

XV6 bootloader

Abhijit A. M.
abhijit.comp@coep.ac.in

Credits:
xv6 book by Cox, Kaashoek, Morris
Notes by Prof. Sorav Bansal

A word of caution

- **We begin reading xv6 code**
- **But it's not possible to read this code in a “linear fashion”**
 - **The dependency between knowing OS concepts and reading/writing a kernel that is written using all concepts**

What we have seen

- **Compilation process, calling conventions**
- **Basics of Memory Management by OS**
- **Basics of x86 architecture**
 - Registers, segments, memory management unit, addressing, some basic machine instructions,
- **ELF files**
 - Objdump, program headers
 - Symbol tables

Boot-process

- **Bootloader itself**
 - Is loaded by the BIOS at a fixed location in memory and BIOS makes it run
 - Our job, as OS programmers, is to write the bootloader code
- **Bootloader does**
 - Pick up code of OS from a 'known' location and loads it in memory
 - Makes the OS run
- **Xv6 bootloader: bootasm.S bootmain.c (see Makefile)**

bootloader

- BIOS Runs (automatically)
- Loads boot sector into RAM at 0x7c00
- Starts executing that code
 - Make sure that your bootloader is loaded at 0x7c00
 - Makefile has

```
bootblock: bootblock.S bootmain.c
$(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c bootasm.S .....
...
$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o bootblock.o
bootasm.o bootmain.o
```

Results in:

00007c00 <start>: in bootblock.asm

Processor starts in real mode

- Processor starts in real mode – works like 16 bit 8088
- eight 16-bit general-purpose registers,
- Segment registers %cs, %ds, %es, and %ss --
> additional bits necessary to generate 20-bit memory addresses from 16-bit registers.

$\text{addr} = \text{seg} \ll 4 + \text{addr}$



**Effective memory translation in the
beginning**

At `_start` in `bootasm.S`:

`%cs=0 %ip=7c00.`

So effective address = $0 \cdot 16 + ip = ip$

bootloader

- **First instruction is 'cli'**
 - **disable interrupts**
- **So that until your code loads all hardware interrupt handlers, no interrupt will occur**

Zeroing registers

Zero data segment registers DS, ES, and SS.

```
xorw  %ax,%ax      # Set %ax to zero
movw  %ax,%ds      # -> Data Segment
movw  %ax,%es      # -> Extra Segment
movw  %ax,%ss      # -> Stack Segment
```

- **zero ax and ds, es, SS**
- **BIOS did not put in anything perhaps**

A not so necessary detail

Enable 21 bit address

seta20.1:

```
inb    $0x64,%al        # Wait for not busy
testb  $0x2,%al
jnz    seta20.1
movb   $0xd1,%al        # 0xd1 -> port 0x64
outb   %al,$0x64
```

seta20.2:

```
inb    $0x64,%al        # Wait for not busy
testb  $0x2,%al
jnz    seta20.2
movb   $0xdf,%al        # 0xdf -> port 0x60
outb   %al,$0x60
```

- Seg:off with 16 bit segments can actually address more than 20 bits of memory. After 0x100000 ($=2^{20}$), 8086 wrapped addresses to 0.
- 80286 introduced 21st bit of address. But older software required 20 bits only. BIOS disabled 21st bit. Some OS needed 21st Bit. So enable it.
- Write to Port 0x64 and 0x60 -> keyboard controller
 - to enable 21st bit out of address translation
 - Why? Before the A20, i.e. 21st bit was introduced, it belonged to keyboard controller
 - For more details see https://en.wikipedia.org/wiki/A20_line
https://en.wikipedia.org/wiki/A20_line

After this

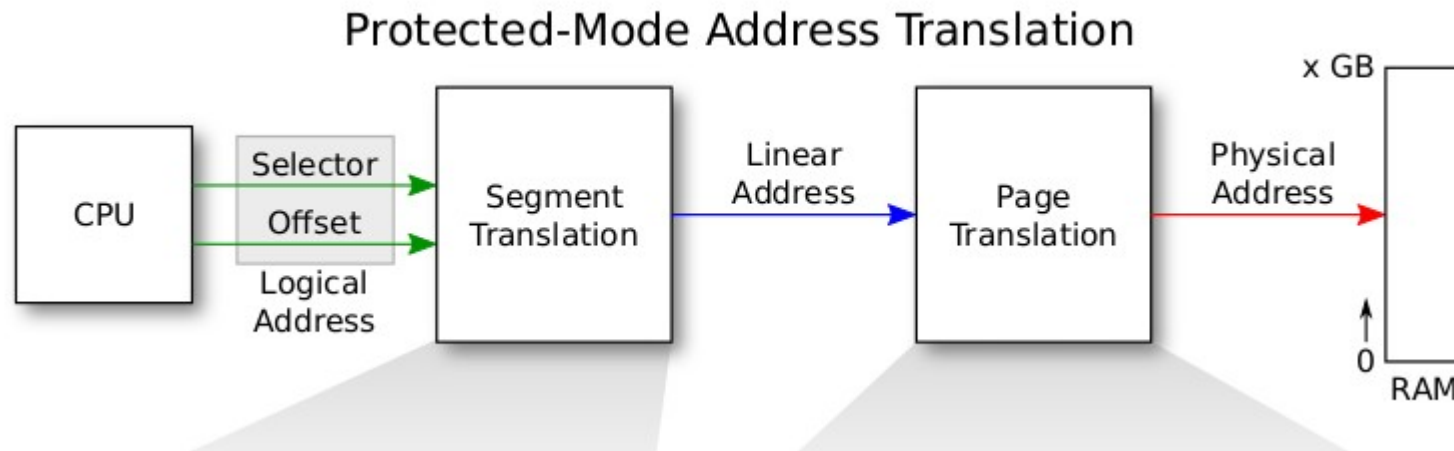
**Some instructions are run
to enter protected mode**

And further code runs in protected mode

Real mode Vs protected mode

- Real mode 16 bit registers
- Protected mode
 - Enables segmentation + Paging both
 - No longer $\text{seg} \times 16 + \text{offset}$ calculations
 - Segment registers is index into segment descriptor table. But segment:offset pairs continue
 - `mov %esp, $32 # SS will be used with esp`
 - More in next few slides
 - Other segment registers need to be explicitly mentioned in instructions
 - `Mov FS:$200, 30`
 - 32 bit registers
 - can address upto 2^{32} memory
 - Can do arithmetic in 32 bits

X86 address : protected mode address translation



Both Segmentation and Paging are used in x86
X86 allows optionally one-level or two-level paging
Segmentation is a must to setup, paging is optional (needs to be enabled)
Hence different OS can use segmentation+paging in different ways

X86 segmentation

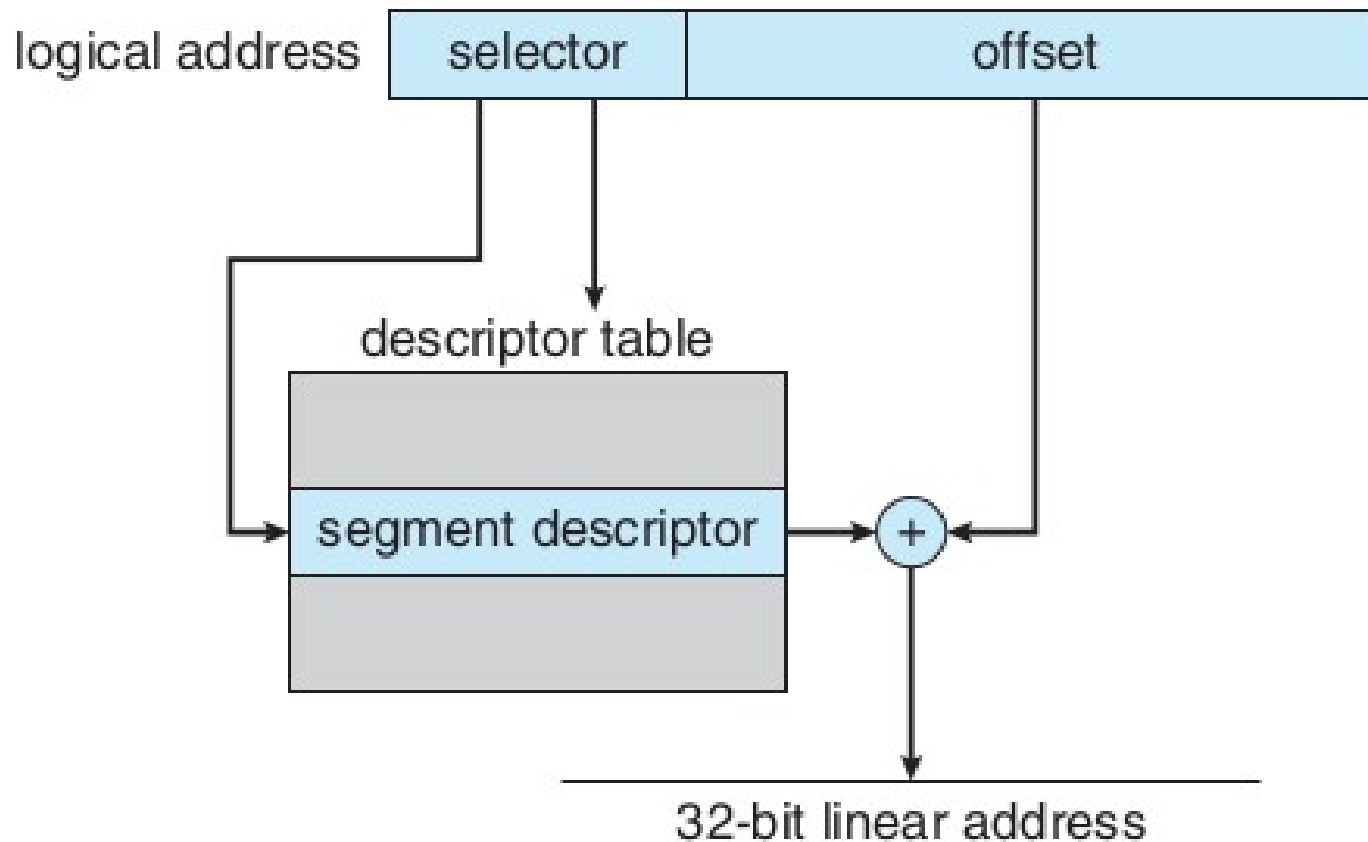


Figure 8.22 IA-32 segmentation.

Paging concept, hierarchical paging

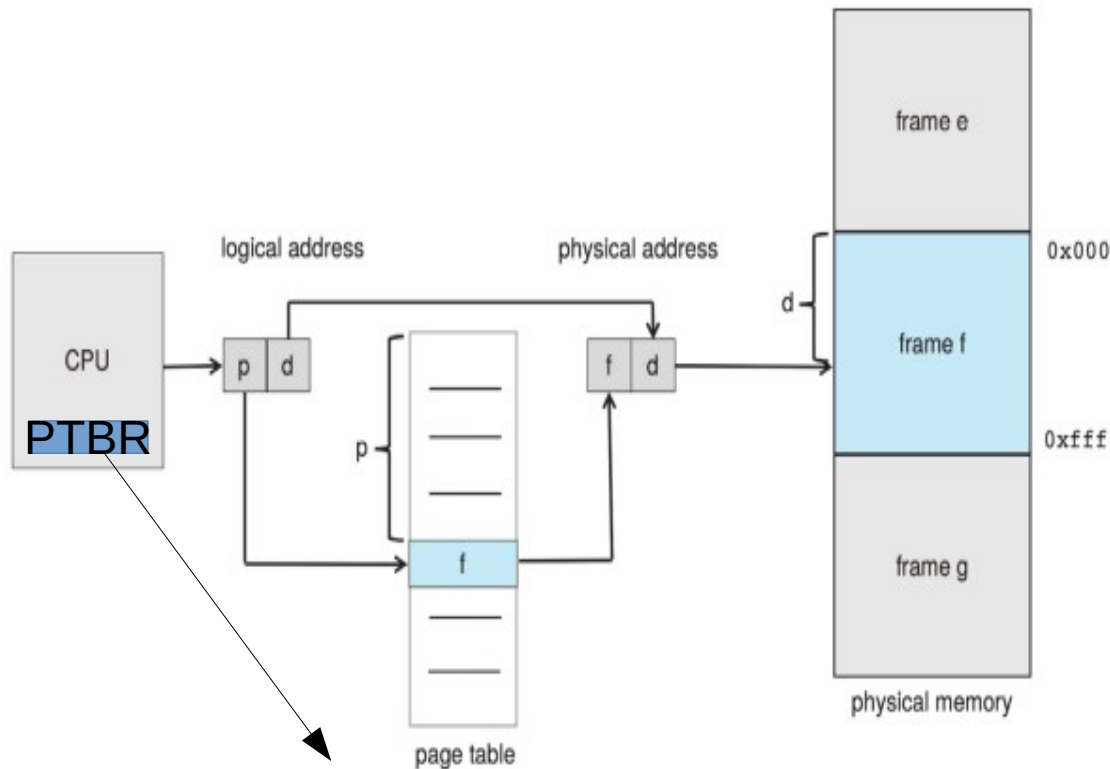


Figure 9.8 Paging hardware.

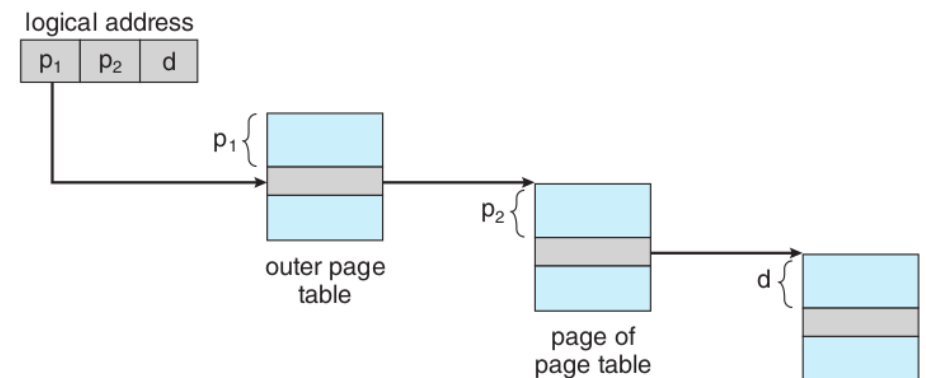


Figure 9.16 Address translation for a two-level 32-bit paging architecture.

X86 paging

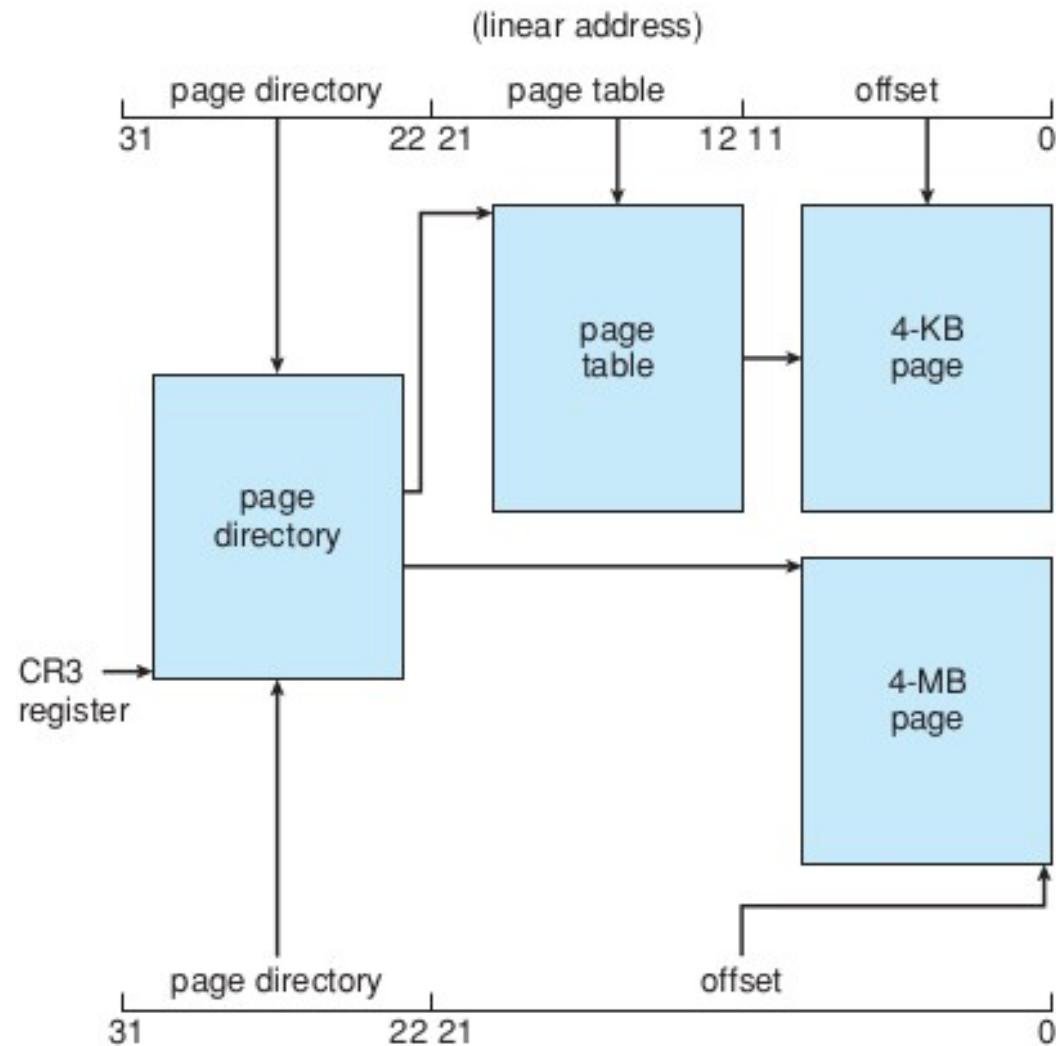
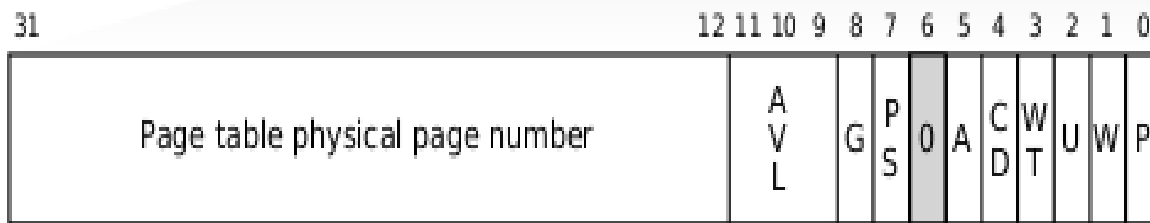


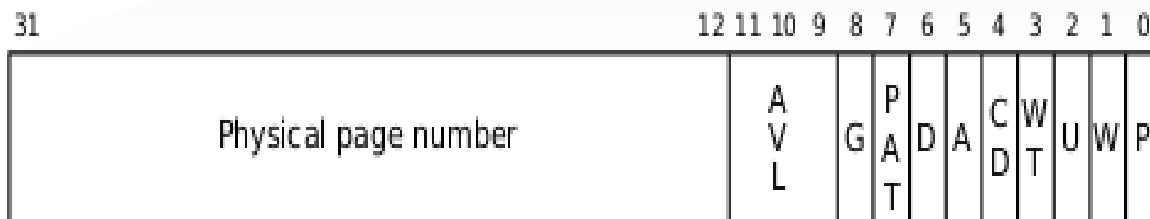
Figure 8.23 Paging in the IA-32 architecture.

Page Directory Entry (PDE)

Page Table Entry (PTE)



PDE



PTE

P Present

W Writable

U User

WT 1=Write-through, 0=Write-back

CD Cache disabled

A Accessed

D Dirty

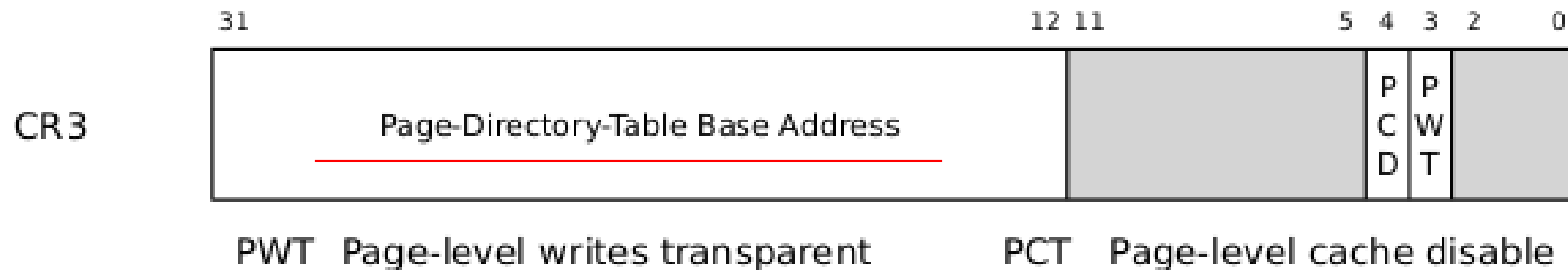
PS Page size (0=4KB, 1=4MB)

PAT Page table attribute index

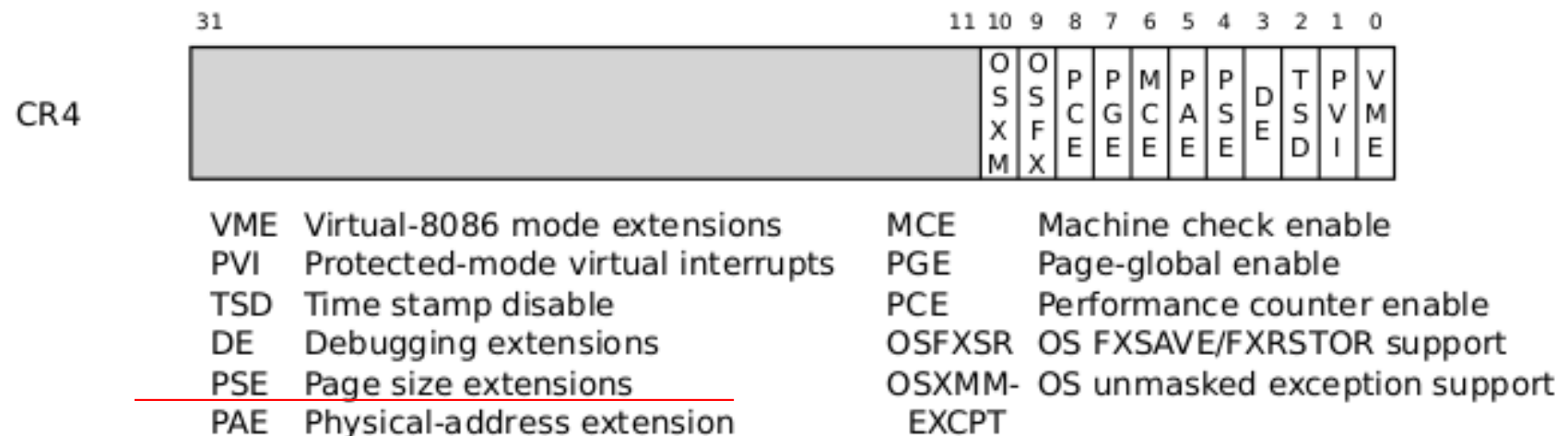
G Global page

AVL Available for system use

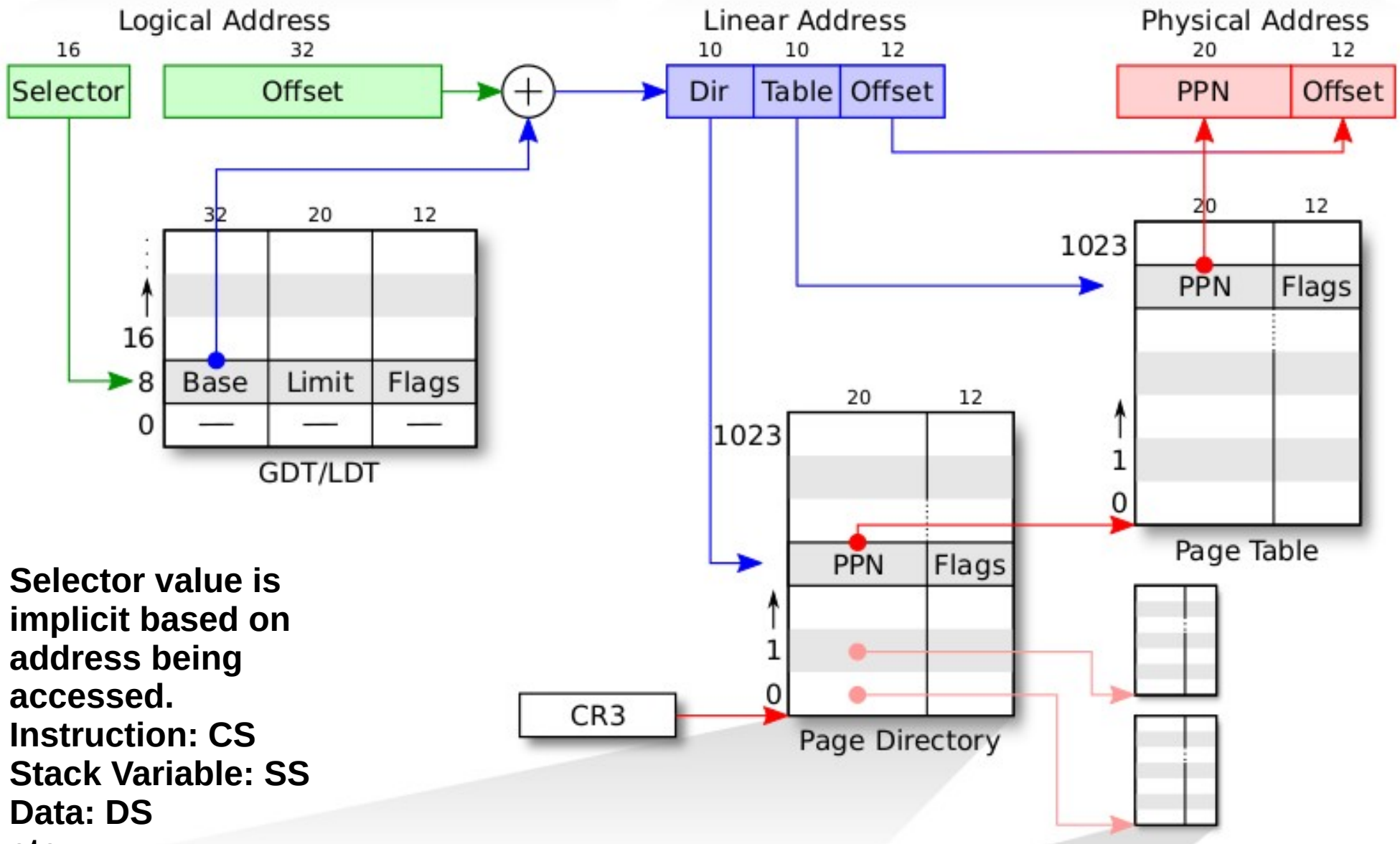
CR3



CR4



Segmentation + Paging



Segmentation + Paging setup of xv6

- **xv6 configures the segmentation hardware by setting Base = 0, Limit = 4 GB**
 - translate logical to linear addresses without change, so that they are always equal.
 - **Segmentation is practically off**
- **Once paging is enabled, the only interesting address mapping in the system will be linear to physical.**
 - In xv6 paging is NOT enabled while loading kernel
 - After kernel is loaded 4 MB pages are used for a while
 - Later the kernel switches to 4 kB pages!

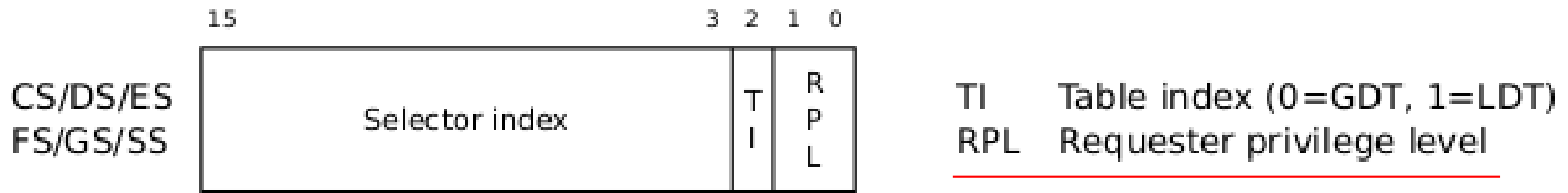
GDT Entry

31				16		15				0			
<u>Base 0:15</u>						<u>Limit 0:15</u>							
63		56		55 52		51 48		47		40 39		32	
<u>Base 24:31</u>		Flags		<u>Limit 16:19</u>		Access Byte		<u>Base 16:23</u>					

asm.h

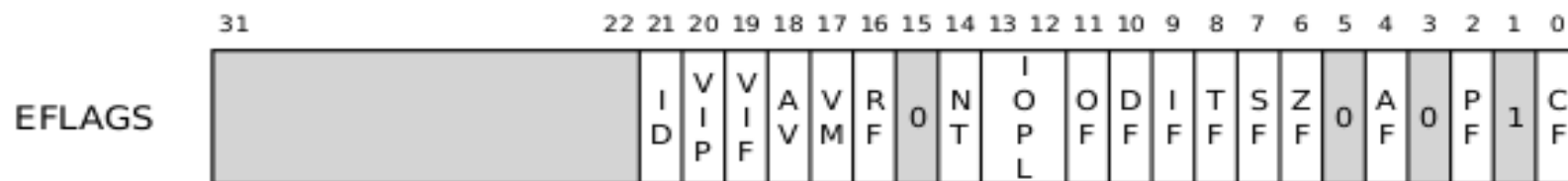
```
#define SEG_ASM(type,base,lim) \
    .word (((lim) >> 12) & 0xffff), ((base) & 0xffff); \
    .byte (((base) >> 16) & 0xff), (0x90 | (type)), \
        (0xC0 | (((lim) >> 28) & 0xf)), (((base) >> 24) & 0xff)
```

Segment selector



Note in 16 bit mode, segment selector is 16 bit, here it's 13 bit + 3 bits

EFLAGS register



Status flags

CF	Carry flag
PF	Parity flag
AF	Auxiliary carry flag
ZF	Zero flag
SF	Sign flag
OF	Overflow flag

Control flags

DF Direction flag

System flags

TF	Trap flag
IF	Interrupt enable flag
IOPL	I/O privilege level
NT	Nested task

RF

Resume flag

VM	Virtual-8086 mode
AC	Alignment check
VIF	Virtual intr. flag
VIP	Virtual intr. pending
ID	ID flag

lgdt gdtdesc

...

Bootstrap GDT

.p2align 2 # force 4 byte
alignment

gdt:

SEG_NULLASM # null seg

SEG_ASM(STA_X|STA_R, 0x0,
0xffffffff) # code seg

SEG_ASM(STA_W, 0x0,
0xffffffff)

data seg

gdtdesc:

.word (gdtdesc - gdt - 1)

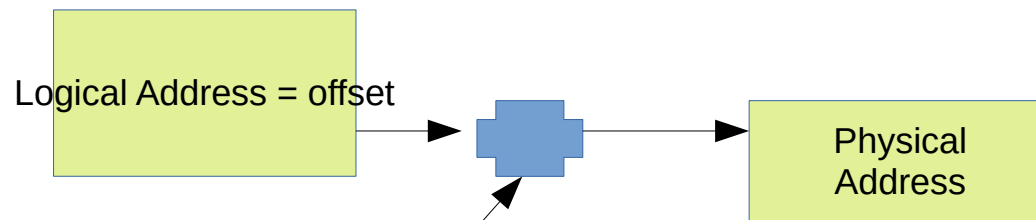
sizeof(gdt) - 1

.long gdt

lgdt

- load the processor's (GDT) register with the value gdtdesc which points to the table gdt.
- **table gdt** : The table has a null entry, one entry for executable code, and one entry to data.
- all segments have a base address of zero and the maximum possible limit
- The code segment descriptor has a flag set that indicates that the code should run in 32-bit mode
- With this setup, when the boot loader enters protected mode, logical addresses map one-to-one to physical addresses.
- At **gdtdesc** we have this **data**
2, <4 byte addr of gdt>
Total 6 bytes
 - GDTR is : <address 4 byte>, <table limit 2 byte>
 - So lgdt gdtdesc loads these two values in GDTR

**bootasm.S after "lgdt gdtDESC"
till jump to "entry"**



Base	Limit	Permissions
0	4GB	Write
0	4GB	Read, Execute
0	0	0

GDT

Still
Logical Address =
Physical address!

But with GDT in picture
and
Protected Mode
operation

During this time,

**Loading kernel from
ELF into physical
memory**

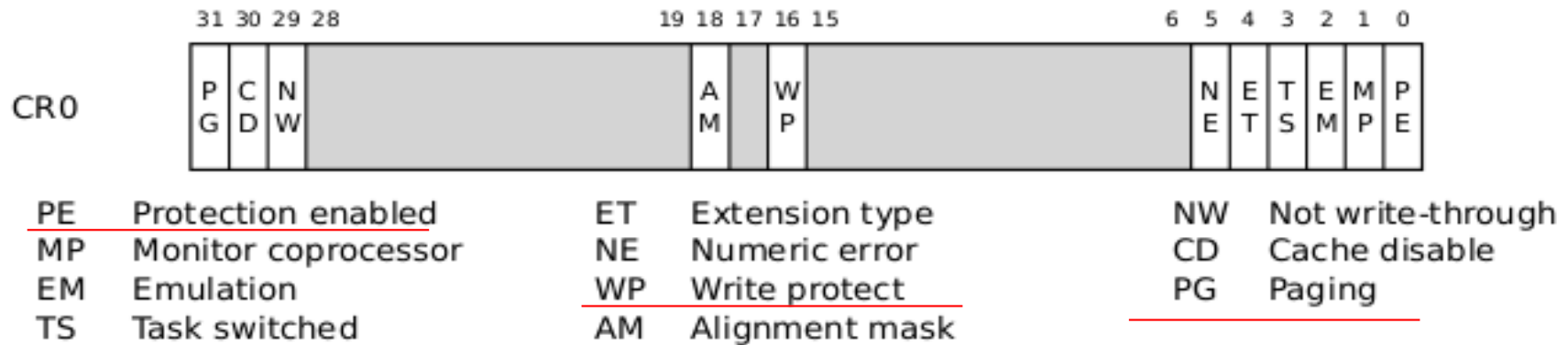
**Addresses in "kernel"
file translate to same
physical address!**

Prepare to enable protected mode

- Prepare to enable protected mode by setting the 1 bit (CR0_PE) in register %cr0

```
movl    %cr0, %eax
orl     $CR0_PE, %eax
movl    %eax, %cr0
```

CRO



PG: Paging enabled or not WP: Write protection on/off
PE: Protection Enabled --> protected mode.

Complete transition to 32 bit mode

ljmp \$(SEG_KCODE<<3), \$start32

Complete the transition to 32-bit protected mode by using a long jmp

to reload %cs (=1) and %eip (=start32).

Note that 'start32' is the address of next instruction after ljmp.

Note: The segment descriptors are set up with no translation (that is 0-4GB setting), so that the mapping is still the identity mapping.

Jumping to “C” code

```
movw  $(SEG_KDATA<<3), %ax  # Our data
segment selector

movw  %ax, %ds               # -> DS: Data
Segment

movw  %ax, %es               # -> ES: Extra
Segment

movw  %ax, %ss               # -> SS: Stack
Segment

movw  $0, %ax                # Zero segments not
ready for use

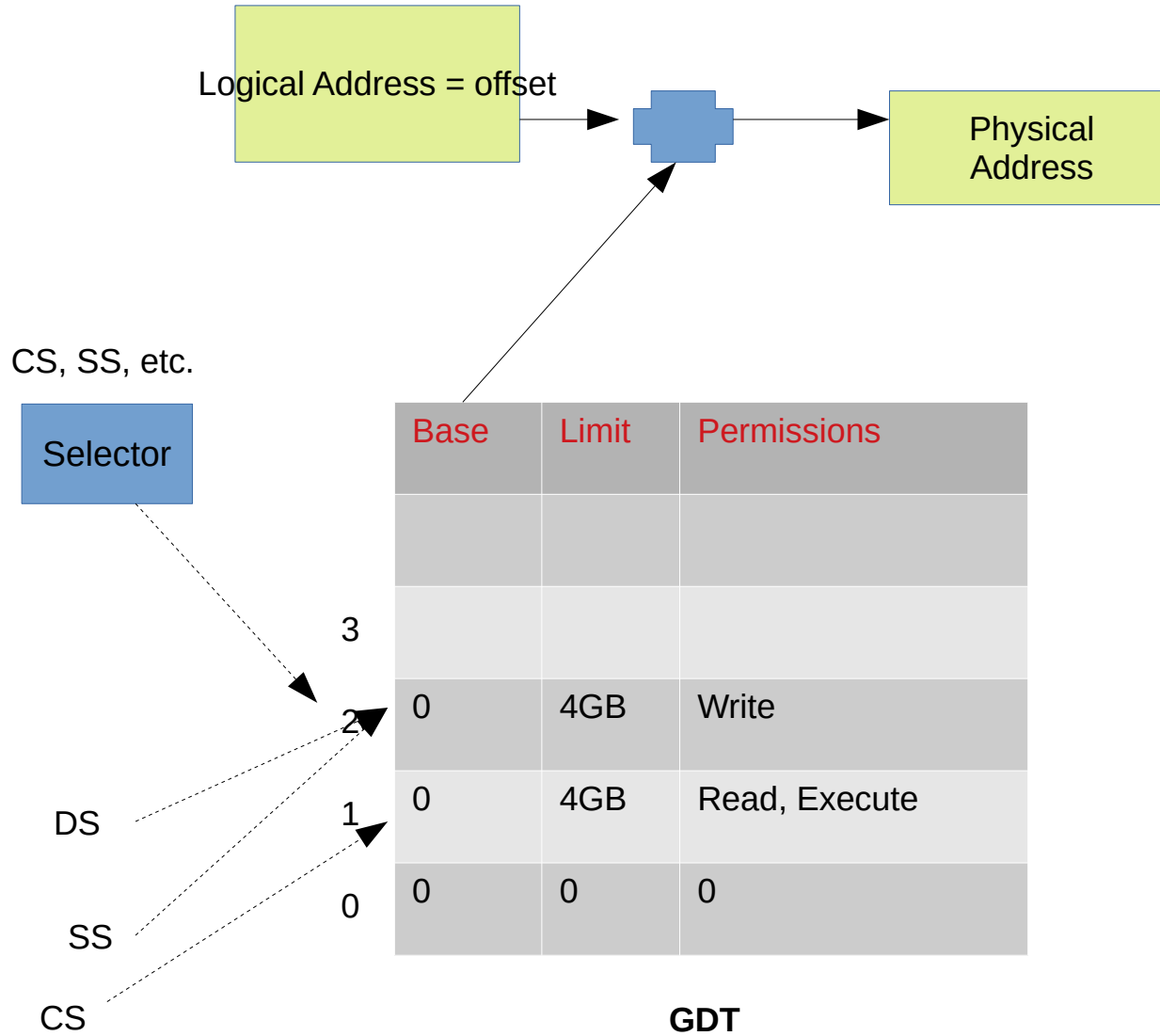
movw  %ax, %fs               # -> FS

movw  %ax, %gs               # -> GS

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

- **Setup Data, extra, stack segment with SEG_KDATA (=2), FS & GS (=0)**
- **Copy “\$start” i.e. 7c00 to stack-ptr**
 - It will grow from 7c00 to 0000
- **Call bootmain() a C function**
 - In bootmain.c

Setup now



bootmain(): already in memory, as part of 'bootblock'

- `bootmain.c` , expects to find a copy of the kernel executable on the disk starting at the second sector (sector = 1).
 - Why?
- The kernel is an ELF format binary
- Bootmain loads the first 4096 bytes of the ELF binary. It places the in-memory copy at address 0x10000
- `readseg()` is a function that runs OUT instructions in particular IO ports, to issue commands to read from Disk

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

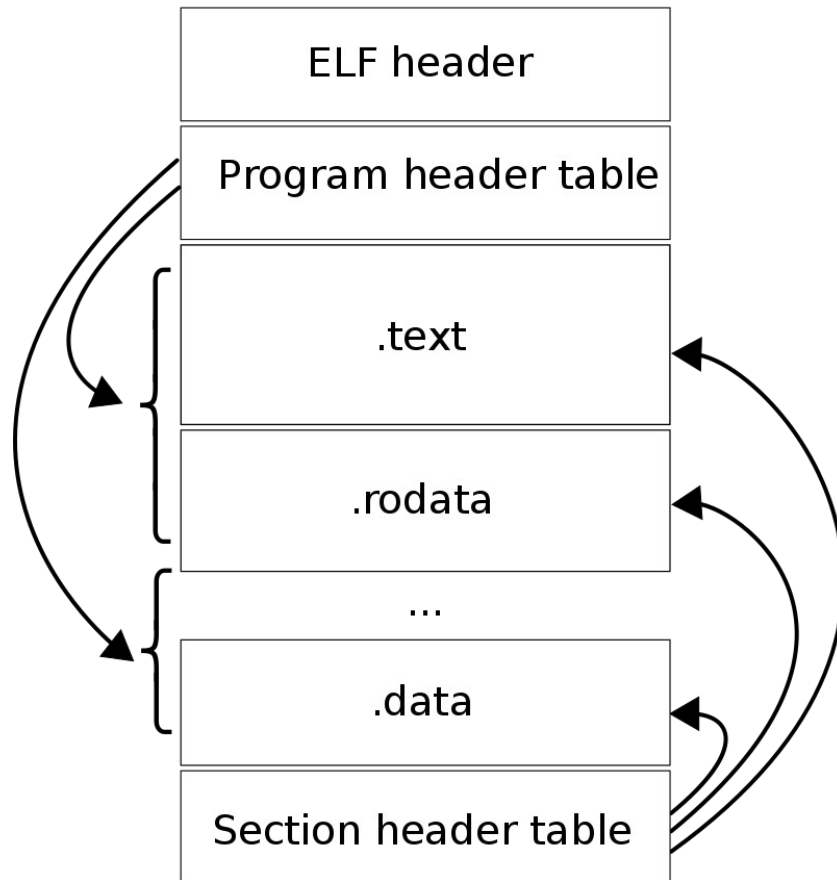
    elf = (struct elfhdr*)0x10000; // scratch
    space

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

bootmain()

- Check if it's really ELF or not `// Is this an ELF executable?`
- Next load kernel code from ELF file "kernel" into memory `if(elf->magic != ELF_MAGIC)`
 `return; // let bootasm.S handle error`

ELF



```
struct elfhdr {
```

```
    uint magic; // must equal  
    ELF_MAGIC
```

```
    uchar elf[12];
```

```
    ushort type;
```

```
    ushort machine;
```

```
    uint version;
```

```
    uint entry;
```

```
    uint phoff; // where is program  
    header table
```

```
    uint shoff;
```

```
    uint flags;
```

```
    ushort ehsize;
```

```
    ushort phentsize;
```

```
    ushort phnum; // no. Of program  
    header entries
```

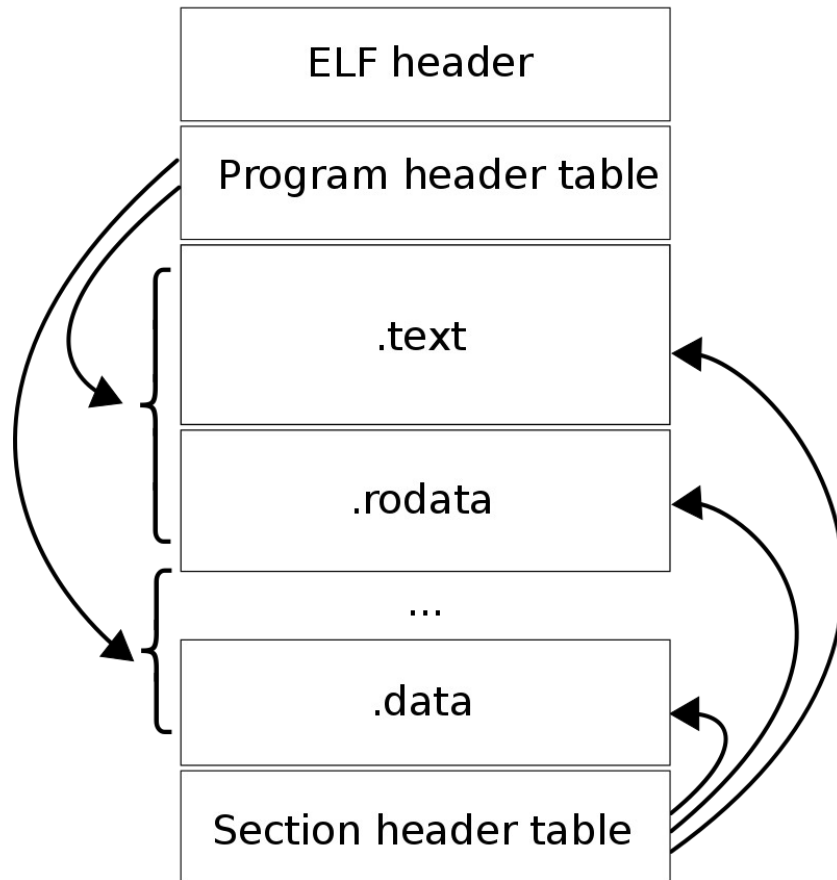
```
    ushort shentsize;
```

```
    ushort shnum;
```

```
    ushort shstrndx;
```

```
};
```


ELF



// Program header

```
struct proghdr {
```

```
    uint type; // Loadable segment ,  
Dynamic linking information ,  
Interpreter information , Thread-  
Local Storage template , etc.
```

```
    uint off; //Offset of the segment  
in the file image.
```

```
    uint vaddr; //Virtual address of  
the segment in memory.
```

```
    uint paddr; // physical address  
to load this program, if PA is  
relevant
```

```
    uint filesz; //Size in bytes of the  
segment in the file image.
```

```
    uint memsz; //Size in bytes of  
the segment in memory. May be  
0.
```

```
    uint flags;
```

```
    uint align;
```

```
};
```

Run 'objdump -x -a kernel | head -15' & see this

kernel: file format elf32-i386
kernel
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0010000c

Code to be
loaded at
KERNBASE +
KERNLINK

Diff
between
memsz &
filesz, will
be filled
with zeroes
in memory

Program Header:

LOAD off 0x00001000 vaddr 0x80100000 paddr 0x00100000 align 2**12
filesz 0x0000a516 memsz 0x000154a8 flags rwx
STACK off 0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**4
filesz 0x00000000 memsz 0x00000000 flags rwx

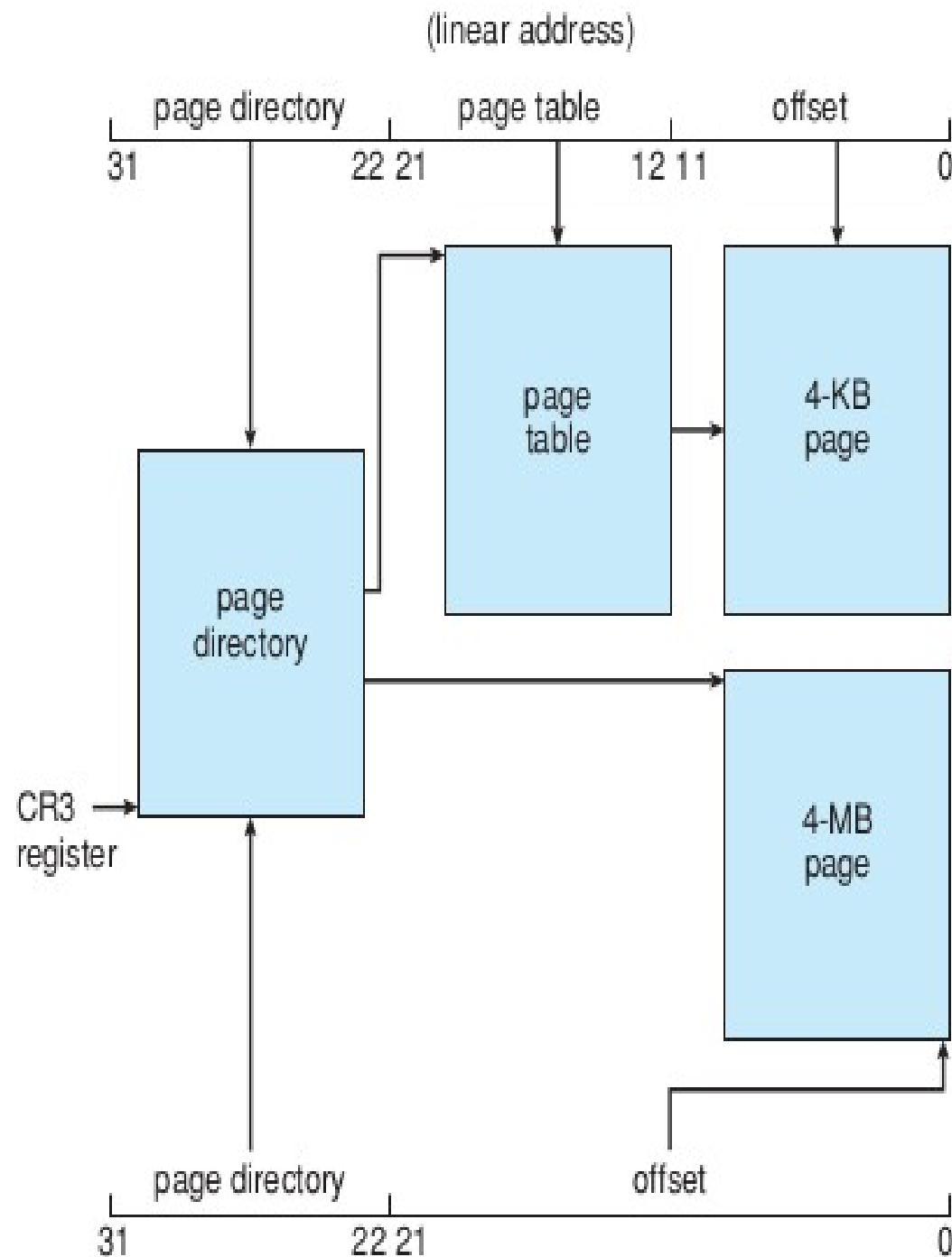
Stack :
everything
zeroes

Load code from ELF to memory

```
// Load each program segment (ignores ph flags).
ph = (struct proghdr*)((uchar*)elf + elf->phoff);
eph = ph + elf->phnum;
// Abhijit: number of program headers
for(; ph < eph; ph++){
    // Abhijit: iterate over each program header
    pa = (uchar*)ph->paddr;
    // Abhijit: the physical address to load program
    /* Abhijit: read ph->filesz bytes, into 'pa',
       from ph->off in kernel/disk */
    readseg(pa, ph->filesz, ph->off);
    if(ph->memsz > ph->filesz)
        stosb(pa + ph->filesz, 0, ph->memsz - ph-
>filesz); // Zero the reminder section*/
}
```

Jump to Entry

```
// Call the entry point from the ELF header.  
// Does not return!  
/* Abhijit:  
 * elf->entry was set by Linker using kernel.ld  
 * This is address 0x80100000 specified in kernel.ld  
 * See kernel.asm for kernel assembly code).  
 */  
entry = (void (*)(void))(elf->entry);  
entry();
```

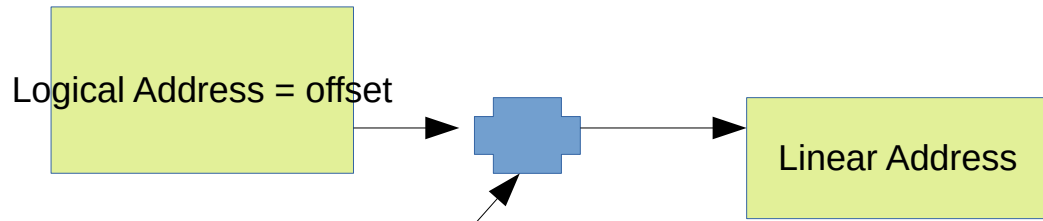


**To understand
further
code**

**Remember: 4
MB pages
are possible**

Figure 8.23 Paging in the IA-32 architecture.

From entry:
Till: inside main(), before kvmalloc()



CS, SS, etc.

Selector

	Base	Limit	Permissions
3			
2	0	4GB	Write
1	0	4GB	Read, Execute
0	0	0	0

GDT

CR3_

entryptgdir



■

■

3

2

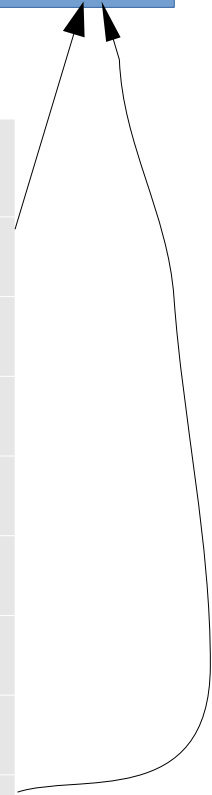
1

0

RAM

4MB

0

[illegible]

From entry:
Till: inside main(), before kvmalloc()

Logical Address = offset

Physical Addr

RAM

4MB

0

CS, SS, etc.

Selector

	Base	Limit	Permissions
3			
2	0	4GB	Write
1	0	4GB	Read, Execute
0	0	0	0

GDT

DS

SS

CS

Even now, every Logical
address = Physical address,
but through Page dir

CR3

entrypgdir

512	0	P,W,PS
.		
.		
.		
3		
2		
1		
0	0	P,W,PS

entrypgdir in main.c, is used by entry()

```
__attribute__((__aligned__(PGSIZE)))  
pde_t entrypgdir[NPDENTRIES] = {
```

```
    // Map VA's [0, 4MB) to PA's [0, 4MB)  
    [0] = (0) | PTE_P | PTE_W | PTE_PS,
```

```
    // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB). This is entry 512  
    [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,  
};
```

```
#define PTE_P      0x001 // Present  
#define PTE_W      0x002 // Writeable  
#define PTE_U      0x004 // User  
#define PTE_PS     0x080 // Page Size  
#define PDXSHIFT   22    // offset of  
PDX in a linear address
```

This is entry page directory during entry(), beginning of kernel Mapping 0:0x400000 (i.e. 0: 4MB) to physical addresses 0:0x400000. is required as long as entry is executing at low addresses, but will eventually be removed.

This mapping restricts the kernel instructions and data to 4 Mbytes.

entry() in entry.S

entry:

```
    movl    %cr4, %eax
    orl     $(CR4_PSE), %eax
    movl    %eax, %cr4
    movl    $
(V2P_WO(entrypgdir)), %eax
    movl    %eax, %cr3
    movl    %cr0, %eax
    orl     $(CR0_PG|CR0_WP),
%eax
    movl    %eax, %cr0
    movl    $(stack +
KSTACKSIZE), %esp
    mov     $main, %eax
    jmp     *%eax
```

- # Turn on page size extension for 4Mbyte pages
- # Set page directory. 4 MB pages (temporarily only. More later)
- # Turn on paging.
- # Set up the stack pointer.
- # Jump to main(), and switch to executing at high addresses. The indirect call is needed because the assembler produces a PC-relative instruction for a direct jump.
-

More about entry()

```
movl $(V2P_WO(entrypgdir)), %eax  
movl %eax, %cr3
```

-> Here we use
physical address
using V2P_WO
because paging is not
turned on yet

- **V2P is simple:
subtract
0x80000000 i.e.
KERNBASE from
address**

More about entry()

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

- But we have already set 0'th entry in pgdir to address 0
- So it still works!

This turns on paging
After this also, entry() is running and processor is executing code at lower addresses

entry()

```
movl $(stack +  
KSTACKSIZE), %esp
```

```
mov $main, %eax
```

```
jmp *%eax
```

```
.comm stack,  
KSTACKSIZE
```

```
# Abhijit: allocate here 'stack' of size  
= KSTACKSIZE
```

- # Set up the stack pointer.
- # Abhijit: +KSTACKSIZE is done as stack grows downwards
- # Jump to main(), and switch to executing at high addresses. The indirect call is needed because the assembler produces a PC-relative instruction for a direct jump.

bootmasm.S bootmain.c: Steps

- 1) Starts in “real” mode, 16 bit mode. Does some misc legacy work.**
- 2) Runs instructions to do MMU set-up for protected-mode & only segmentation (0-4GB, identity mapping), changes to protected mode.**
- 3) Reads kernel ELF file and loads it in RAM, as per instructions in ELF file**
- 4) Sets up paging (4 MB pages)**
- 5) Runs main() of kernel**

Code from bootasm.S bootmain.c is over!
Kernel is loaded.
Now kernel is going to prepare itself