Writing the operating system from scratch

Questions 1: how to boot?

What we Know:

BIOS loads first sector from boot device into memory 0x7c00, then jump to it.

Solutions:

Write a boot program into a boot device.

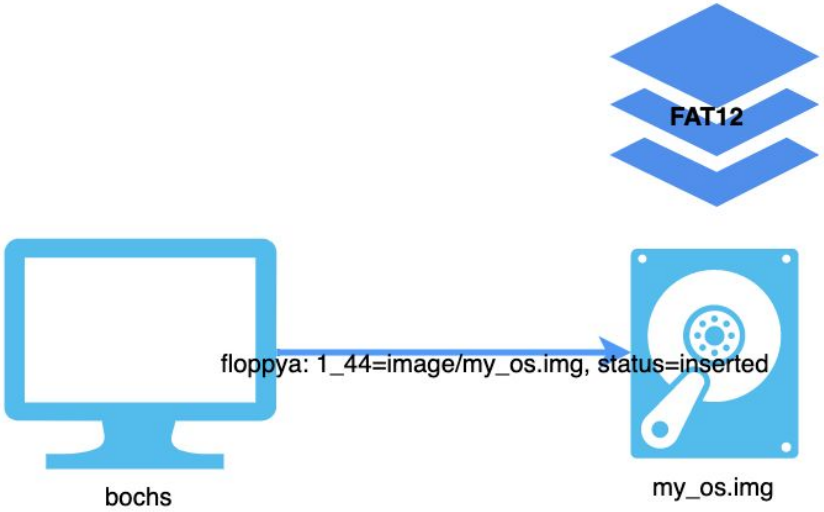
Choose a floppy disk as a boot device.

Boot program size limited by 512 bytes, so make a filesystem on floppy disk, write a loader program into it.

Boot program loads this loader program into memory, then jump to it.

Loader program is designed to implement more complex functions

Environment



Install OS

```
dd if=build/boot.bin of=image/my_os.img bs=512 count=1 conv=notrunc
sudo mount image/my_os.img /mnt/floppy
sudo cp build/loader.bin build/kernel.bin /mnt/floppy
sleep 1 sudo
umount /mnt/floppy
```

Make a floppy disk image

```
bximage
mkfs.msdos my_os.img
```

Boot

Read a sector from Root Directory

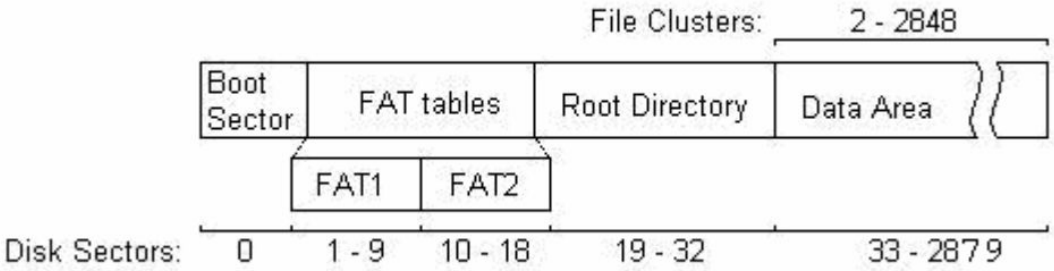
For each directory entry(32 Bytes)

Find name “loader.bin”

Get First Logical Cluster No.

Read sector from Data Area into memory 0x90000

JMP 0x9000:0x100



each entry(12 bits) in FAT tables pointer to a cluster/sector on Data Area

Specifically, the FAT entry values signify the following:

Value	Meaning
0x00	Unused
0xFF0-0xFF6	Reserved cluster
0xFF7	Bad cluster
0xFF8-0xFFF	Last cluster in a file
(anything else)	Number of the next cluster in the file

Questions 2: how to manage memory?

What we Know:

Real mode is characterized by a 20-bit segmented memory address space (giving exactly 1 MiB of addressable memory) and unlimited direct software access to all addressable memory, I/O addresses and peripheral hardware.

Solutions:

Let’s talk about Protected Mode

Segmentation

Segmentation provides a mechanism for dividing the processor's addressable memory space (called the linear address space) into smaller protected address spaces called segments.

Segments can be used to hold the code, data, and stack for a program.

To locate a byte in a particular segment, a logical address (also called a far pointer) must be provided. A logical address consists of a segment selector and an offset

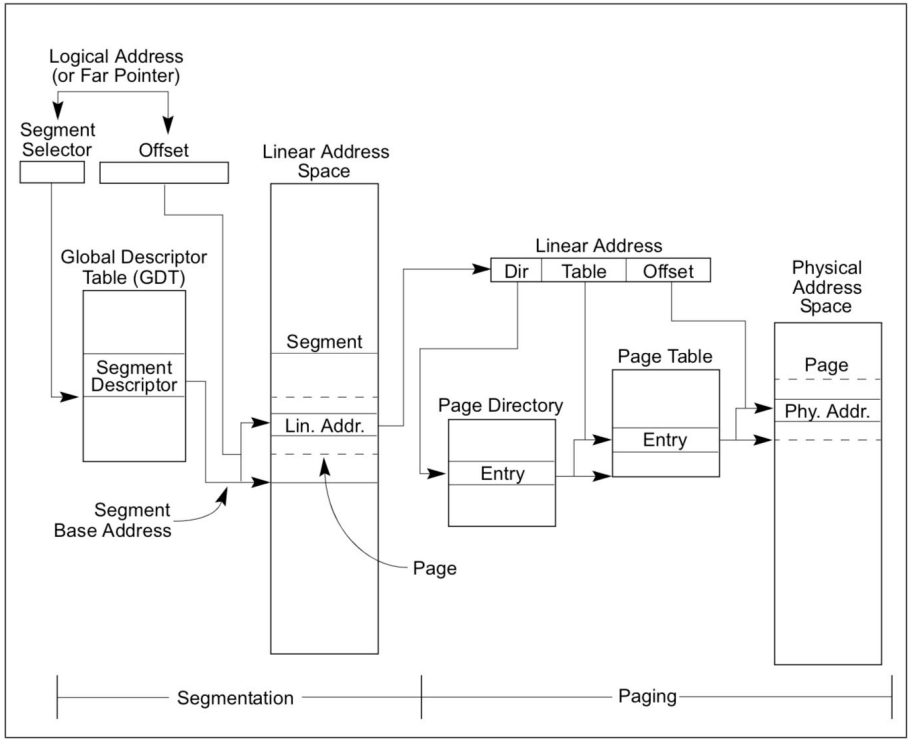


Figure 3-1. Segmentation and Paging

Segment Selector
Segment Descriptor

A segment selector is a 16-bit identifier for a segment (see Figure 3-6). It does not point directly to the segment, but instead points to the segment descriptor that defines the segment.

A segment descriptor is a data structure in a GDT or LDT that provides the processor with the size and location of a segment, as well as access control and status information.

Segment descriptors are typically created by compilers, linkers, loaders, or the operating system or executive, but not application programs.

```
%macro Descriptor 3
    dw  %2 & 0FFFFh
    dw  %1 & 0FFFFh
    db  (%1 >> 16) & 0FFh
    dw  ((%2 >> 8) & 0F00h) | (%3 & 0F0FFh)
    db  (%1 >> 24) & 0FFh
%endmacro
```

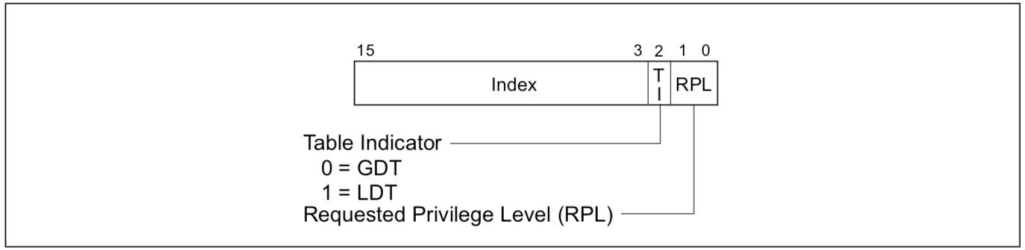


Figure 3-6. Segment Selector

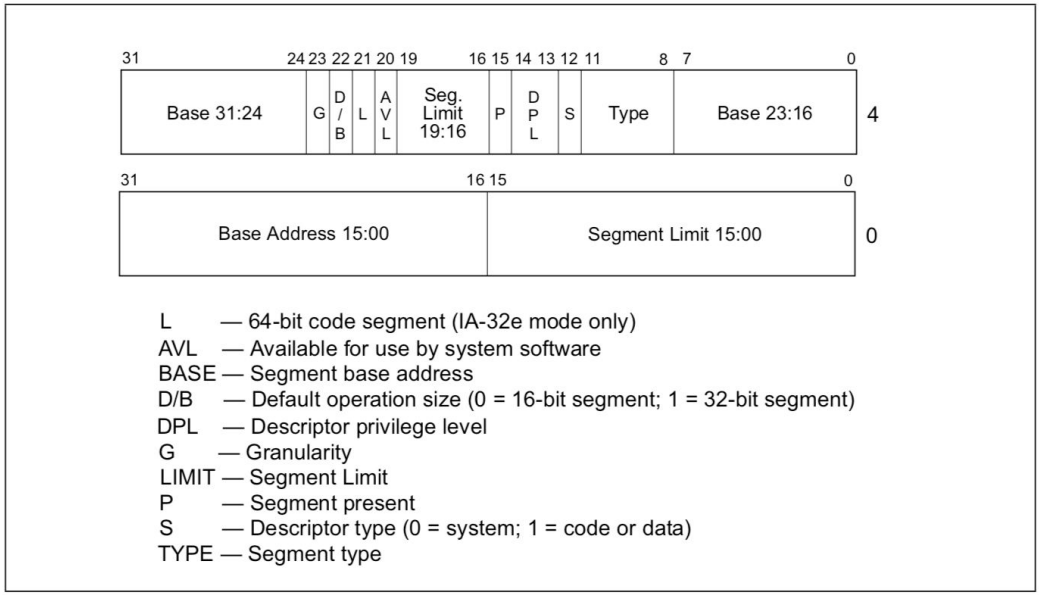


Figure 3-8. Segment Descriptor

GDT & LDT

The GDT is not a segment itself; instead, it is a data structure in linear address space. The base linear address and limit of the GDT must be loaded into the GDTR register.

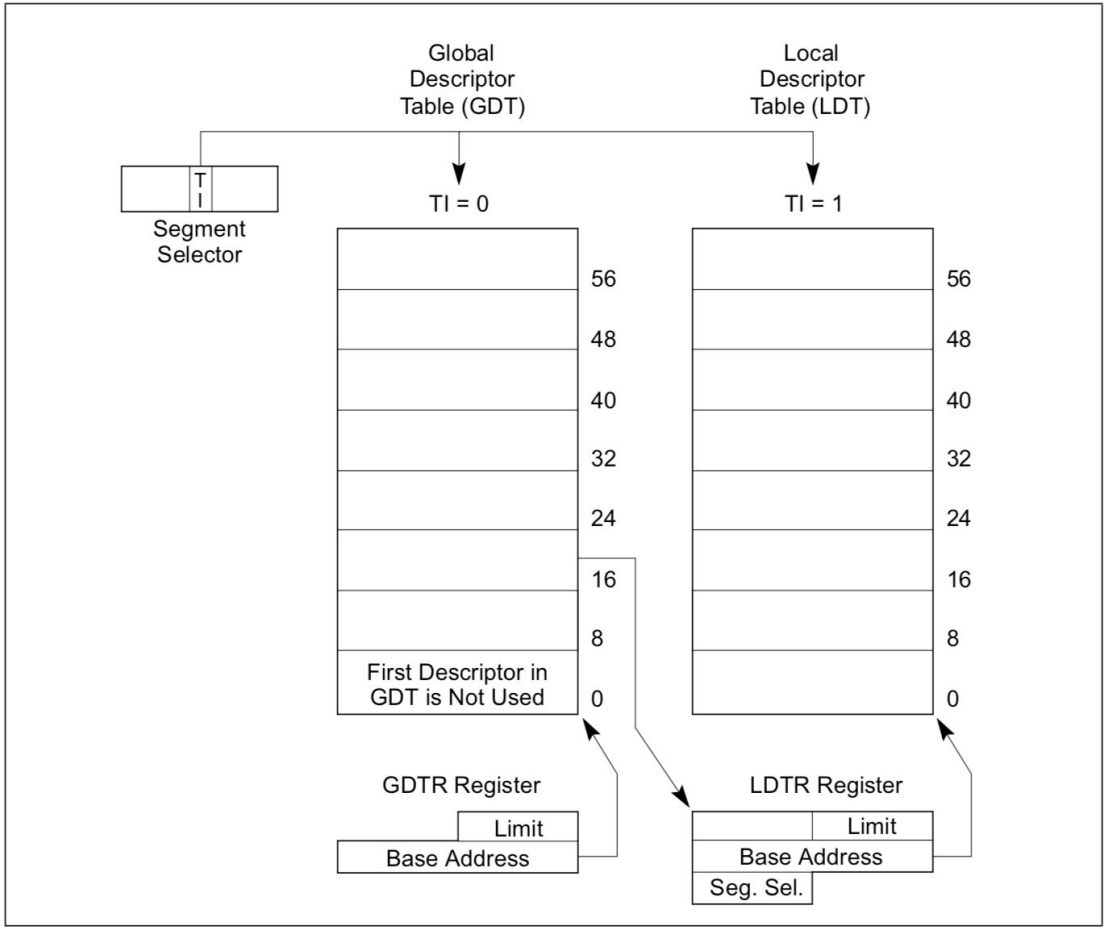


Figure 3-10. Global and Local Descriptor Tables

Codes

Prepare GDT

Set cr0 PE flag

Initialize segment register

```
; GDT
LABEL_GDT:       Descriptor 0, 0, 0
LABEL_DESC_FLAT_C:    Descriptor 0, 0fffffh, DA_CR|DA_32|DA_LIMIT_4K; 0-4G
LABEL_DESC_FLAT_RW:   Descriptor 0, 0fffffh, DA_DRW|DA_32|DA_LIMIT_4K; 0-4G
LABEL_DESC_VIDEO:     Descriptor 0b8000h, 0ffffh, DA_DRW|DA_DPL3;

GdtLen equ $ - LABEL_GDT
GdtPtr dw GdtLen - 1; boundary
    dd BaseOfLoader*10h + LABEL_GDT
; Selector
SelectorFlatC equ LABEL_DESC_FLAT_C - LABEL_GDT
SelectorFlatRW equ LABEL_DESC_FLAT_RW - LABEL_GDT
SelectorVideo equ LABEL_DESC_VIDEO - LABEL_GDT + SA_RPL3
```

```
    lgdt [GdtPtr]
    cli

    in al, 92h
    or al, 00000010b
    out 92h, al

    mov eax, cr0
    or eax, 1
    mov cr0, eax

    jmp dword SelectorFlatC:(BaseOfLoaderPhyAddr+LABEL_PM_START)
```

```
LABEL_PM_START:
    mov ax, SelectorVideo
    mov gs, ax

    mov ax, SelectorFlatRW
    mov ds, ax
    mov es, ax
    mov fs, ax
    mov ss, ax
```

Paging

Paging provides a mech- anism for implementing a conventional demand-paged, virtual-memory system where sections of a program's execution environment are mapped into physical memory as needed.

Paging is enabled if CR0.PG = 1. Paging can be enabled only if protection is enabled (CR0.PE = 1).

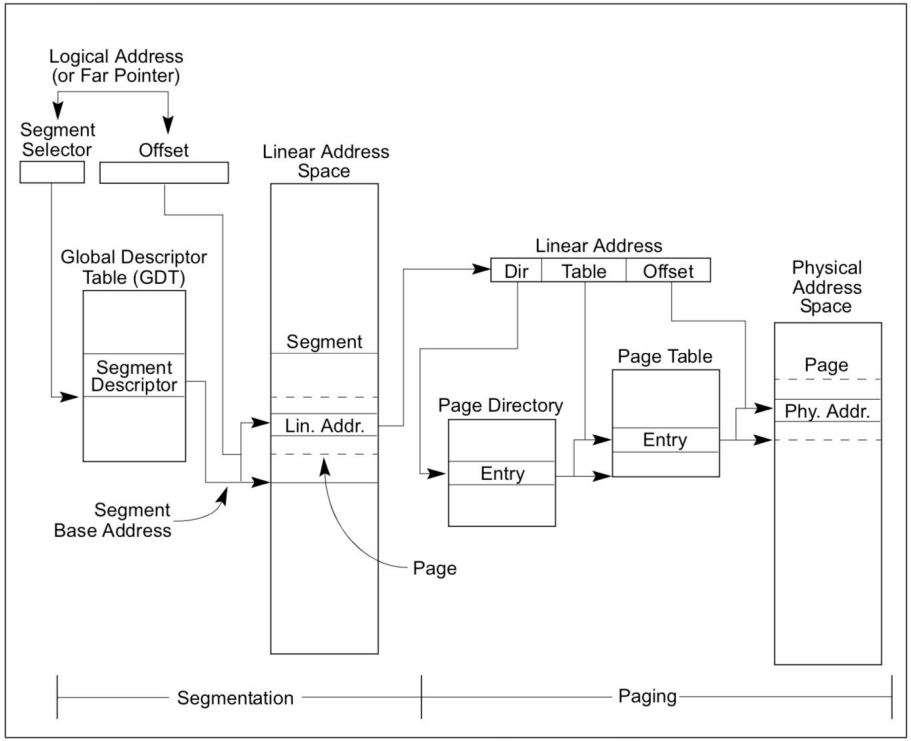


Figure 3-1. Segmentation and Paging

```
;-------------------------
; Function SetupPaging
SetupPaging:
    xor edx, edx
    mov eax, [dwMemSize]
    mov ebx, 400000h
    div ebx
    mov ecx, eax
    test edx, edx
    jz .no_remainder
    inc ecx
.no_remainder:
    push ecx


    ; init page dir table
    mov ax, SelectorFlatRW
    mov es, ax
    mov edi, PageDirBase
    xor eax, eax
    mov eax, PageTblBase | PG_P | PG_USU | PG_RWW
.1:
    stosd
    add eax, 4096
    loop .1

    ;init page table
    pop eax
    mov ebx, 1024
    mul ebx
    mov ecx, eax
    mov edi, PageTblBase
    xor eax, eax
    mov eax, PG_P | PG_USU | PG_RWW
.2:
    stosd
    add eax, 4096 ; page manage 4K space
    loop .2

    mov eax, PageDirBase
    mov cr3, eax
    mov eax, cr0
    or eax, 80000000h
    mov cr0, eax
    jmp short .3
.3:
    nop

    ret
```

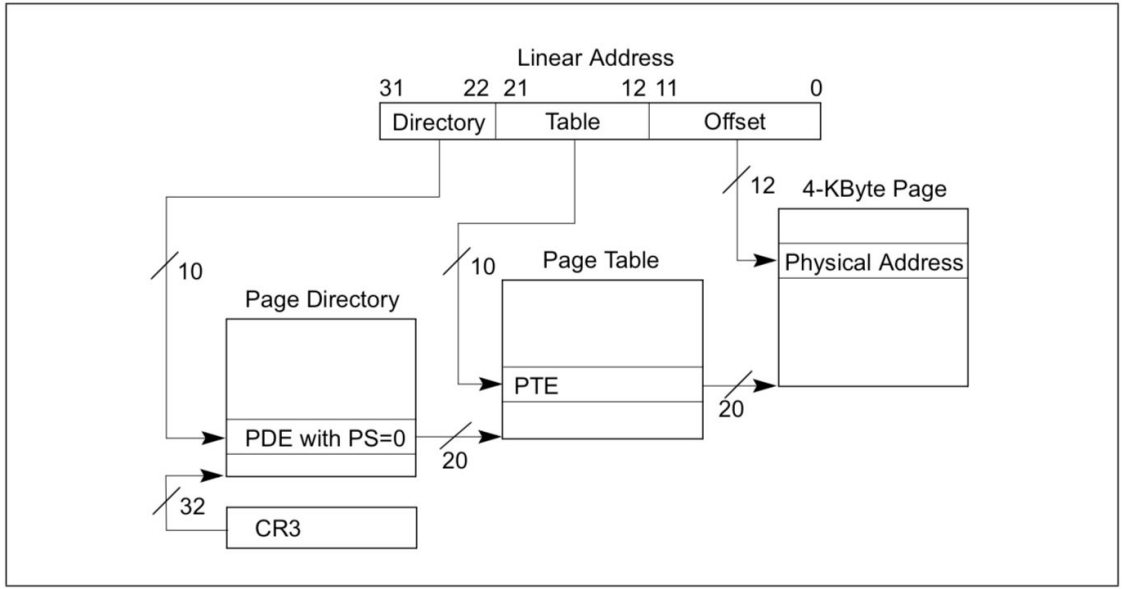


Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging

Questions 3: how to design User mode and Kernel mode?

What we Know：

OS like unix usually forbid user accessing hardware directly.

The processor uses privilege levels to prevent a program or task operating at a lesser privilege level from accessing a segment with a greater privilege.

Solutions:

User mode run privilege level 1, implement Process

Kernel mode run privilege level 0, implement Interrupt handle, such as clock to schedule Process

Privilege Level

The processor's segment-protection mechanism recognizes 4 privilege levels, numbered from 0 to 3. The greater numbers mean lesser privileges.

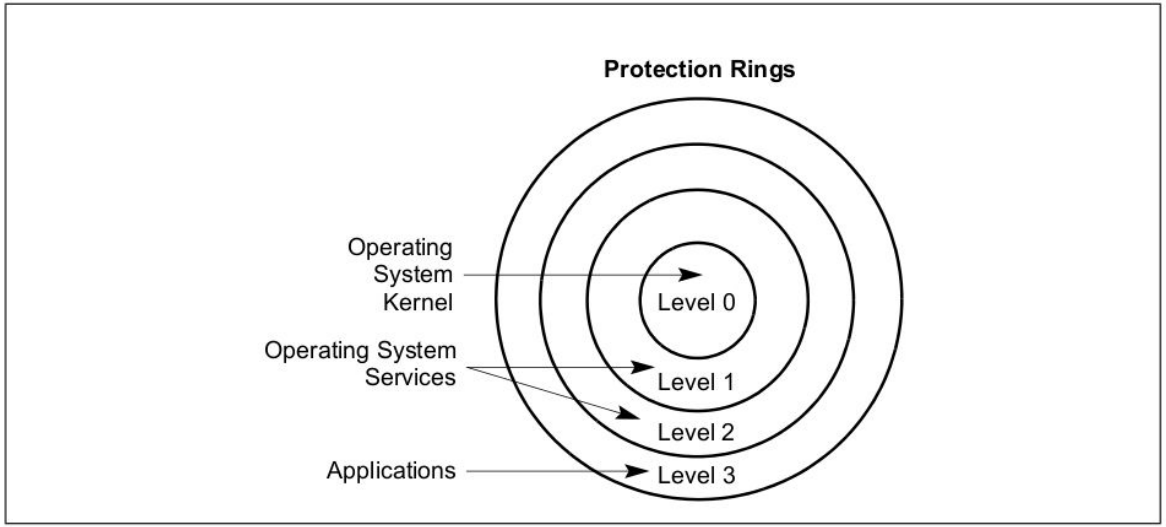


Figure 5-3. Protection Rings

Accessing Data Segments

The CPL is the privilege level of the currently executing program or task. It is stored in bits 0 and 1 of the CS and SS segment registers

The DPL is the privilege level of a segment or gate. It is stored in the DPL field of the segment or gate descriptor for the segment or gate.

The RPL is an override privilege level that is assigned to segment selectors. It is stored in bits 0 and 1 of the segment selector.

Max(CPL, RPL) <= DPL

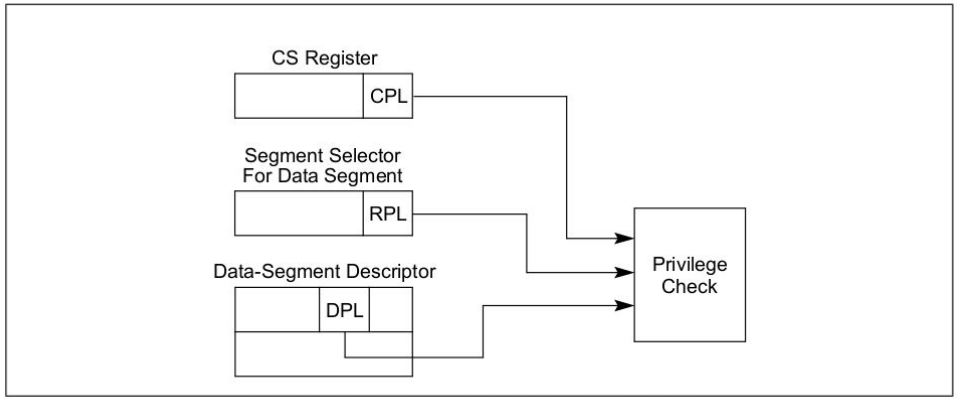


Figure 5-4. Privilege Check for Data Access

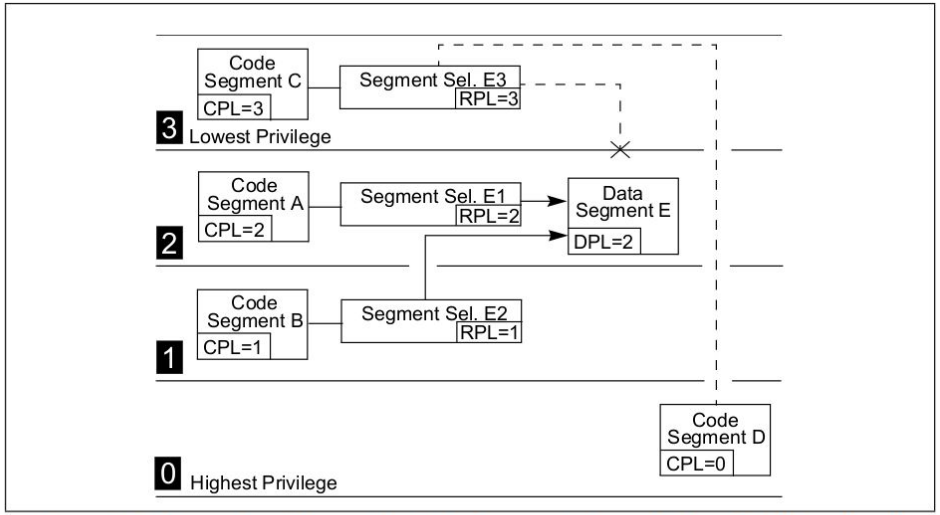


Figure 5-5. Examples of Accessing Data Segments From Various Privilege Levels

Accessing Code Segments

Most code segments are nonconforming
For these segments, program control car
transferred only to code segments at the
same level of privilege, unless the transfe
carried out through a call gate, as describ
in the following sections.

Nonconforming: CPL = DPL and RPL ≤ DPL

Conforming: CPL ≥ DPL, CPL not change

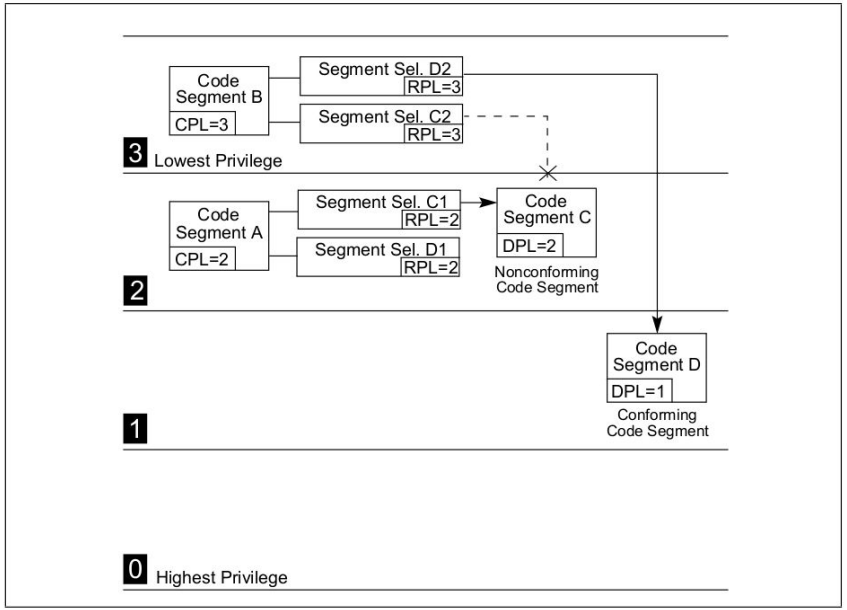


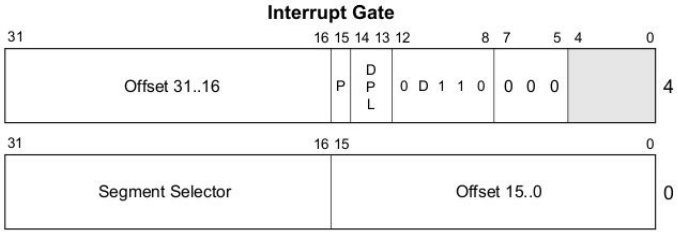
Figure 5-7. Examples of Accessing Conforming and Nonconforming Code Segments From Various Privilege Levels

Gate Descriptors

To provide controlled access to code segments with different privilege levels, the processor provides special set of descriptors called gate descriptors.

There are four kinds of gate descriptors:

- Call gates
- Trap gates
- Interrupt gates
- Task gates



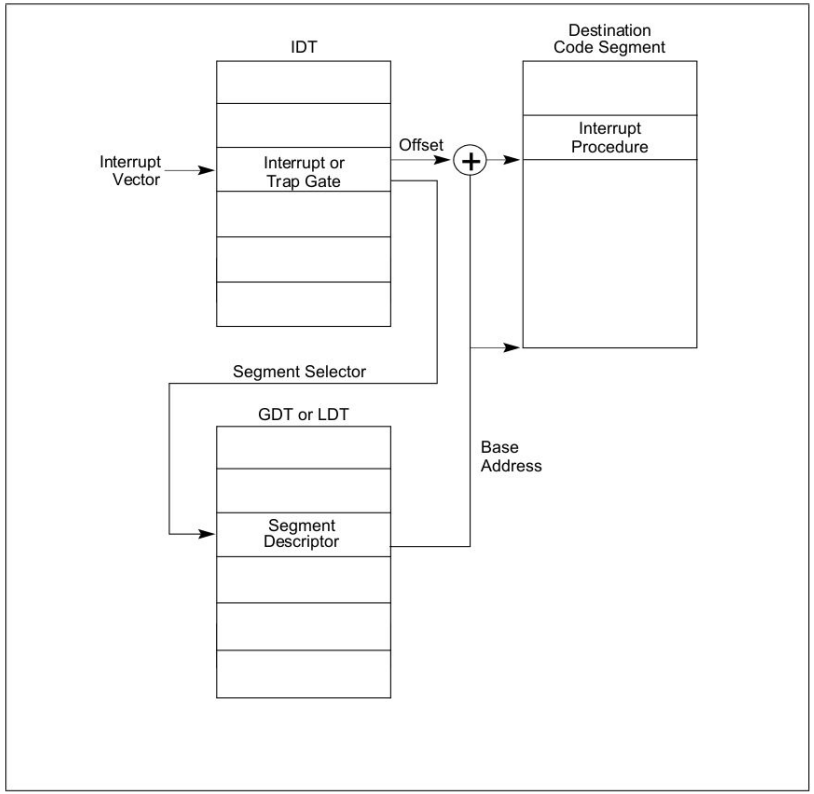


Figure 6-3. Interrupt Procedure Call

Interrupt
Init & Handle

```c
void init_mask_interrupt()
{
    /* master*/
    init_idt_desc(INT_VECTOR_IRQ0 + 0, DA_386IGate,
                  mask_int_func0, PRIVILEGE_KRNL);

    init_idt_desc(INT_VECTOR_IRQ0 + 1, DA_386IGate,
                  mask_int_func1, PRIVILEGE_KRNL);
```

```asm
ALIGN 16
mask_int_func0:                ; Interrupt r
        mask_int_func_master    0

ALIGN 16
mask_int_func1:                ; Interrupt r
        mask_int_func_master    1
```

```asm
%macro  mask_int_func_master    1
        call save ; save reg

        in al, INT_M_CTLMASK
        or al, (1<<%1)
        out INT_M_CTLMASK, al; disable this irq
        mov al, EOI
        out INT_M_CTL, al

        sti ; enable other irq

        push    %1
        call    [g_irq_table + 4 * %1]
        pop ecx

        cli ; disable all irq

        in al, INT_M_CTLMASK
        and al, ~(1<<%1)
        out INT_M_CTLMASK, al; enable this irq
        ret ; jmp to restart or restart_reenter
%endmacro
; ---------------------------------
```

Stack Switching

The operating system is responsible for creating stacks and stack-segment descriptors for all the privilege levels to be used and for loading initial pointers for these stacks into the TSS

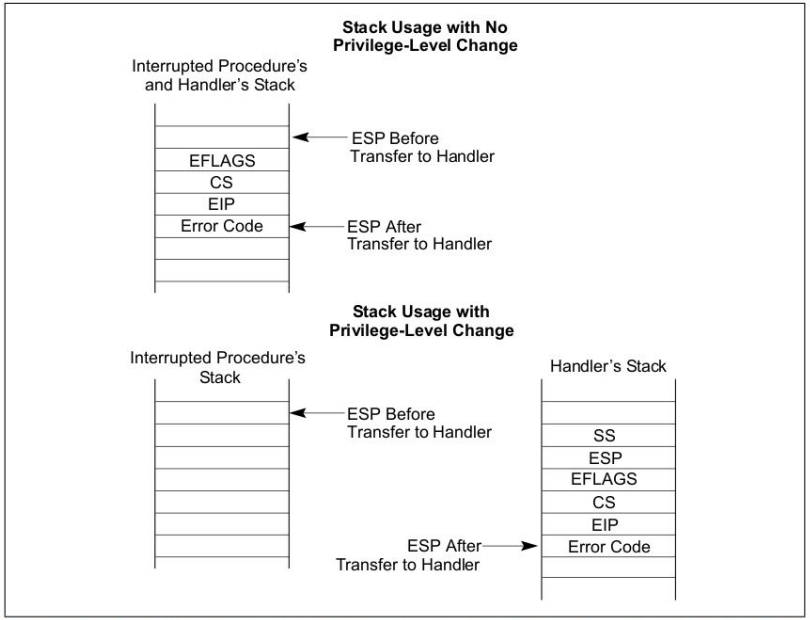


Figure 6-4. Stack Usage on Transfers to Interrupt and Exception-Handling Routines

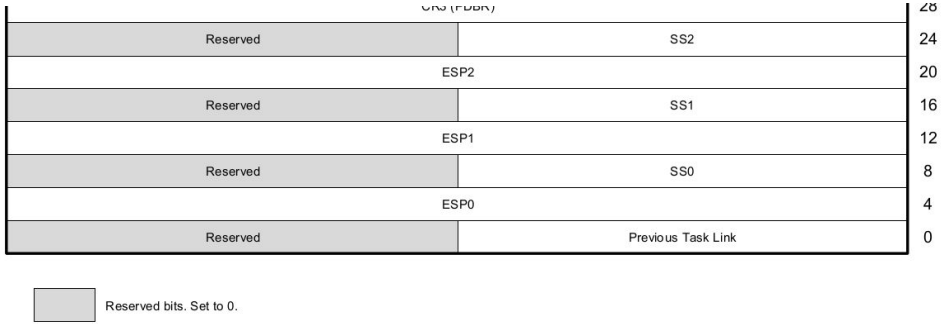


Figure 7-2. 32-Bit Task-State Segment (TSS)

Save

Step by step to draw 3 stacks!!!

```
; interrupt occur, save the reg;
; if this irq is the only one, which means
; esp point to process stackframe(not process's stack)
; we need switch to kernel stack
; otherwise, this irq may interrupt other irq,
; stack here is just the kernel stack
save:
    pushad
    push ds
    push es
    push fs
    push gs
    mov dx, ss
    mov ds, dx
    mov es, dx

    mov esi, esp

    ;Judge reenter or not
    inc dword [g_k_reenter]
    cmp dword [g_k_reenter], 0
    jne .reenter

    ;switch to kernel stack
    mov esp, StackTop
    push restart
    jmp [esi+RETADR-P_STACKBASE]
.reenter:
    push restart_reenter
    jmp [esi+RETADR-P_STACKBASE]
```

User Process Restart

Move stackframe address to esp

Move top of stackframe to tss.sp0

Pop …

Iretd back to process instruction

```
; return from irq,
; if irq is reeneter one, recover the previous irq
; otherwise, recover the process
restart:
    mov esp, [g_proc_ready]
    lldt [esp + P_LDT_SEL]
    lea eax, [esp + P_STACKTOP]
    mov dword [g_tss + TSS3_S_SP0], eax
restart_reenter:
    dec dword [g_k_reenter]
    pop gs
    pop fs
    pop es
    pop ds
    popad
    add esp, 4

    iretd
```

Questions 4: how to design Multiprocess?

What we Know：

Interrupt handle save registers to a stackframe from tss.sp0

Every process should prepare this stackframe before runing

Solutions:

Define a struct contains stackframe for process

User Process Data Structure

Prepare stack at Level 0

```c
extern proc_t* g_proc_ready;
extern proc_t g_proc_table[PROC_MAX];

extern char g_task_stack[STACK_SIZE*PROC_MAX];

```

```c
typedef struct stackframe_s
{
    uint32_t gs;
    uint32_t fs;
    uint32_t es;
    uint32_t ds;
    uint32_t edi;
    uint32_t esi;
    uint32_t ebp;
    uint32_t kernel_esp;
    uint32_t ebx;
    uint32_t edx;
    uint32_t ecx;
    uint32_t eax;
    uint32_t retaddr;
    uint32_t eip;
    uint32_t cs;
    uint32_t eflags;
    uint32_t esp;
    uint32_t ss;
}stackframe_t;

typedef struct proc_s
{
    stackframe_t regs;
    uint16_t ldt_sel;
    descriptor_t ldts[LDT_SIZE];
    uint32_t pid;
    char proc_name[PROC_NAME_MAX];
}proc_t;
```

```asm
P_STACKBASE equ 0
GSREG       equ P_STACKBASE
FSREG       equ GSREG       + 4
ESREG       equ FSREG       + 4
DSREG       equ ESREG       + 4
EDIREG      equ DSREG       + 4
ESIREG      equ EDIREG      + 4
EBPREG      equ ESIREG      + 4
KERNELESPREG    equ EBPREG      + 4
EBXREG      equ KERNELESPREG    + 4
EDXREG      equ EBXREG      + 4
ECXREG      equ EDXREG      + 4
EAXREG      equ ECXREG      + 4
RETADR      equ EAXREG      + 4
EIPREG      equ RETADR      + 4
CSREG       equ EIPREG      + 4
EFLAGSREG   equ CSREG       + 4
ESPREG      equ EFLAGSREG   + 4
SSREG       equ ESPREG      + 4
P_STACKTOP  equ SSREG       + 4
P_LDT_SEL   equ P_STACKTOP
P_LDT       equ P_LDT_SEL   + 4

TSS3_S_SP0  equ 4
```

User Process Initialize

Each process has a stack and a entry function.

```c
    proc_t* p_proc = g_proc_table;

    int idx = 0;
    for (idx = 0; idx < PROC_MAX; ++idx)
    {
        p_proc = g_proc_table + idx;
        p_proc->ldt_sel = SELECTOR_LDT_FIRST + (idx<<3);
        memcpy(&p_proc->ldts[0], &g_gdt[SELECTOR_KERNEL_CS>>3], sizeof(descriptor_t));
        p_proc->ldts[0].attr1 = DA_C | PRIVILEGE_TASK << 5; /* change the DPL*/
        memcpy(&p_proc->ldts[1], &g_gdt[SELECTOR_KERNEL_DS>>3], sizeof(descriptor_t));
        p_proc->ldts[1].attr1 = DA_DRW | PRIVILEGE_TASK << 5;   /* change the DPL*/

        p_proc->regs.cs = (0 & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK;/* this RPL is
            import when iretd*/
        p_proc->regs.ds = (8 & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK;
        p_proc->regs.es = (8 & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK;
        p_proc->regs.fs = (8 & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK;
        p_proc->regs.ss = (8 & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK;
        p_proc->regs.gs = (SELECTOR_KERNEL_GS & SA_RPL_MASK) | RPL_TASK;
        p_proc->regs.eip= (uint32_t)g_func_table[idx];
        p_proc->regs.esp= (uint32_t) g_task_stack + STACK_SIZE * idx;
        p_proc->regs.eflags = 0x1202;   /* IF=1, IOPL=1, bit 2 is always 1.*/
    }

    g_proc_ready = g_proc_table;
```

Simple & Stupid Schedule

```c
void kernel_schedule()
{
    ++g_proc_ready;
    if (g_proc_ready >= g_proc_table + PROC_MAX)
        g_proc_ready = g_proc_table;
}
```

Reference

https://software.intel.com/en-us/articles/intel-sdm