you about task gate in the fifth assignment. So, a trap gate for an interrupt gate should have worked in your third assignment, if you did not work then send us the code I will clarify it.

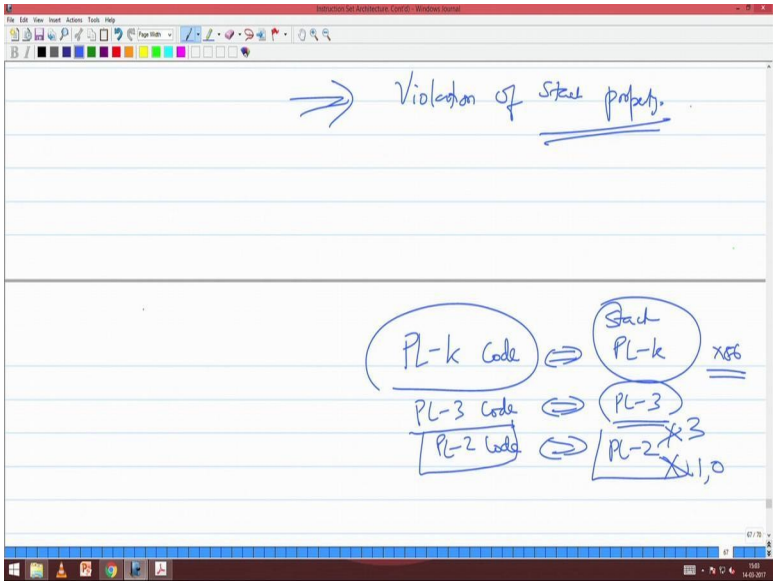
Student: Sir.

Hm.

Student: (Refer Time: 23:06) of the entry of the IDT, that is used such that the (Refer Time: 23:13) segment that is trying to access it will have the privilege (Refer Time: 23:18) than that of the entry of the IDT, right.

Right, so there are several issues here, I can only talk about; I will give you a brief of what is happening.

(Refer Slide Time: 23:30)

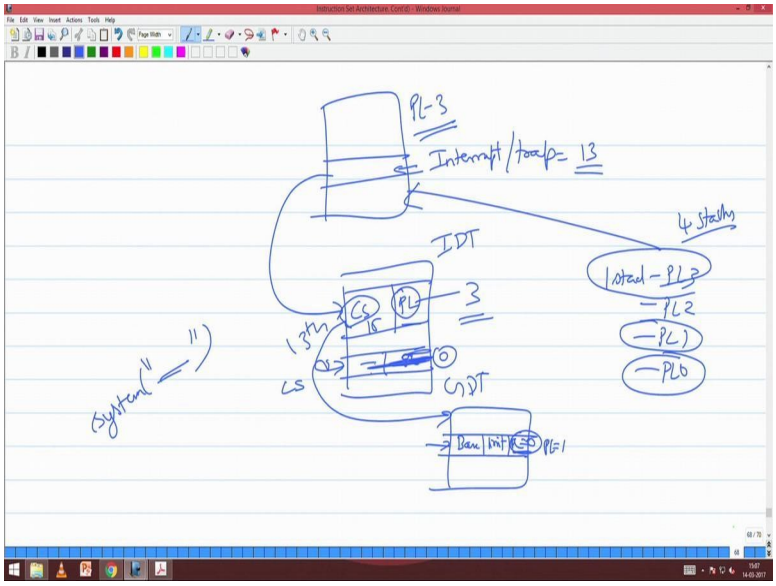



See if I am a privilege k code then my stack has to be privilege k. Stack cannot be k plus 1 or k minus 1. So, this is very important. This is what (Refer Time: 23:46) 6 will ask for. So, if I am a privilege k code my stack also will be privilege k.

So, if I am a privilege 3 code the stack I am going to use should also be privilege 3. It cannot be privilege 2 or 1, if I am a privilege 2 code the stack should also be privilege 2. It cannot be 3, it cannot be 1 or 0, please note that. It is unlike a privilege level 3 code can access a code segment of privilege level 3 and privilege level 2 can access 2 and 3, but for stack if I am executing a privilege 3 code the stack should also be privilege 3. It cannot be privilege 4 or that descript are should have privilege 3.

So, this is how x 8 6 is define the reason is as follows.

(Refer Slide Time: 24:46)




Now, let us say I am executing a privilege level 3 code; there is one instruction which generates an interrupt or trap. Now what will happen is this will go to the interrupt. So, let me state is giving you interrupt 30. So, I go to the IDT table, I go to the 13th entry and here I have a code segment selector and then there is some privilege level here. This privilege level should be at least 3 then only this fellow; if this was PL 3 this was also be 3, if it is PL 2 then this should be 2 or 3 then only this interrupts itself will be executed.

Now this will point to you a code segment, if this code segment is set privilege 0. This code segment this is a selector right this will say some 15 or something. So, 15th entry in your not in your IDT, this will be a 15 entry in your GDT or LDT depending on that. There you will go; here there will be a base limit etcetera. And then there will be a privilege level. This privilege level is 0. That means, now when it 3 code it is executing because of an interrupt I am going to execute a privilege level 0 code.

So, what you will do is you will have a stack define for every process that I am creating, I will have 3 stacks or 4 stacks. Every process I am creating I could have 4 stacks: one stack is at privilege 3, another at privilege 2, another at the privilege 1, another will be privilege 0. If this interrupt service routine in is at privilege 1 or 0 right, then this stack either the zeros stack or the ones stack will

be executed depending on what the privilege level here.

So, for all the processing of your interrupt service routine this PL 3 stack will not be used your Pl 1 or Pl 0 stack will be used; depending on what the privilege level. So, why is this done? Because, I do not want the system stack and the user stack to be the same. I am a program, I am asking for a service from the operating system that is called a system call; I say print f scan, I go and ask many many things from the operating system. In c there is a command called system right, are you aware of this system and I can put whatever I want. So, essentially I transfer control to a system: malloc for example is a system call, f open is a system call free is a system call.

So, when I call the system, the system call should not execute on the same stack as my. Then what will happen after I return? I have access to the stack; I accept user level process has access to the stack. So, I will start knowing more about what the stack contains right. So, I will start having more ideas about what the stack is trying to do. So, I do not want the privilege level 0 code to execute on the same stack, because after I come back I will have access to all that has been done. In the imagine password right this will take your password do a hash and then compute there, so all those computations, all those received you things can done.

So, that is why x 8 6 says that you are a user fellow you use stack when you want to go to a privilege level 0 when you want executive interrupt service routine, whenever when you come up right when you are actually spooned as a process when you start executing you will have your stack, then there will be three most stacks. Suppose I am a privilege 3 there will be three most stack privilege 1 2 and 0, 1 and 2; if the interrupt service routine is going to be for 0 then it will reuse your Pl 0 stack. As a process I have four stacks and my Pl 0 stack will be used by the Pl 0 code. If my interrupt service routine is Pl 1: for example, print f f print f can be PL 1 while malloc can be PL 0. Or some exit can be Pl 0.

So, I will have PL 1 PL 0. So, depending upon what my system call is which privilege level my system call is that corresponding stack can be used. So, what would have happened? When you are used and interrupt gate and your code segment was at PL 0 your 0 stack could not have been set correctly. And that is why you are interrupt gate intra slash trap gate would not have work correctly. Are you getting this?

So, when a processes is (Refer Time: 31:02), when we will doing the task switching I will I will cover it in great detail when a process is (Refer Time: 31:09) these stacks

should be set. Once the stacks are set when I do an interrupt then these stacks will be use. Now what I want you to do is, you use that you see there is an interrupt generated go and see which what is the interrupt number, go to the code segment go and find the privilege level of the code segment if it is 0 then you see whether there is a 0 stack set. Where will that 0 stack set? It will be set in a task state segment.

So, we will come to it. First find out what is the privilege level of the interrupts service and then you send me the code we will debug and send it back. Are you able to follow?