

Outline

- Before buildup your own OS, you need have a basic I/O system library
 - In this lab we use VGA screen as output and Keyboard as input interface.
 - Implement a simple printf() and getc() function.
- To do so you will need know the x86 system level and its I/O system behavior.

- X86 system registers and each relationship

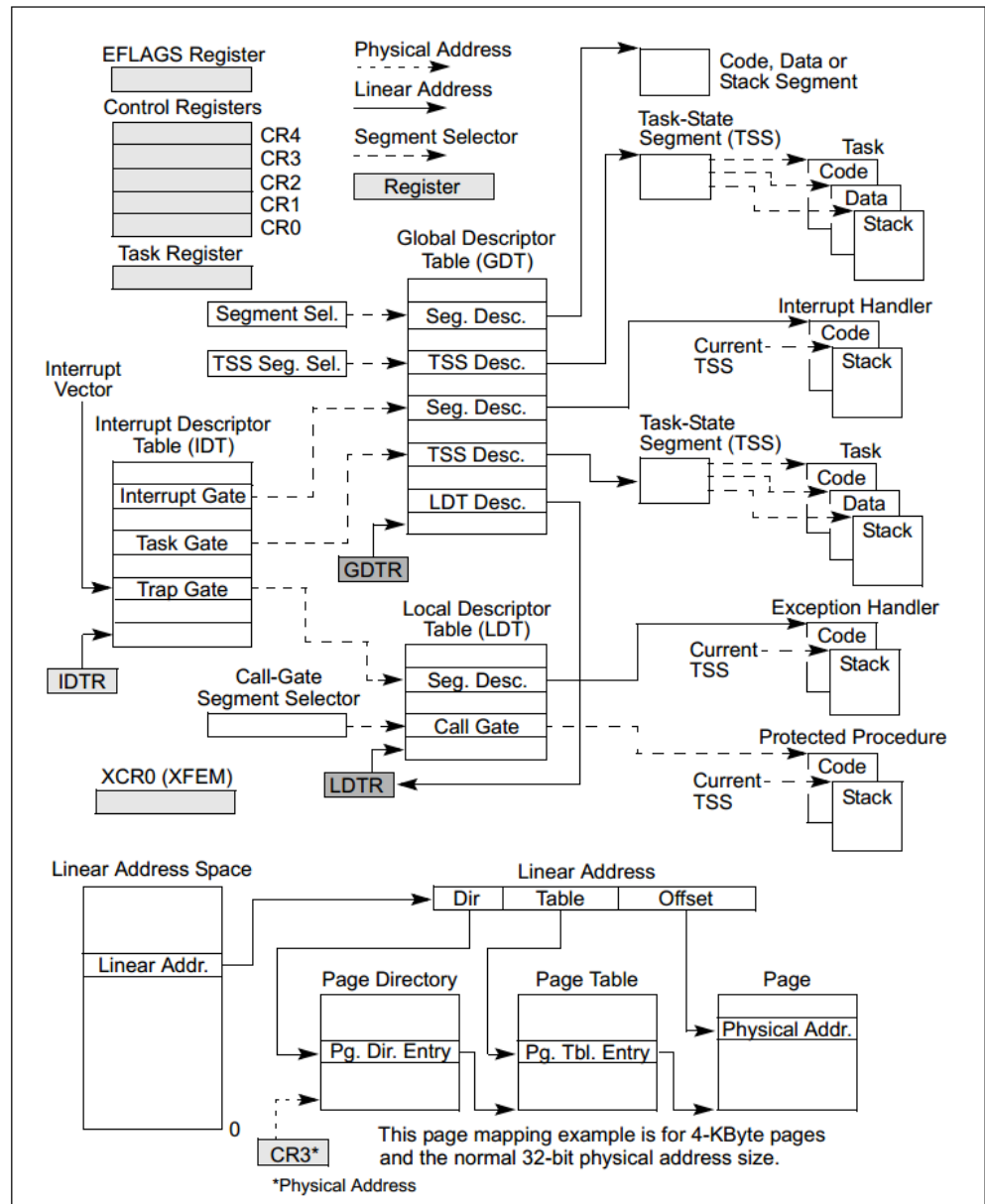


Figure 2-1. IA-32 System-Level Registers and Data Structures

X86 system registers

- Memory management registers: use for control task segment and interrupt
 - GDTR: Global Descriptor Table Register
 - LDTR: Local Descriptor Table Register
 - IDTR: Interrupt Descriptor Table Register
 - TR: Task Register
- Control registers
 - CR0, CR1, CR2, CR3

X86 Control Registers

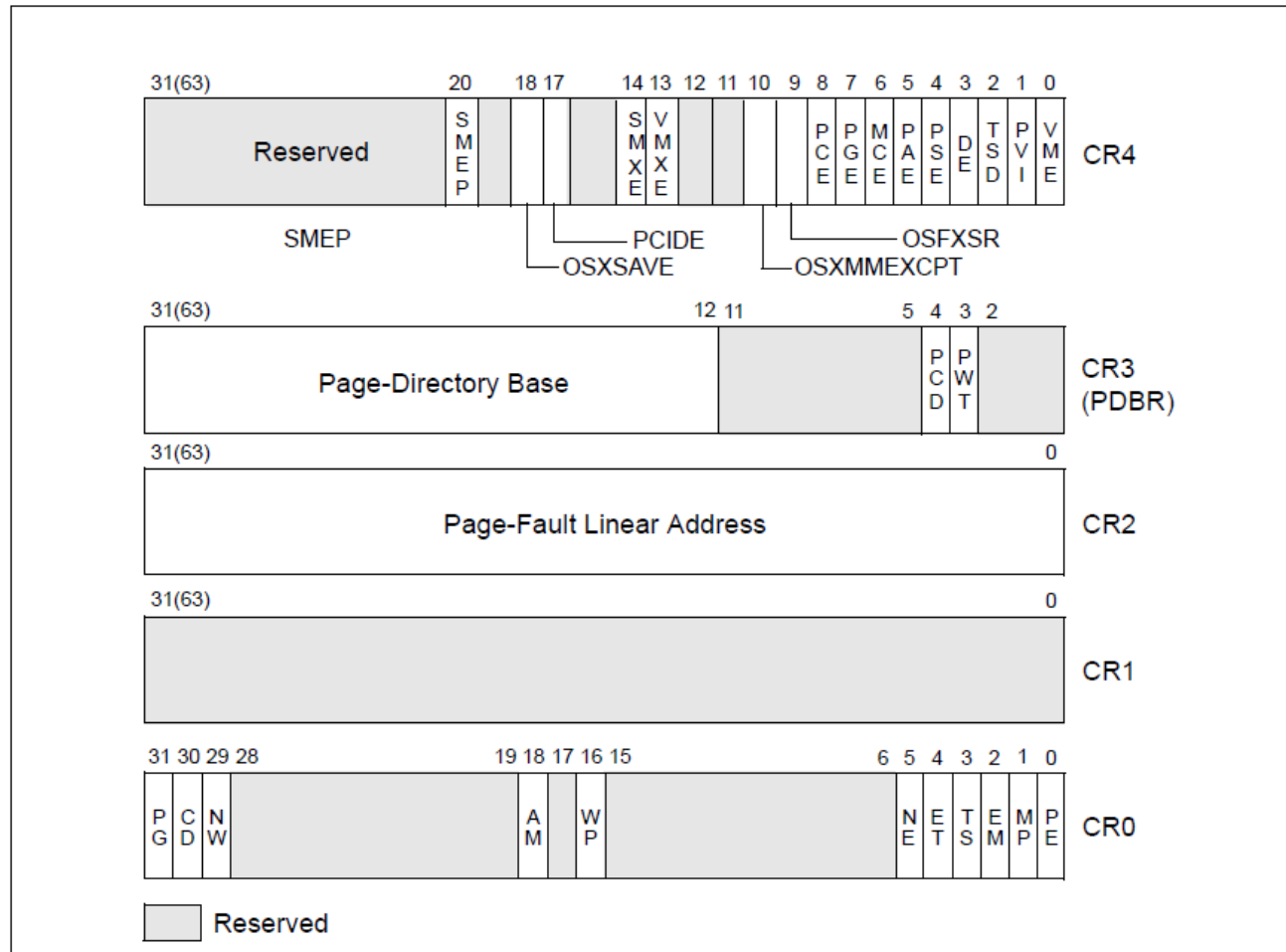


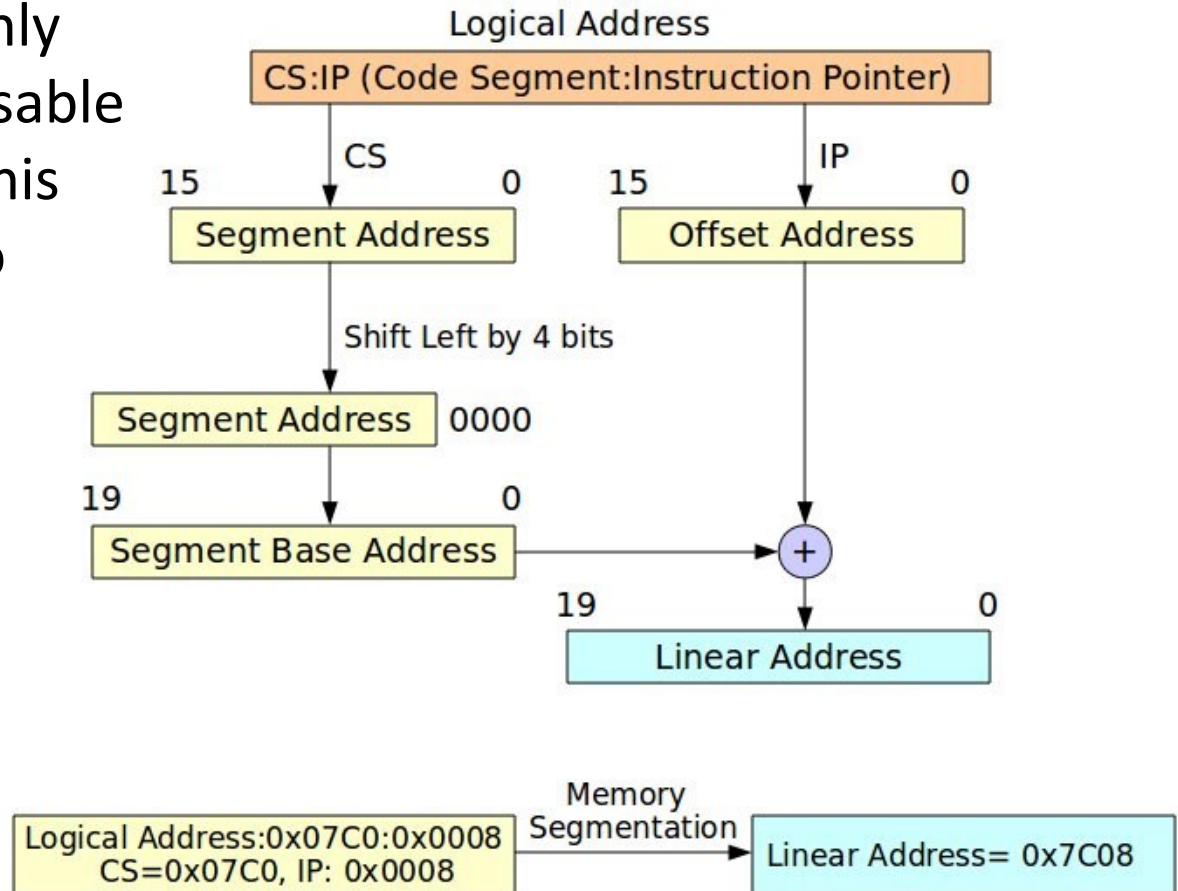
Figure 2-6. Control Registers

Segmented Addressing

- The processor uses byte addressing. The range of memory that can be addressed is called an **address space**.
- This is a form of addressing where a program may have many independent address spaces, called **segments**.
- The following notation is used to specify a byte address within a segment:
 - Segment-register:Byte-address
- Reference: <http://wiki.osdev.org/Segmentation>

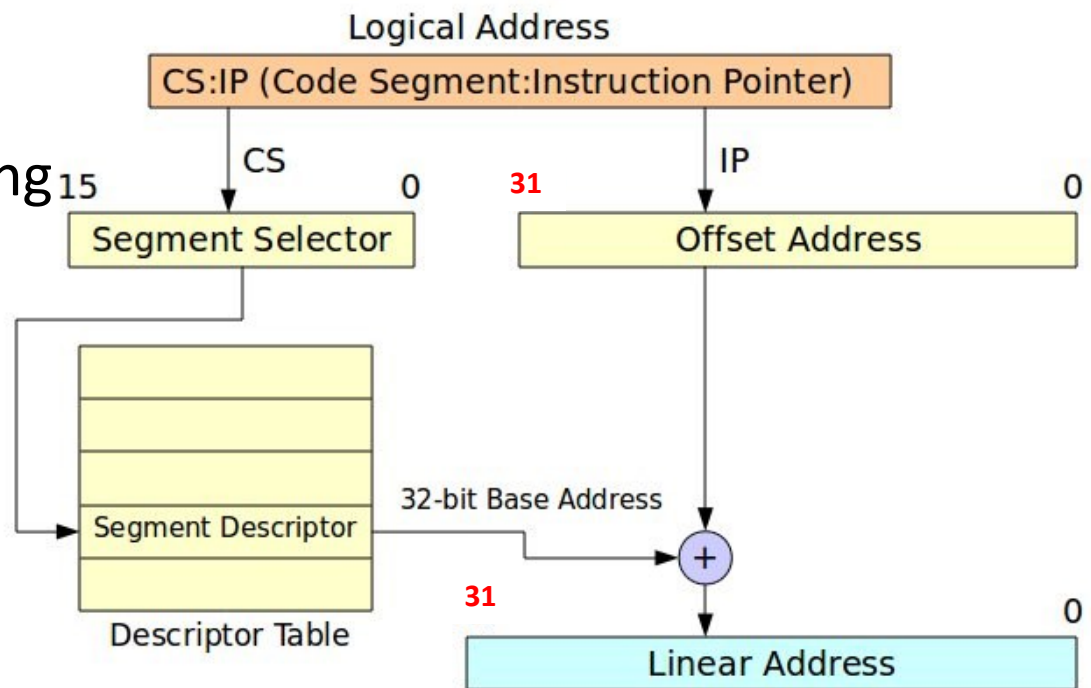
Real mode Addressing

- In x86 real mode only have 20bits addressable space 0~4KBytes, this addressing bus also called A20



Protected mode Addressing

- In protected mode each task have self own **32bits** address space
- Each logic address use descriptor table mapping to linear address



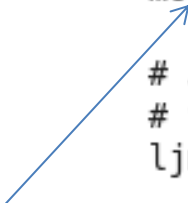
CPU Switch to Protected Mode

```
# Switch from real to protected mode, using a bootstrap GDT
# and segment translation that makes virtual addresses
# identical to their physical addresses, so that the
# effective memory map does not change during the switch.
```

```
lgdt    gdt_desc
movl    %cr0, %eax
orl     $CR0_PE_ON, %eax
movl    %eax, %cr0
```

```
# Jump to next instruction, but in 32-bit code segment.
# Switches processor into 32-bit mode.
ljmp    $PROT_MODE_CSEG, $protcseg
```

Setting the CR0's PE bit.
It means next
instruction is "Protected
mode" enable



Segment Descriptor

- In X86 protected mode “Segment Descriptor” is use for record the segment ***type, base address, limit,...***
- It may store in global descriptor table(GDT), local descriptor table(LDT) or interrupt descriptor table(IDT)

```
// Segment Descriptors
struct Segdesc {
    unsigned sd_lim_15_0 : 16; // Low bits of segment limit
    unsigned sd_base_15_0 : 16; // Low bits of segment base address
    unsigned sd_base_23_16 : 8; // Middle bits of segment base address
    unsigned sd_type : 4; // Segment type (see STS_constants)
    unsigned sd_s : 1; // 0 = system, 1 = application
    unsigned sd_dpl : 2; // Descriptor Privilege Level
    unsigned sd_p : 1; // Present
    unsigned sd_lim_19_16 : 4; // High bits of segment limit
    unsigned sd_avl : 1; // Unused (available for software use)
    unsigned sd_rsv1 : 1; // Reserved
    unsigned sd_db : 1; // 0 = 16-bit segment, 1 = 32-bit segment
    unsigned sd_g : 1; // Granularity: limit scaled by 4K when set
    unsigned sd_base_31_24 : 8; // High bits of segment base address
};
```

Segment Types

Table 3-2. System-Segment and Gate-Descriptor Types

| Type Field | | | | | Description | |
|------------|----|----|---|---|------------------------|---------------------------------------|
| Decimal | 11 | 10 | 9 | 8 | 32-Bit Mode | IA-32e Mode |
| 0 | 0 | 0 | 0 | 0 | Reserved | Upper 8 byte of an 16-byte descriptor |
| 1 | 0 | 0 | 0 | 1 | 16-bit TSS (Available) | Reserved |
| 2 | 0 | 0 | 1 | 0 | LDT | LDT |
| 3 | 0 | 0 | 1 | 1 | 16-bit TSS (Busy) | Reserved |
| 4 | 0 | 1 | 0 | 0 | 16-bit Call Gate | Reserved |
| 5 | 0 | 1 | 0 | 1 | Task Gate | Reserved |
| 6 | 0 | 1 | 1 | 0 | 16-bit Interrupt Gate | Reserved |
| 7 | 0 | 1 | 1 | 1 | 16-bit Trap Gate | Reserved |
| 8 | 1 | 0 | 0 | 0 | Reserved | Reserved |
| 9 | 1 | 0 | 0 | 1 | 32-bit TSS (Available) | 64-bit TSS (Available) |
| 10 | 1 | 0 | 1 | 0 | Reserved | Reserved |
| 11 | 1 | 0 | 1 | 1 | 32-bit TSS (Busy) | 64-bit TSS (Busy) |
| 12 | 1 | 1 | 0 | 0 | 32-bit Call Gate | 64-bit Call Gate |
| 13 | 1 | 1 | 0 | 1 | Reserved | Reserved |
| 14 | 1 | 1 | 1 | 0 | 32-bit Interrupt Gate | 64-bit Interrupt Gate |
| 15 | 1 | 1 | 1 | 1 | 32-bit Trap Gate | 64-bit Trap Gate |

GDT

- Use for place **kernel space** segment descriptors

| | | | | |
|---------------------------------------|------|-------|---------|------------------------------------|
| | gdt: | .word | 0,0,0,0 | ! dummy |
| Global Code Segment Descriptor | { | .word | 0x07FF | ! 8Mb - limit=2047 (2048*4096=8Mb) |
| | | .word | 0x0000 | ! base address=0x000000 |
| | | .word | 0x9A00 | ! code read/exec |
| | | .word | 0x00C0 | ! granularity=4096, 386 |
| Global Data Segment Descriptor | { | .word | 0x07FF | ! 8Mb - limit=2047 (2048*4096=8Mb) |
| | | .word | 0x0000 | ! base address=0x000000 |
| | | .word | 0x9200 | ! data read/write |
| | | .word | 0x00C0 | ! granularity=4096, 386 |

LDT

- Use for place **process/task** local segment descriptors

Task **Code** Segment Descriptor

```
ldt0: .quad 0x0000000000000000
      .quad 0x00c0fa00000003ff
      .quad 0x00c0f200000003ff
```

Task **data** Segment Descriptor

```
# 0x0f, base = 0x00000
# 0x17
```

Why 0x17?

Load GDT and LDT

- Load GDT use linear address, and load LDT via LDT descriptor

```
setup_gdt:
    lgdt lgdt_opcode
    ret
lgdt_opcode:
    .word (end_gdt-gdt)-1 # so does gdt
    .long gdt              # This will be rewrite by code.
```

```
movl $TSS0_SEL, %eax
ltr %ax
movl $LDT0_SEL, %eax
lldt %ax
```

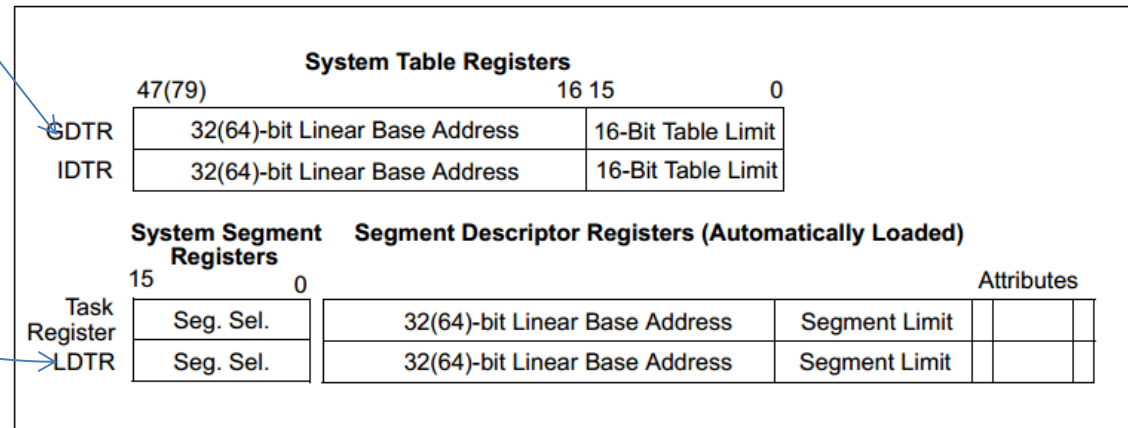


Figure 2-5. Memory Management Registers

Segment Selector

- What is the segment selector?
- Ans: Use for point a segment descriptor

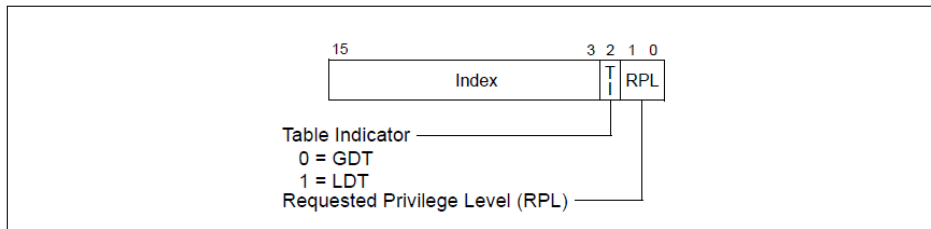


Figure 3-6. Segment Selector

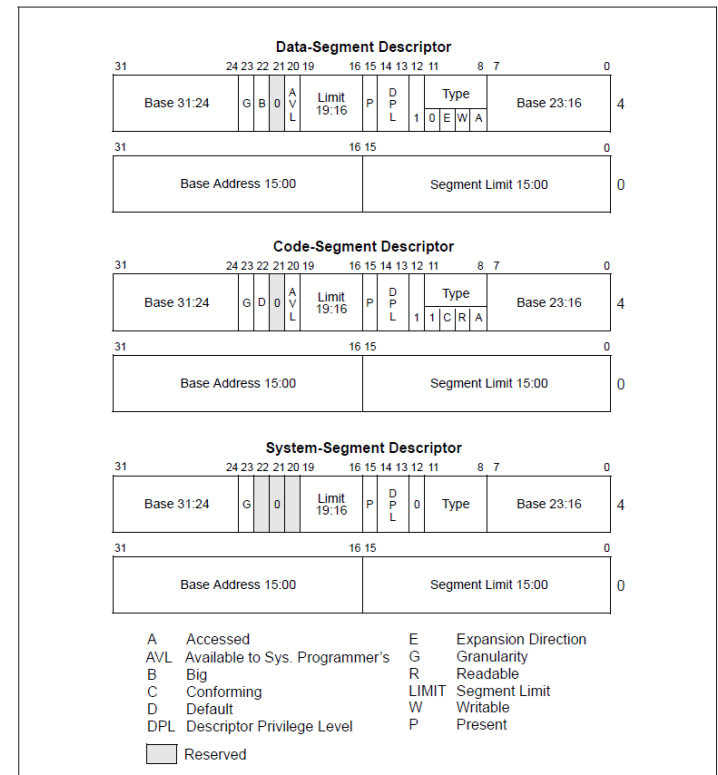


Figure 5-1. Descriptor Fields Used for Protection

X86 Protection Ring

- Why need protection ring?
 - Some instruction do not use in user/application mode, such like reload interrupt table, setup page table,...,etc.
- Linux only use level 0 and level 3 as kernel and user mode.

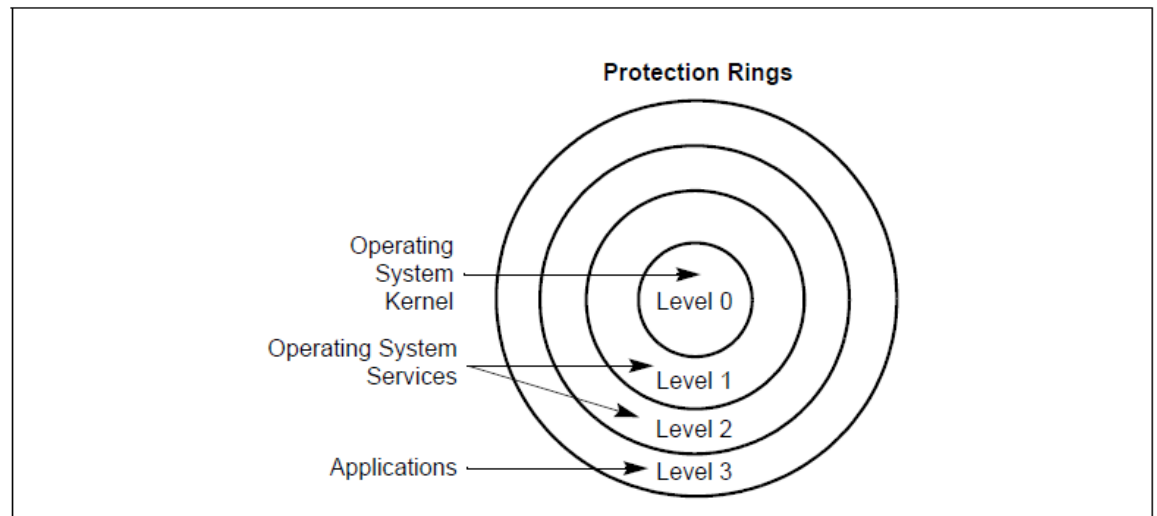


Figure 5-3. Protection Rings

Interrupt Setup

- In x86 system use the interrupt description table (IDT) to store each interrupt gate
 - Use “lidt” instruction load the IDT base address to IDTR register
- For example (In Linux)
 - 0x80 is system call
 - 0x08 is timer interrupt

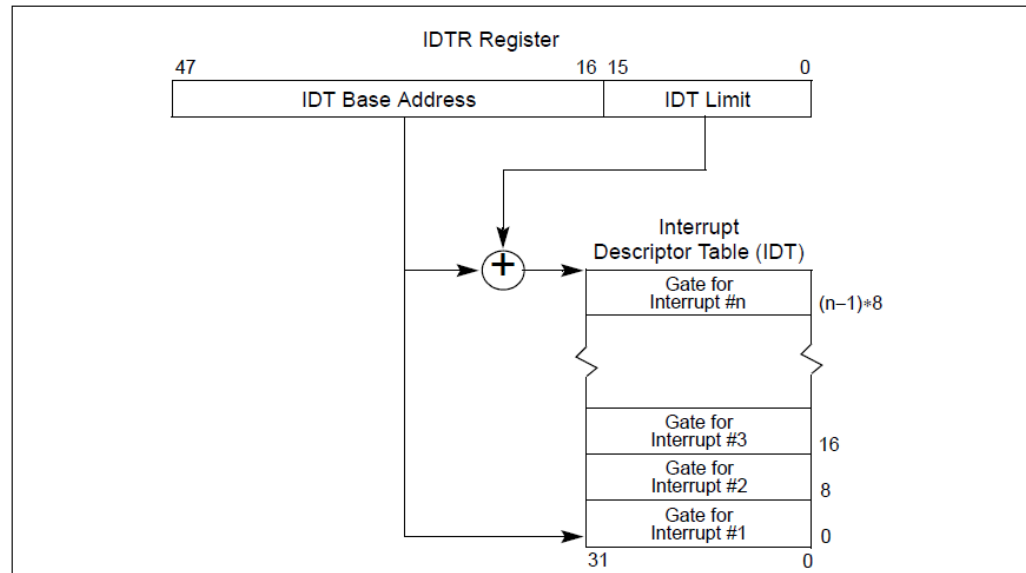


Figure 6-1. Relationship of the IDTR and IDT

Type of interrupts in protected mode

Table 6-1. Protected-Mode Exceptions and Interrupts

| Vector No. | Mne-monic | Description | Type | Error Code | Source |
|------------|-----------|--|------------|------------|---|
| 0 | #DE | Divide Error | Fault | No | DIV and IDIV instructions. |
| 1 | #DB | RESERVED | Fault/Trap | No | For Intel use only. |
| 2 | — | NMI Interrupt | Interrupt | No | Nonmaskable external interrupt. |
| 3 | #BP | Breakpoint | Trap | No | INT 3 instruction. |
| 4 | #OF | Overflow | Trap | No | INTO instruction. |
| 5 | #BR | BOUND Range Exceeded | Fault | No | BOUND instruction. |
| 6 | #UD | Invalid Opcode (Undefined Opcode) | Fault | No | UD2 instruction or reserved opcode. ¹ |
| 7 | #NM | Device Not Available (No Math Coprocessor) | Fault | No | Floating-point or WAIT/FWAIT instruction. |
| 8 | #DF | Double Fault | Abort | Yes (zero) | Any instruction that can generate an exception, an NMI, or an INTR. |
| 9 | | Coprocessor Segment Overrun (reserved) | Fault | No | Floating-point instruction. ² |
| 10 | #TS | Invalid TSS | Fault | Yes | Task switch or TSS access. |
| 11 | #NP | Segment Not Present | Fault | Yes | Loading segment registers or accessing system segments. |
| 12 | #SS | Stack-Segment Fault | Fault | Yes | Stack operations and SS register loads. |
| 13 | #GP | General Protection | Fault | Yes | Any memory reference and other protection checks. |
| 14 | #PF | Page Fault | Fault | Yes | Any memory reference. |
| 15 | — | (Intel reserved. Do not use.) | | No | |
| 16 | #MF | x87 FPU Floating-Point Error (Math Fault) | Fault | No | x87 FPU floating-point or WAIT/FWAIT instruction. |

Table 6-1. Protected-Mode Exceptions and Interrupts (Contd.)

| | | | | | |
|--------|-----|--|-----------|------------|---|
| 17 | #AC | Alignment Check | Fault | Yes (Zero) | Any data reference in memory. ³ |
| 18 | #MC | Machine Check | Abort | No | Error codes (if any) and source are model dependent. ⁴ |
| 19 | #XM | SIMD Floating-Point Exception | Fault | No | SSE/SSE2/SSE3 floating-point instructions ⁵ |
| 20-31 | — | Intel reserved. Do not use. | | | |
| 32-255 | — | User Defined (Non-reserved) Interrupts | Interrupt | | External interrupt or INT <i>n</i> instruction. |

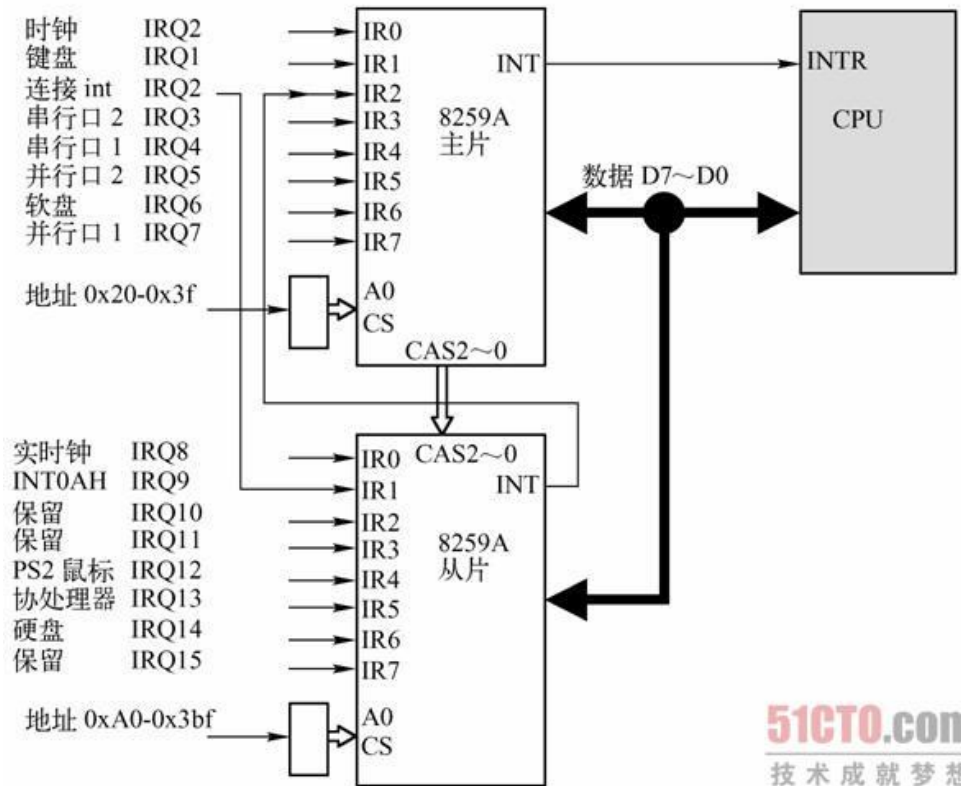
NOTES:

1. The UD2 instruction was introduced in the Pentium Pro processor.
2. Processors after the Intel386 processor do not generate this exception.
3. This exception was introduced in the Intel486 processor.
4. This exception was introduced in the Pentium processor and enhanced in the P6 family processors.
5. This exception was introduced in the Pentium III processor.

- OS can use these vectors to design its own traps
- In our lab, we define the vector number in inc/trap.h

Programmable Interrupt Controller

- In tradition x86 PC system use 8259A chip as programmable interrupt controller (PIC)
- It allows CPU to handle more then 8 device interrupt but only use single INTR line from CPU.
- Relate controller code is at kernel/picirq.c



VGA

- How to control VGA
 - http://wiki.osdev.org/VGA_Hardware
- In this lab we use the video buffer to print our message on the screen.
 - http://wiki.osdev.org/Printing_To_Screen
 - The video memory base is 0xB8000
- We already implemented a simple video driver, you can trace code at file kernel/kbd.c

Keyboard

- In this lab we implemented a simple driver at kernel/kbd.c
- A simple keyboard scanner driver
 - <http://www.osdever.net/bkerndev/Docs/keyboard.htm>

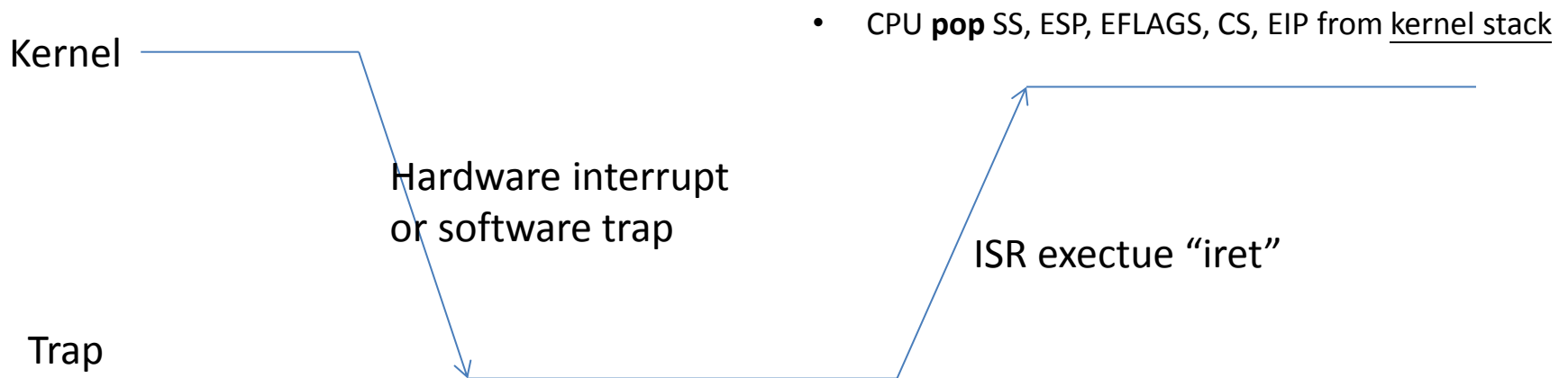
System prepare

Main.c

- Lab3:
 - System initial
 - Video
 - PIC
 - Trap and interrupt (IDT initial)
 - Keyboard
 - Timer
- Lab4:
 - User level task initial
 - Setup TSS, GDT
 - Load GDT, LDT and TSS
 - Privilege level switch to user mode
 - Privilege level switch to user mode
 - Fork system call
 - Context switch

Interrupt/Exception behavior

- External interrupt generated from Programmable Interrupt Controller (PIC or APIC)
- Software interrupt could generated from user trap (e.g. int xx) or exceptions (e.g. stack fault, divide zero...)



- CPU found the trap gate in IDT
- CPU check the Privilege level (DPL)
- CPU **push** EIP, CS, EFLAGS, ESP, SS to kernel stack
- CPU load the code segment for IDT
- CPU jump trap entry (use trap gate)

Reference

- IO memory map:
 - <http://www.pcguide.com/ref/mbsys/res/ioSummary-c.html>
- x86 instruction set:
 - <http://x86.renejeschke.de/>
- IO address:
 - <http://www.pcguide.com/ref/mbsys/res/ioSummary-c.html>
- video and system BIOS area:
 - <http://www.pcguide.com/ref/ram/umaMap-c.html>
- PIC reference:
 - <http://www.brokenthorn.com/Resources/OSDevPic.html>
 - http://wiki.osdev.org/8259_PIC
- Linker script
 - <https://sourceware.org/binutils/docs/ld/Scripts.html>
 - <http://www.osdever.net/bkerndev/Docs/basickernel.htm>
- Trap and Interrupt
 - <http://www.osdever.net/bkerndev/Docs/isrs.htm>
 - <http://pdos.csail.mit.edu/6.828/2012/labs/lab3/>