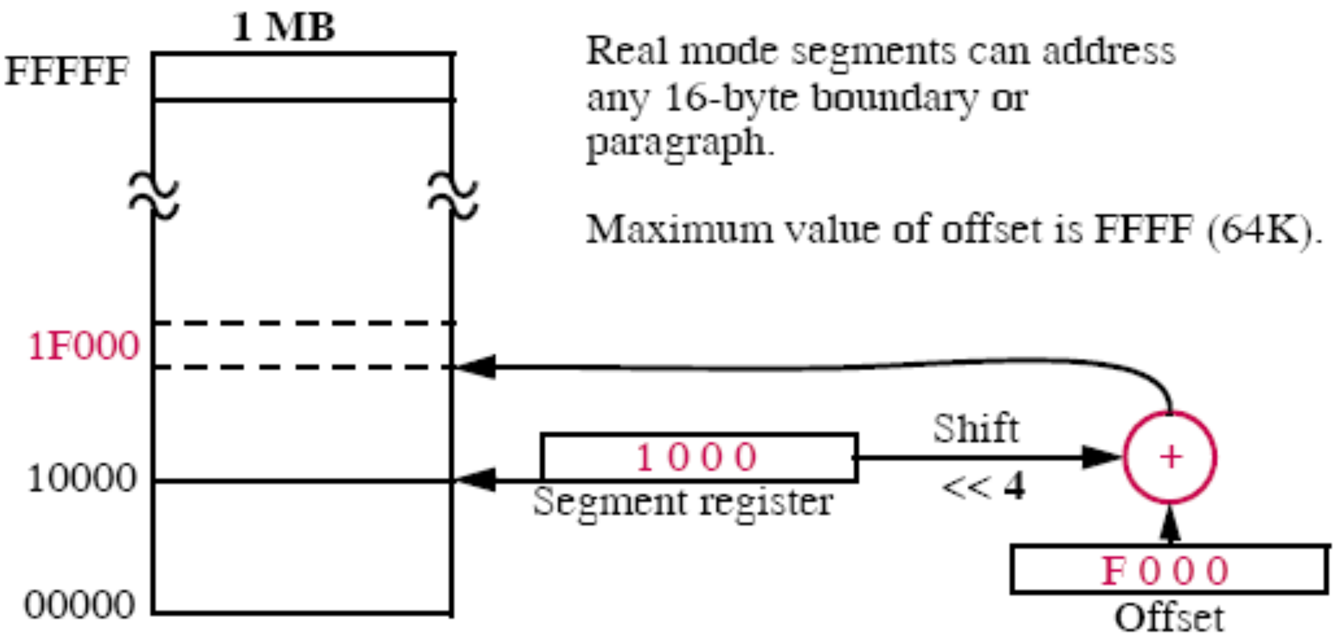
12 Linux User Programs

Multitasking

- Multitasking
  - Allows programs to take turns using processor
  - Scheduler decides to force user programs to yield to one another even when they are not finished
  - Task state segment: bumped process saves its current state into a region in memory
- Memory protection
  - Multitasking requires memory protection
  - Linux uses paging to implement a memory protection
  - Every memory reference in a user program is turned into a request for memory access which is required to go across the desk of the paging system for approval

Cf) Real Mode Memory Addressing

- Only mode available to the 8086 and 8088.
  - Allow the processor to address only the first 1MB of memory.
  - DOS requires real mode.
- ***Segments and Offsets:***
  - Effective address = Segment address + an offset.



Protected Mode

- To change to protected mode from DOS

```
MOV EAX, CR0
OR EAX, 1        ; Make the least significant bit 1.
MOV CR0, EAX     ; Here goes!
```

- x86 CR0 Register

Bits	Label	Full Name
31	PG	Paging Enable
30	CD	Cache Disable
29	NW	No Write Through
18	AM	Alignment Mask
15	WP	Write Protect
5	NE	Numeric Error Enable
4	ET	Extension Type (287 vs. 387)
3	TS	Task Switched
2	EM	Emulate Math Chip
1	MP	Math Chip Present
0	PE	Protected Mode Enable

Protected Mode

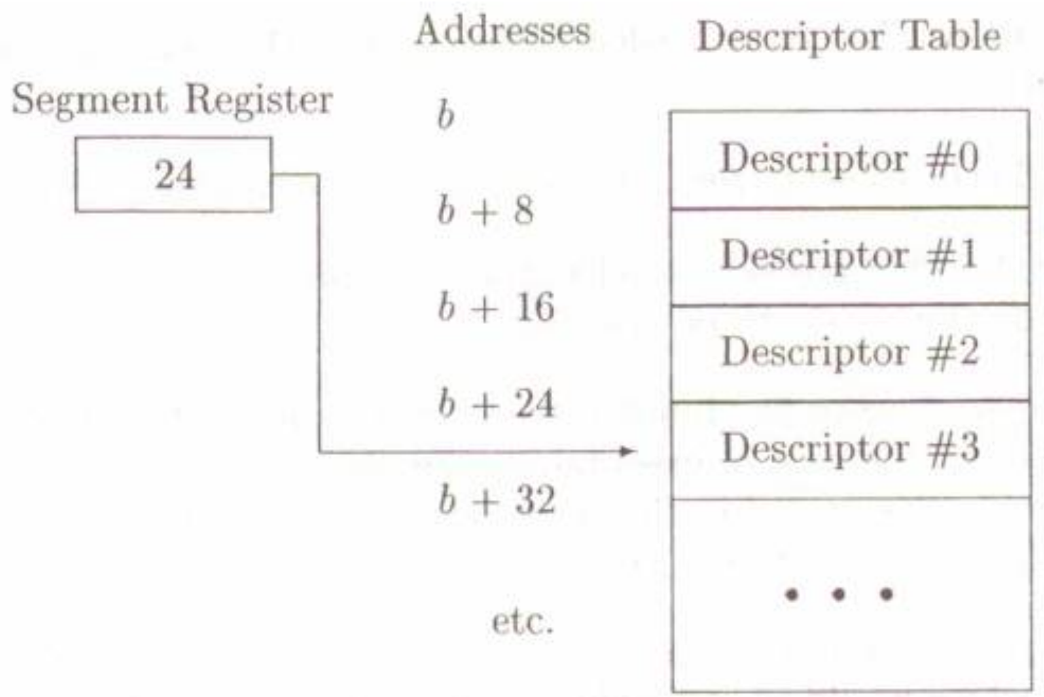
- In protected mode
  - all memory references depend on *global descriptor table*
  - instructions are fetched from a different segment
  - each entry in the interrupt table is a specially formatted eight-byte entry; before going into protected mode, a new interrupt table must be created

Protected Mode Segmentation

- A protected mode memory segment

Protected Mode Segmentation

- segment, segment descriptor, descriptor table, selector

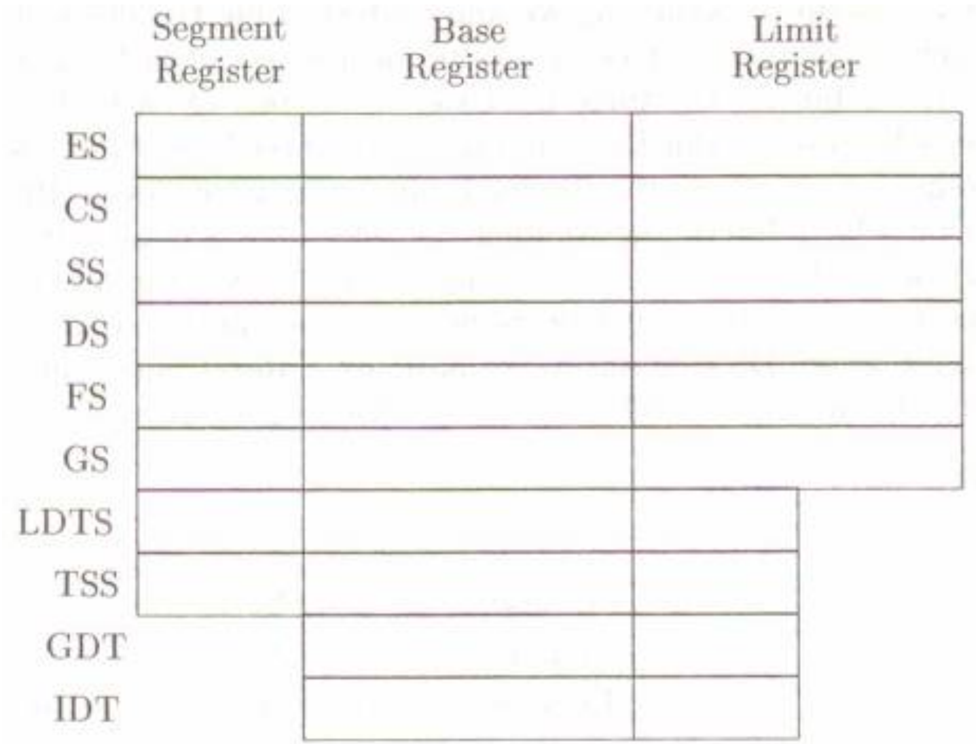


Protected Mode Memory Segments

- Characteristics of a protected mode memory segment
  - Base : the address of the first byte of the segment
  - Limit: the last valid offset address
  - Access permissions: permissions to read, write, and execute can be specified
  - Access privilege level: the least privileged value that CPL (current privilege level) may have in order for access to be granted

Protected Mode Memory Segments

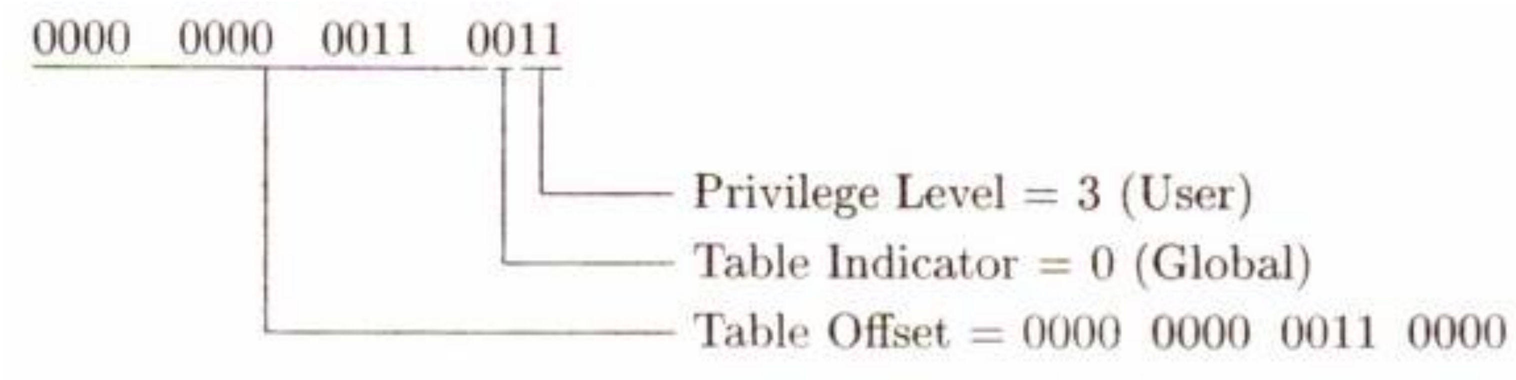
- Selector, base, and limit registers



- Special memory segments, GDT and IDT

Selectors

- Selector encoding



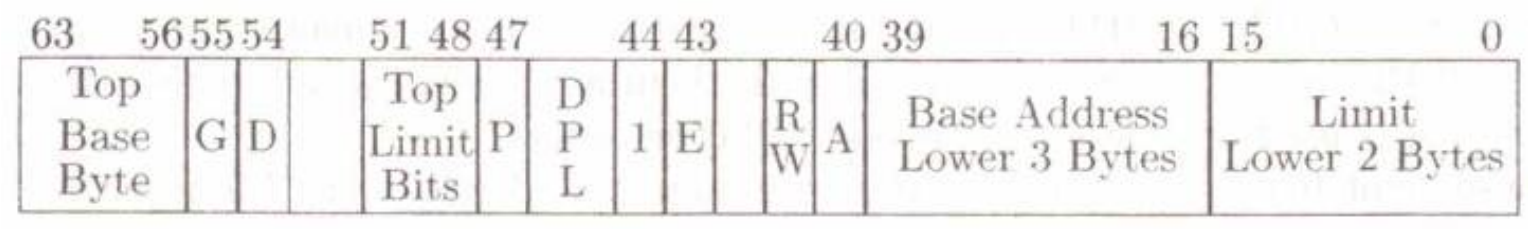
Segment Operations

- Segment load and store commands

	Load	Store	MOV Load	MOV Store
ES	LES		X	X
CS				X
SS	LSS		X	X
DS	LDS		X	X
FS	LFS		X	X
GS	LGS		X	X
LDTS	LLDT	SLDT		
TSS	LTR	STR		
GDT	LGDT	SGDT		
IDT	LIDT	SIDT		

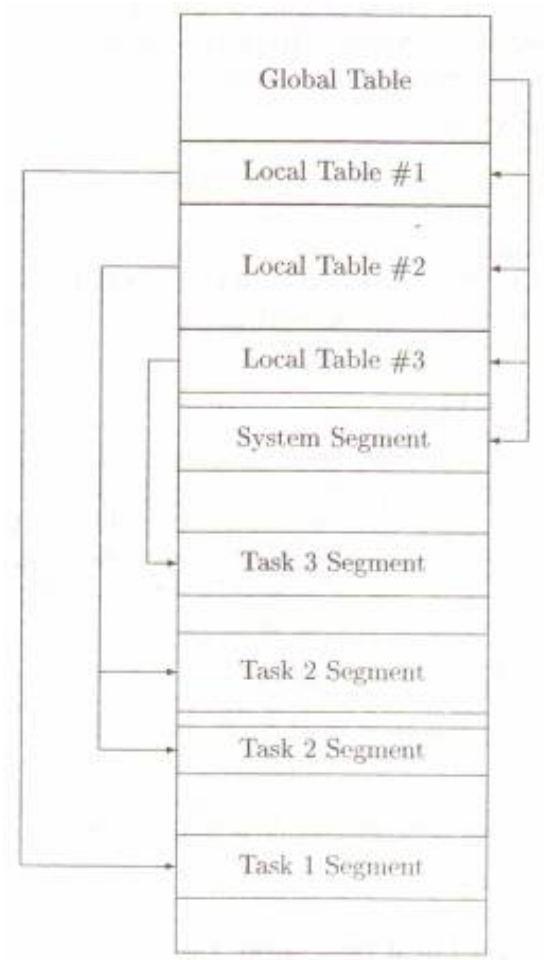
Descriptor Encoding

- Segment descriptor layout



Task Isolation with LDT

- Memory with protected segments



Task Isolation with LDT

- Consider this code

```
       MOV AX, 0     ;
                     ; AX is a 16-bit selector.
                     ; Bit 2 is 0. ; We're pointing into the global table.
                     ; Bits 0 and 1 are 0
                     ; That's system level privilege.
HAK:   LLDT AX       ;
                     ; AX is pointing at some descriptor
                     ; It could be a local table descriptor
                     ; If so it's probably not our own local table.
                     ; And in that case we have just borrowed someone
                     ; else's private descriptor table and we can
                     ; access anything listed in there.
       ADD AX, 8     ; Maybe not.
                     ; Go to the next descriptor
       JMP HAK       ; The local tables are in here.
                     ; We'll find them.
```

- Memory protection by itself doesn't work; user processes must not be allowed to execute code like the above

Setting Up the GDT

```
MOV AX, 2000H     ; Set up a segment for the scratch area.
MOV DS, AX        ;
MOV AX, 4FH       ; GDT's limit is a 2 byte value.
MOV [DS:0], AX    ; Store 79 into the first 2 bytes of
                  ; scratch.
MOV EAX, 10000H   ; GDT's base address is a 4 byte value.
MOV [DS:2], EAX   ; Store this 4 byte address
                  ; 2 bytes into the scratch.
                  ; (Don't overwrite the first two bytes.)
LGDT [DS:0]       ;
; This loads the both the GDT base address register
                  ; and the GDT limit register
                  ; i.e., load all 6 bytes.
```

Paging

- x86 I/O address vs. memory address

IN AL,[DX]   ; where DX=FC30H

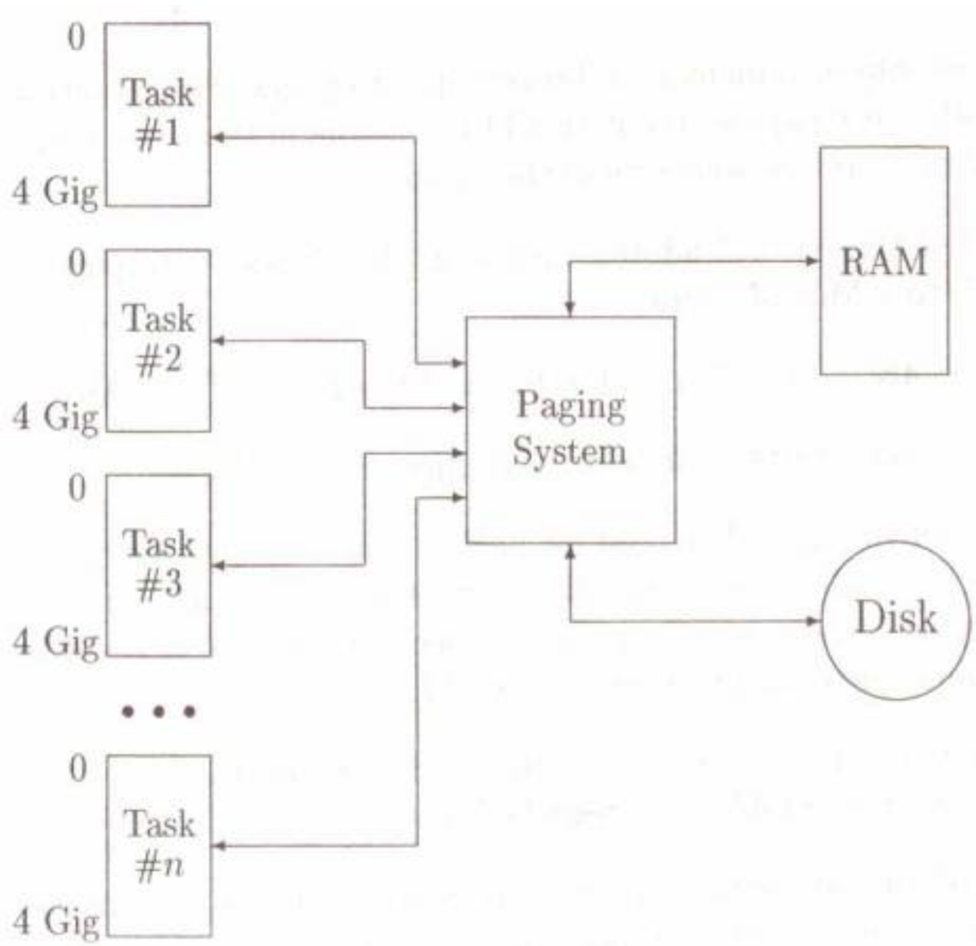
→ 16 bits of DX is directly applied to address bus

MOV AL,[0FC30H]

→ memory address FC30 undergo a two-step process of *segmentation* and ***paging*** *

* Linux uses only paging

Virtual Memory



- x86 virtual memory system
  - Each process has address space of 0 ~ 4G
  - Process's address space is broken up into *pages*
  - Only parts of pages are loaded into memory
  - Each process has its own set of *page tables* to translate the virtual addresses into physical ones

4K pages

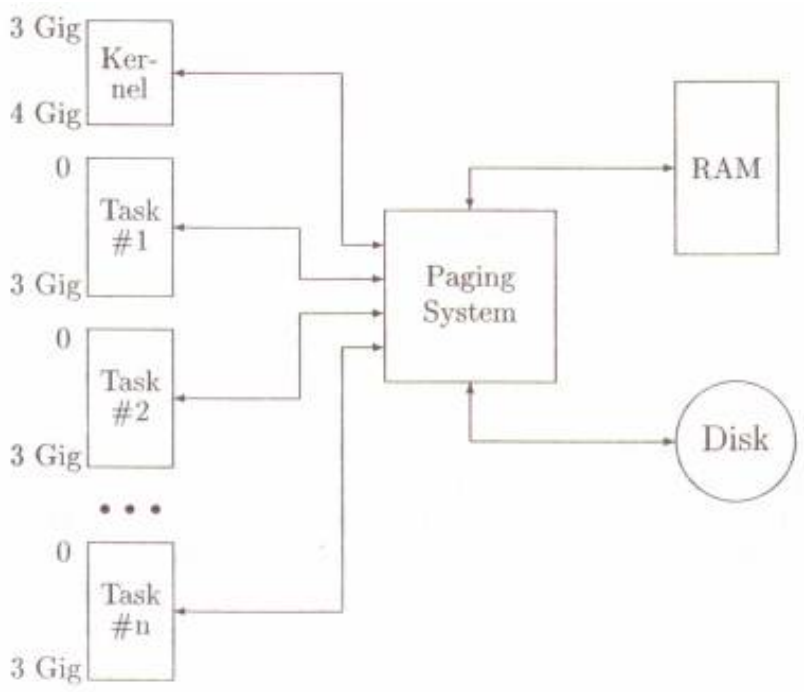
- In order to set up a paging system
  - The available physical memory is broken up into 4K size page frames
  - Each task's 4 Gig virtual address space is also divided into pages
  - The portion of the disk used for paging, called swap space, is also divided into pages
  - The page tables used to translate virtual addresses are set up in units of pages

Paging vs. swapping

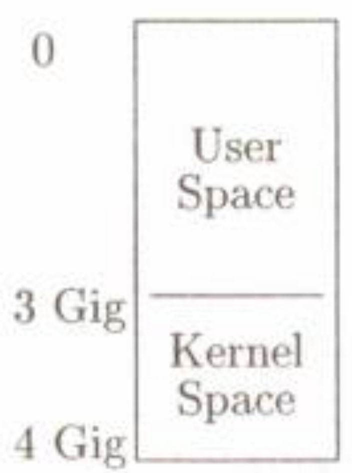
- Swapping
  - Copying entire process to and from the disk
- Page faults
  - When a machine instruction refers to a memory location actually on disk, this is called *page fault*
  - When a page fault occurs, some page frame in RAM has to be chosen so that the needed page may be loaded into it from the disk
  - If no page frame is free, then some page of memory has to be chosen for transfer to the disk so as to free up a page frame

Kernel addresses

- Linux virtual memory system

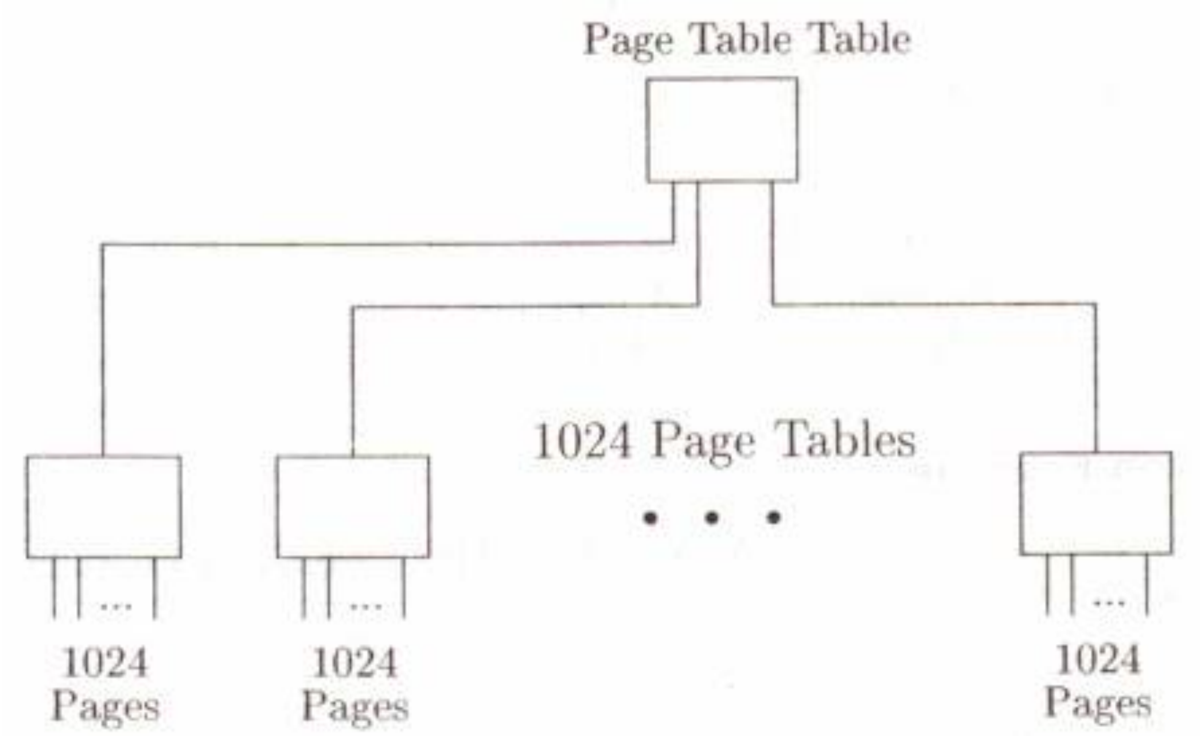


- Single process view of memory



Address Translation

- Two-tier page tables



Kernel

- Kernel pages
  - The Linux paging never pages kernel memory out to disk
  - The Linux kernel is therefore always in RAM
- Kernel modules
  - It has been possible to add and delete kernel memory
  - *insmod* command install module
  - *rmmod* command remove module
  - *free* inspect memory to find out the amount of memory available

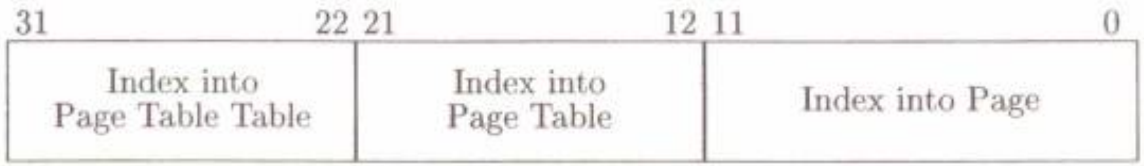
Address Translation

- x86 paging obtains its translation from page tables set up by the operating system
- Since the translation process is hard-wired into the processor, x86 page tables must have the two-tired structure

- X86 page table
  - Each table has 1024 four-byte entries -> 4K size -> same to page size
  - page table lists the location of 1024 page tables
  - Total memory referenced by a page table is 1024 * 1024 * 4096 = 4 GB -> size of virtual address space

Address Translation

- Address parsing

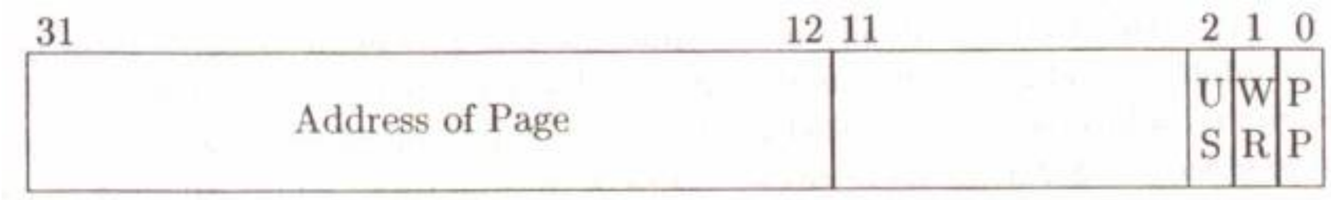


- Example of a virtual address of '1A2B3C4DH'

```
  1    A    2    B    3    C    4    D
0001 1010 0010 1011 0011 1100 0100 1101    virtual address
0001 1010 00                               index into page table table
            10 1011 0011                   index into page table
                         1100 0100 1101    index into page
```

  - 104th entry of the page table table
  - 691st entry of the page table
  - Offset is 0C4DH
  - If the beginning of the page is 12340000H the effective address is 1234C4DH

Page table entries



- PP : page is located in RAM
- If PP is not set
  - It is located on disk -> page fault
  - The address is not valid at all -> segmentation fault
- CR3 register
  - The physical address of the page table table

Program Segments

- Non-writeable memory
  - W_R bit
    - Set means that page is writeable
    - Clear means not writeable
  - Attempt to write to not writeable memory -> segmentation fault
- Writeable memory
  - At least two different pages are needed

Program Segments

- Segments (in executable file)
  - Text segment (W_R bit is cleared)
  - Data segment (W_R bit is set)
- Sections (in object file)
  - An executable file has segments, an object code file has sections
- Section definition directives
  - `section .text`
  - `section .data`

Program with section

- Program produce .o file with both text and data section

```
; Tiny program which increments a byte of stored data.
;
        global main
        section .text
main:   MOV AL,[XYZ]
        INC AL
        MOV [XYZ], AL
        RET
        section .data
XYZ:    db 3
```

- Segments in C code

```
char a = 'A';
main()
{   char *n;
    int i,t;

    n = &a;
    printf("Enter a pointer displacement in hex:");
    scanf("%x",&i);
    printf("Pointer value in hex = %x\n", (n + i));
    printf("Read, write, or skip?  (Enter 0, 1, or 2):");
    scanf("%d", &t);
    if (t == 0)
        printf("Contents there= %x", *(n + i));
    else if (t == 1)
        *(n + i) = 'Y';
    printf("\n");
}
```

- Calling C from Assembler

```
; This program calculates y = 2 * x + 1
        section .text
        global main
        extern printf
        extern scanf
main:   PUSH ABC
        CALL printf
        ADD ESP, 4
        PUSH XYZ
        PUSH BCD
        CALL scanf
        ADD ESP, 8
        MOV EAX, [XYZ]
        ADD EAX, EAX
        INC EAX
        PUSH EAX
        PUSH CDE
        CALL printf
        ADD ESP, 8
        RET
ABC:    db "Enter a number:  ",0AH,0
BCD:    db "%d \x0" ,0AH,0
CDE:    db "You get %d.\x0" ,0AH,0
        section .data
XYZ:    db 4 * 0
```

Other Data Segments

- Initialized data space

```
section .data
```

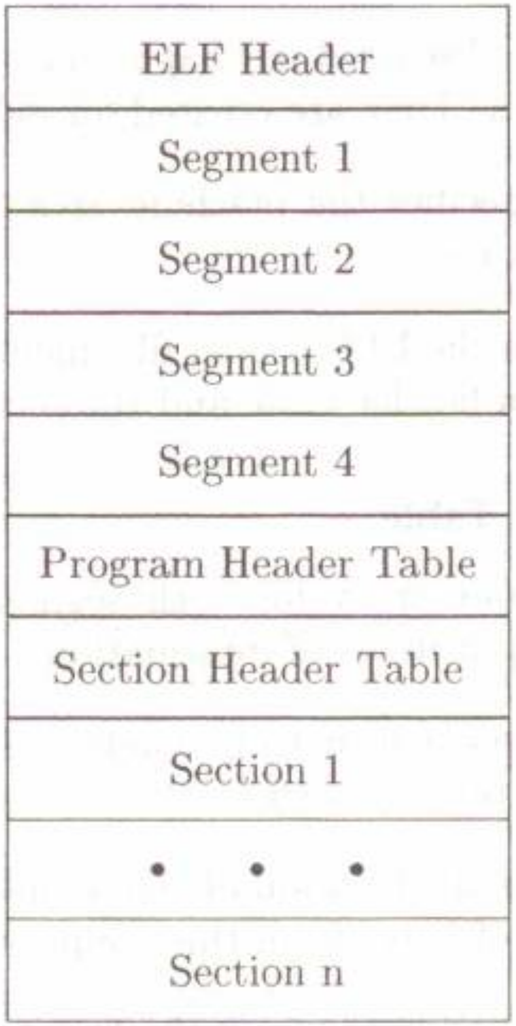
- Uninitialized data space

```
int x[10000];
```

- Dynamically allocated memory

```
malloc()
```

ELF Format



- Executable files in ELF format
- ELF header
  - Identifier string
  - File type
  - Machine architecture
  - Size, location, …
- Program header
  - Type
  - File offset
  - Virtual address
  - Physical address
  - File size
  - Memory size
  - Permissions
  - Alignment

ELF Format

- Program header table for `chmod`

```
0034:00d4 Program Header Table
     Seg File Virtual Physical File Mem       Align
Num Type Offst Address Address Size Size Perm -ment
00:  0006 0034 8048034 8048034 00a0 00a0 0005 0004
01:  0003 00d4 80480d4 80480d4 0013 0013 0004 0001
02:  0001 0000 8048000 8048000 1d29 1d29 0005 1000
03:  0001 1d30 804ad30 804ad30 011c 01fc 0006 1000
04:  0002 1dc4 804adc4 804adc4 0088 0088 0006 0004

Type 1 = loadable, 2 = dynamic info, 3 = interpreter
     4 = note, 6 = program header
```

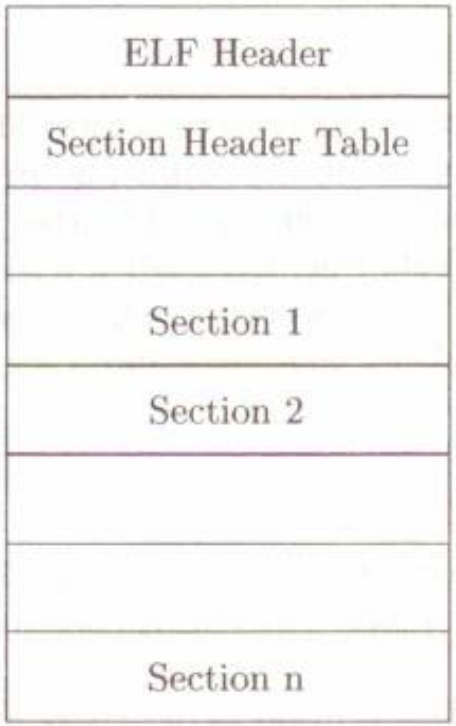
ELF Format

- ELF segments

```
int a = 0x55555555, b = 0x66666666;
int c = 0x77777777, d = 0x88888888;

main()
{

/* Store numbers which will be very easy to pick out
from the machine code of the compiled program.  */

    a = 0x11111111;
    b = 0x22222222;
    c = 0x33333333;
    d = 0x44444444;

    printf("Variables:  Addresses:\n")
    printf(" a %x\n", &a);
    printf(" b %x\n", &b);
    printf(" c %x\n", &c);
    printf(" d %x\n", &d);

/* %x means to use "hexadecimal" output.  */

    printf("\n");
}
```

Object Files in ELF Format

- ELF object file format
  - Section header table
  - Section



- Section header table

01bc:0374 Section Header Table

	Name	Type	Flgs	Virtual Address	Off-set	Size	Link	Info	Algn Mod	Entry Size
00:	0000	0000	0000	00000000	0000	0000	0000	0000	0000	0000
01:	001b	0001	0006	00000000	0034	0091	0000	0000	0004	0000
02:	0021	0009	0000	00000000	0490	00a0	0009	0001	0004	0008
03:	002b	0001	0003	00000000	00c8	0010	0000	0000	0004	0000
04:	0031	0008	0003	00000000	00d8	0000	0000	0000	0004	0000
05:	0036	0007	0000	00000000	00d8	0014	0000	0000	0001	0000
06:	003c	0001	0002	00000000	00ec	0070	0000	0000	0001	0000
07:	0044	0001	0000	00000000	015c	0012	0000	0000	0001	0000
08:	0011	0003	0000	00000000	016e	004d	0000	0000	0001	0000
09:	0001	0002	0000	00000000	0374	00f0	000a	0009	0004	0010
0a:	0009	0003	0000	00000000	0464	002b	0000	0000	0001	0000

Name: index into the Section Header String Table (.shstrab)
Type: 1 = Program Bits, 2 = Symbol Table, 3 = String Table
4 = Relocation Table, 5 = Hash Table, 6 = Dynamic Info
7 = Note, 8 = No Bits, 9 = Relocation Table