ANDYOS 实践笔记

准备工作

- 参考书《Orange'S 一个操作系统的实现》
- 参考书《NASM 中文手册》
- 安装 Bochs x86 PC 模拟器

```
$ wget "http://tenet.dl.sourceforge.net/project/bochs/bochs/2.6.8/\
    bochs-2.6.8.tar.gz"
$ tar -xzvf bochs-2.6.8.tar.gz
$ cd bochs-2.6.8
$ ./configure --enable-debugger --enable-disasm
$ make
$ sudo make install
```

- 安装 NASM 汇编器

```
sudo apt-get install build-essential nasm
```

build-essential 软件包中包含了 GCC 和 GNU Make。

启动并执行软盘引导程序

1. 启动 Bochs x86 PC 模拟器

执行 bochs 命令。执行结果提示 Bochs 需要一个配置文件：

```
...
What is the configuration file name?
To cancel, type 'none'. [none]
```

2. 添加配置文件

我们需要配置文件配置 Bochs 的外围设备和启动选项（默认是软盘启动），于是创建一个空文件，取名 *bochsrc*。之所以使用 bochsrc 这个名字，是因为 bochs 命令能自动加载名为 bochsrc 的配置文件，更简便些。

暂时不往 bochsrc 文件里填写任何内容，再次运行命令 bochs，执行结果这次变成一个 *panic*，表明这个 Bochs 没有一个可引导的设备：

```
...
00013918812p[BIOS  ] >>PANIC<< No bootable device.
```

3. 制作软盘映像文件

对于可引导设备，这里选择最简单的软盘：创建一个空的软盘映像文件，大小为 160KB（软盘最小容量，40 轨 8 扇区），取名 a.img。通过 *xxd -a* 命令可以查看 a.img 的内容。过程演示如下：

```
$ dd if=/dev/zero of=a.img bs=512 count=320
$ xxd -a a.img
0000000: 0000 0000 0000 0000 0000 0000 0000 0000  ................
*
0027ff0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

4. 修改 bochsrc 文件

这个时候 a.img 和 Bochs 没有任何联系，需要通过配置文件来将 a.img 作为软盘插入 Bochs。

```
echo 'floppya: image="a.img", status=inserted'>> bochsrc
```

5. 制作可引导扇区

现在 Bochs 已经拥有一个软盘设备，但这个软盘仍然不是可引导的。Bochs 的 BIOS 认为一个可引导软盘必须有一个引导扇区，引导扇区具备以下要素：

- 引导扇区是软盘的第一个扇区，即开始的 512 个字节
- 引导扇区的最后两个字节必须是 0xAA55
- 引导程序也是必须的，引导程序从第 1 个字节开始，长度不能超过 510 字节

对于引导程序，这里选择 NASM 格式的汇编。创建 boot.asm 文件，写入以下内容：

```
jmp $
times 510-($-$$) db 0
dw 0xAA55
```

将以上汇编代码编译成无格式的二进制文件 boot.bin。通过 *xxd -a* 查看 boot.bin 的内容，正好 512 字节，且最后两个字节是 0xAA55。最后将 boot.bin 这 512 字节拷贝到 a.img，就完成了。

6. 最后的成果

演示过程如下：

```
$ nasm -o boot.bin boot.asm
$ xxd -a boot.bin
0000000: ebfe 0000 0000 0000 0000 0000 0000 0000  ................
0000010: 0000 0000 0000 0000 0000 0000 0000 0000  ................
*
00001f0: 0000 0000 0000 0000 0000 0000 0000 55aa  ..............U.
$ dd if=boot.bin of=a.img conv=notrunc
$ xxd -a a.img
0000000: ebfe 0000 0000 0000 0000 0000 0000 0000  ................
0000010: 0000 0000 0000 0000 0000 0000 0000 0000  ................
*
00001f0: 0000 0000 0000 0000 0000 0000 0000 55aa  ..............U.
0000200: 0000 0000 0000 0000 0000 0000 0000 0000  ................
*
0027ff0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

那么，现在为止，Bochs 就有了一个可引导的软盘，现在启动 Bochs 应该没有问题了。结果一切顺利，bochs 的 BIOS 成功识别软盘启动盘，加载并执行其中的引导程序，最终引导程序停在 0000:7c00 处。最终结果是这样的：

```
...
00014040199i[BIOS  ] Booting from 0000:7c00
^C00118932892i[      ] Ctrl-C detected in signal handler.
Next at t=118932893
(0) [0x000000007c00] 0000:7c00 (unk. ctxt): jmp .-2 (0x00007c00)      ; ebfe
<bochs:2>
```

引导程序读取软盘，并执行内核程序

Bochs BIOS 的 13h 号中断可以实现从软盘读取数据，其描述如下：

寄存器	含义
ah	2
al	扇区数
es:bx	内存目标地址
ch	磁道号
cl	扇区号（** 从 1 开始 **）
dh	磁头号
dl	驱动器号（0 表示 A 盘）

新建一个 andyos.asm 文件，其中的 NASM 汇编代码如下：

```
    mov ax, 1000h
    mov es, ax
    mov bx, 0       ;es:bx
    mov ch, 0       ;trk
    mov cl, 1       ;sctr
    mov dh, 0       ;head
    mov dl, 0       ;floppya
LOOP:
    mov ah, 2
    mov al, 8       ;count
    int 13h
    cmp ax, 8
    jne ERROR       ;if read-error; then error
    inc ch          ;trk++
    cmp ch, 16
    je  NEXT_64K    ;if ch=16; then next64k
    cmp ch, 32
    je  NEXT_64K    ;if ch=32; then next64k
    cmp ch, 40
    je  FINISH      ;if ch=40; then finish
    add bx, 1000h   ;bx+=512*8
    jmp LOOP
NEXT_64K:
    mov ax, es
    add ax, 1000h
    mov es, ax
    xor bx, bx
    jmp LOOP
```

```
FINISH:
    jmp 1000h:512
ERROR:
    jmp $
times 510-($-$$) db 0
dw 0xaa55

    jmp $

times 512*320-($-$$) db 0xA
```

这个文件中编译完成后的二进制文件正好是 160KB，可以作为最小的软盘映像文件。上面的代码完成的功能是加载整个 160KB 的软盘映像文件到内存 0x10000 到 0x37FFF 位置。由于 160KB 软盘是 40 轨 8 扇区的，因此每次循环读 1 轨，直到读完 40 轨。

修改 makefile 如下：

```
all:
    echo 'floppya: image="a.img", status=inserted' > bochsrc
    nasm andyos.asm -o a.img

clean:
    rm -f a.img bochsrc

run:
    bochs -q
```

演示结果如下：

```
$ make
echo 'floppya: image="a.img", status=inserted' > bochsrc
nasm andyos.asm -o a.img
$ make run
...
^C00199247823i[      ] Ctrl-C detected in signal handler.
Next at t=199247824
(0) [0x000000010200] 1000:0200 (unk. ctxt): jmp .-2 (0x00010200)      ; ebfe
<bochs:2> x /16xb 0x37ff8
[bochs]:
0x00037ff8 <bogus+     0>:   0x0a   0x0a   0x0a   0x0a   0x0a   0x0a   0x0a   0x0a
0x00038000 <bogus+     8>:   0x04   0xff   0x76   0xf8   0xb8   0x04   0x00   0x50
...
```

保护模式

- lgdt [gdt_ptr]
- open a20 to access more memory
- pe(protect enabled?) bit in cr0
- jmp to change value of cs

gdt_ptr 是一个 6 字节共 48 位的变量，低 16 位表示 limit，高 32 位表示 base，base 和 limit 结合指示 gdt 在内存中的位置。

```
gdt_ptr{15..0}  = limit
gdt_ptr{47..16} = base
```

base 是基地址，是一个 32 位的绝对地址；limit 是界限，数值上等于长度 -1（长度单位可以是字节，也可以是 4KB）。

gdt 是一个数组，每个元素是一个 DESCRIPTOR 描述符，占 8 个子节。一个 DESCRIPTOR 描述一个内存区间，或者说内存段。这个段在内存中的位置由 base 和 limit 共同决定，此外段还有属性集合 attr，包含段的各种属性。

attr 的各个属性含义如下：

位名称	含义
G 位	粒度，0 为字节，1 为 4 KB
D/B 位	1 为 32 位，0 为 16 位（代码段 cs/堆栈段 ss）
AVL 位	系统保留位
P 位	Present，1 表示在内存中
DPL 位	特权级，0 最高 3 最低
S 位	1 为数据/代码段，0 为系统段/门描述符
TYPE 位	见下表

DESCRIPTOR 中 base 占 32 位，limit 占 20 位，attr 占 12 位。他们在 DESCRIPTOR 中的组织形式如下：

```
DESCRIPTOR{15..0}   = limit{15..0}
DESCRIPTOR{39..16}  = base{23..0}
DESCRIPTOR{43..40}  = attr.TYPE
DESCRIPTOR{44}      = attr.S
DESCRIPTOR{46..45}  = attr.DPL
DESCRIPTOR{47}      = attr.P
DESCRIPTOR{51..48}  = limit{19..16}
DESCRIPTOR{52}      = attr.AVL
DESCRIPTOR{53}      = 0
DESCRIPTOR{54}      = D/B
DESCRIPTOR{55}      = G
DESCRIPTOR{63..56}  = base{31..24}
```

TYPE 值	数据/代码段 (S=1)	系统段/门描述符 (S=0)
0	只读	< 未定义 >
1	只读,AC	可用 286TSS
2	读/写	LDT
3	读/写,AC	忙的 286TSS
4	只读,ED	286 调用门
5	只读,ED,AC	任务门
6	读/写,ED	286 中断门
7	读/写,ED,AC	286 陷阱门
8	只执行	< 未定义 >
9	只执行,AC	可用 386TSS
A	执行/读	< 未定义 >
B	执行/读,CC	忙的 386TSS
C	只执行,CC	386 调用门
D	只执行,CC,AC	< 未定义 >
E	执行/读,CC	386 中断门
F	执行/读,CC,AC	386 陷阱门

属性集合使用示例：

- (1100 0000 1001 1010) 表示 4KB 粒度，可执行且可读的 32 位数据/代码段
- (1100 0000 1001 0010) 表示 4KB 粒度，可读写的数据/代码段（如果 ss 指向这个段，则隐式堆栈访问指令 push，pop，call 等使用 32 位 esp）
- (0000 0000 1111 0010) 表示可读写的 DPL 为 3 的数据/代码段

在 andyos.asm 文件中中添加如下代码。
这里是 DESCRIPTOR 宏的定义：

```
;define or consts(NAME_IT_LIKE_THIS)
;-----------------------------------
%macro DESCRIPTOR   3   ;base limit attr
    dw  %2 & 0xFFFF
    dw  %1 & 0xFFFF
    db  (%1 >> 16) & 0xFF
    dw  ((%2 >> 8) & 0xF00) + (%3 & 0xF0FF)
    db  (%1 >> 24) & 0xFF
%endmacro
```

这里是 gdt 和 gdt_ptr 的定义：

```
;variables that need memory(name_it_like_this)
;---------------------------------------------
gdt:
```

```
    DESCRIPTOR  0, 0, 0 ;(H->L:||G|DB|0|AVL|+|0000|+|P|DPL2|S|+|TYPE4||)
    DESCRIPTOR  0, 0xFFFFF, ((1100b<<12)+(1001b<<4)+0xA);0xA exec/read
    DESCRIPTOR  0, 0xFFFFF, ((1100b<<12)+(1001b<<4)+0x2);0x2 read/write
    DESCRIPTOR  0xB8000, 0xFFFF, ((0000b<<12)+(1001b<<4)+0x2)
gdt_ptr:
    dw      31
    dd      0x10202
```

这里是代码段的定义：

```
jmp _start
;here is define and consts
;-------------------------
;......
;here is variables
;-----------------
;......
;codes(_name_it_like_this)
;-------------------------
_start:
    mov     ax, cs
    mov     ds, ax
    lgdt    [ds:(gdt_ptr-$$)]
    cli
    in      al, 0x92
    or      al, 2
    out     0x92, al
    mov     eax, cr0
    or      eax, 1
    mov     cr0, eax
    jmp     dword 8:((_protect_entry-$$)+0x10000)

[BITS 32]
_protect_entry:
    mov     ax, 24
    mov     gs, ax
    mov     ax, 16
    mov     ds, ax
    mov     es, ax
    mov     fs, ax
    mov     ss, ax
    mov     esp, 0x10000
```

```
mov     ah, 0x0C
mov     al, 'p'
mov     [gs:0], ax
jmp     $
```

中断门和 *IDT*

中断实际上是程序执行过程中的强制转移，转移到相应的处理程序。中断通常在程序执行时因为硬件而随机发生，他们通常用来处理外部的事件，比如外围设备的请求。软件通过执行 *int n* 指令也可以产生中断。

简单而言，中断是软件或者硬件发生了某种情形而通知处理器的行为。于是，由此引出三个要素：

- 处理器可以对何种类型的通知做出反应
- 处理器接到通知后做出何种处理
- 不同中断和不同处理之间的对应关系（由 IDT 来映射）

中断门和 DESCRIPTOR 类似，占用 8 个字节共 64 位。中断门由 selector，offset 和 attr 三部分组成，selector 指定中断处理程序所在的代码段，offset 指定中断处理程序在该段中的偏移，attr 是一个属性集合。其中 selector 占用 16 位，offset 占用 32 位，attr 占用 16 位。具体的结构如下：

```
INTERRUPT_GATE{15..0}    = offset{15..0}
INTERRUPT_GATE{31..16} = selector
INTERRUPT_GATE{47..32} = attr
INTERRUPT_GATE{63..48} = offset{31..16}
```

attr 的具体结构和内容（中断门的属性是确定的）如下：

```
attr{4..0}  = reserved
attr{7..5}  = 0
attr{11..8} = TYPE(0xE：386中断门)
attr{12}    = S(0：系统段/门描述符)
attr{14..13}= DPL(0)
attr{15}    = P(1)
```

将中断门写成宏的形式：

```
%macro INTERRUPT_GATE   2   ;selector offset
    dw  %2 & 0xFFFF
    dw  %1
    dw  (10001110b<<8)      ;(P=1,DPL=00,S=0,TYPE=1110:386int_gate)
    dw  (%2 >> 16) & 0xFFFF
%endmacro
```

IDT 的内容如下：

```
idt:
    %rep    32
    INTERRUPT_GATE      0, 0
    %endrep
    INTERRUPT_GATE      8, (0x10000+(_int_handler-$$))
```

```
idt_ptr:
    dw      (33*8-1)
    dd      (0x10000+(idt-$$))
```

其中 int*handler* 指向的中断处理程序如下：

```
_int_handler:
    mov     ah, 0x0C
    mov     al, 'I'
    mov     [gs:2], ax
    jmp     $
    iretd
```

修改 protect*entry* 中的指令，使之加载 IDT 并使用 *int 32* 指令调用第 32 号中断：

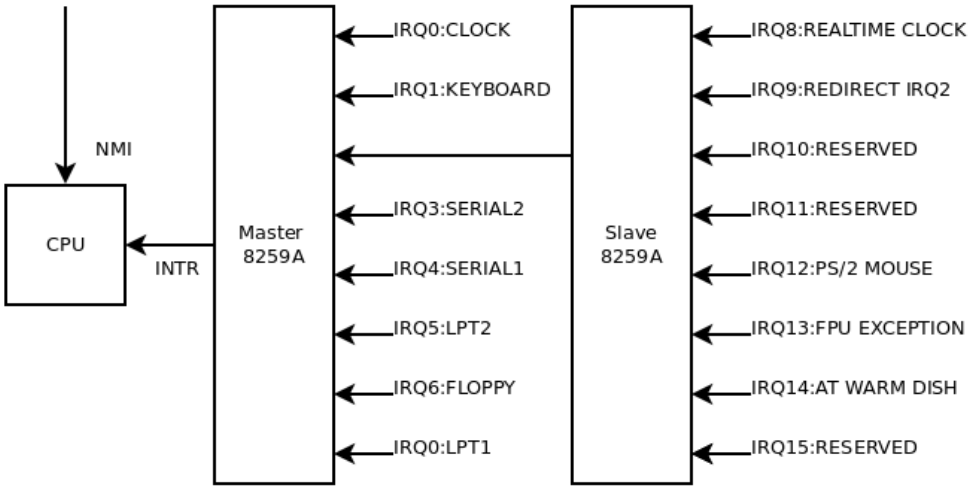
```
    lidt    [ds:(0x10000+(idt_ptr-$$))]
    int     32
```

至此就完成了一个简单的中断门实验。

时钟中断

中断分为软件中断和硬件中断。上一节中使用 *int 32* 来调用中断属于软件中断，这一节要实现的时钟中断属于硬件中断。硬件中断又称外部中断。

外部中断的情况更复杂一些，因为需要建立硬件中断与向量号之间的对应关系。外部中断分为不可屏蔽中断（NMI）和可屏蔽中断两种，分别由 CPU 的两根引脚 NMI 和 INTR 来接收。NMI 不可屏蔽，也就是说它与 IF 位是否被设置无关。



可屏蔽中断与 CPU 的关系是通过对可编程中断控制器 *8259A* 建立起来的。8259A 相当于外围设备的一个代理，这个代理不但可以根据优先级在同时发生中断的设备中选择应该处理的请求，而且可以通过对其寄存器的设置来屏蔽或打开相应的中断。

在上图中可以看到和 CPU 相连的是两片级联的 8259A，每个 8259A 有 8 根中断信号线，于是两片级联总共可以挂接 15 个不同的外部设备。

通过对 8259A 的设置可以将设备发出的中断信号和中断向量号对应起来。主 8259A 对应的端口地址是 20h 和 21h，从 8259A 对应的端口地址是 A0h 和 A1h。通过向 8259A 相应端口写入特定的内容可以进行设置。写入的内容包括 ICW1，ICW2，ICW3，ICW4，OCW1，OCW2 六个控制字。ICW 即 Initialization Control Word；OCW 即 Operation Control Word。

ICW 和 OCW 都是 1 个字节共 8 位，其各个位的含义如下：

```
ICW1 (20h或者A0h)
ICW1{0}     = 1 (需要ICW4)
ICW1{1}     = 0 (级联8259A)
ICW1{2}     = 0 (8字节中断向量)
ICW1{3}     = 0 (edge triggered模式，即满足条件触发)
ICW1{4}     = 1 (对ICW1必须为1，端口必须为20h或A0h)
ICW1{7..5}  = 0 (对PC系统必须为0)

ICW2 (21h或者A1h)
ICW2{2..0}  = 0 (80x86系统)
ICW2{7..3}  = 0x20 (中断向量号，表示IRQ0或者IRQ8对应的中断向量号，其他递增)
```

```
ICW3 (21h, 主片)
ICW3    = 00000100b (表示IRQ2接从片)

ICW3 (A1h, 从片)
ICW3{2..0}  = 0x2 (表示连的主片IRQ号)
ICW3{7..3}  = 0

ICW4 (21h或者A1h)
ICW4{0}     = 1 (80x86模式)
ICW4{1}     = 0 (正常EOI)
ICW4{3..2}  = 0 (主/从缓冲模式)
ICW4{4}     = 0 (顺序模式)
ICW4{7..5}  = 0 (未使用)

OCW1 (21h或者A1h)
OCW1{7..0}  = 11111110b (1该IRQ禁用, 0打开)

OCW2 (20h或者A0h)
OCW2{5}     = 1 (发送EOI位使之继续接收中断, 否则不再接收任何中断)
```

初始化过程需要向 8259A 写入 ICW（顺序很重要）：

1. 向 20h 端口或 A0h 端口写入 ICW1
2. 向 21h 端口或 A1h 端口写入 ICW2
3. 向 21h 端口或 A1h 端口写入 ICW3
4. 向 21h 端口或 A1h 端口写入 ICW4

向代码中添加 *IO_DELAY* 宏，因为在操作 8259A 时，应该等待其完成后再继续执行其他指令。

```
%define IO_DELAY    times 10 nop
```

添加初始化 8259A 的代码：

```
_init_8259a:
    mov     al, 0x11    ;ICW1
    out     0x20, al
    IO_DELAY
    out     0xA0, al
    IO_DELAY
    mov     al, 0x20    ;ICW2
    out     0x21, al
    IO_DELAY
    mov     al, 0x28
```

```
    out     0xA1, al
    IO_DELAY
    mov     al, 0100b   ;ICW3
    out     0x21, al
    IO_DELAY
    mov     al, 0x2
    out     0xA1, al
    IO_DELAY
    mov     al, 0x1     ;ICW4
    out     0x21, al
    IO_DELAY
    out     0xA1, al
    IO_DELAY
    mov     al, 0xFE    ;OCW1
    out     0x21, al
    IO_DELAY
    mov     al, 0xFF
    out     0xA1, al
    IO_DELAY
    ret
```

修改 _int_handler 的代码，重要的是向 8259A 发送 EOI 使其继续接收中断。

```
_int_handler:
    inc     word [gs:2]
    mov     al, 0x20
    out     0x20, al
    IO_DELAY
    iretd
```

最后修改 _protect_entry 中的代码，调用初始化 8259A 的代码，并设置 IF 位打开：

```
    call    _init_8259a
    sti
    jmp     $
```

进程和多进程调度

目标是并行执行 A 和 B 两个进程，A 进程的执行的程序如下，B 进程的和 A 基本一样，只不过不打印'A' 而打印'_'。

```
_process_a:
    mov     ebx, 0xFFFFFF
.loop:
    mov     ah, 0x7
    mov     al, 'A'
    mov     edi, dword [cursor]
    mov     [gs:edi], ax
    add     edi, 2
    mov     dword [cursor], edi
.wait:
    cmp     ebx, 0
    jne     .continue
    mov     ebx, 0xFFFFFF
    jmp     .loop
.continue:
    dec     ebx
    jmp     .wait
```

PCB 是必要的，由于没有特权级转移（只有内核级），所以只需要保存每个进程的 8 个普通寄存器，eip 及 eflags。这是新增的全局数据结构，其中 PCB 只是 10 个 4 字节的寄存器。

```
cursor      equ ADDR
    dd      0
process_a   equ ADDR
    times   10  dd 0    ;8 normal+eip+eflags
process_b   equ ADDR
    times   10  dd 0
proc_running equ ADDR
    dd      process_a
proc_waiting equ ADDR
    dd      process_b
```

对每个 PCB 都要初始化，包括 eip，eflags 和 esp。以下代码在 _protect_entry 中：

```
    mov     eax, dword (0x10000+(_process_a-$$))
    mov     dword [process_a+EIP], eax
    mov     eax, 0x10000
    mov     dword [process_a+ESP], eax
    mov     eax, 0x202      ;IF=1,IOPL=0
```

```
    mov     dword [process_a+EFLAGS], eax

    mov     eax, dword (0x10000+(_process_b-$$))
    mov     dword [process_b+EIP], eax
    mov     eax, 0xF000
    mov     dword [process_b+ESP], eax
    mov     eax, 0x202      ;IF=1,IOPL=0
    mov     dword [process_b+EFLAGS], eax

    lidt    [idt_ptr]
    call    _init_8259a
    sti
    jmp     _process_a
```

最重要的进程调度模块。

```
_clock_handler:
    cli
    push    edi
    mov     edi, dword [proc_running]
    mov     dword [edi+EAX], eax
    mov     dword [edi+EBX], ebx
    mov     dword [edi+ECX], ecx
    mov     dword [edi+EDX], edx
    mov     eax, edi
    pop     edi
    mov     dword [eax+EDI], edi
    mov     dword [eax+ESI], esi
    mov     ebx, ebp
    mov     dword [eax+EBP], ebx
    pop     ebx     ;eip
    mov     dword [eax+EIP], ebx
    pop     ebx     ;cs no use
    pop     ebx     ;eflags
    mov     dword [eax+EFLAGS], ebx
    mov     ebx, esp
    mov     dword [eax+ESP], ebx

    mov     eax, dword [proc_running]
    mov     ebx, dword [proc_waiting]
    mov     dword [proc_running], ebx
    mov     dword [proc_waiting], eax
```

```
mov     edi, dword [proc_running]
mov     ebx, dword [edi+EBX]
mov     ecx, dword [edi+ECX]
mov     edx, dword [edi+EDX]
mov     esi, dword [edi+ESI]
mov     eax, dword [edi+EBP]
mov     ebp, eax
mov     eax, dword [edi+ESP]
mov     esp, eax
mov     eax, dword [edi+EFLAGS]
push    eax     ;eflags
push    8       ;cs
mov     eax, dword [edi+EIP]
push    eax     ;eip

inc     word [gs:0]
mov     al, 0x20
out     0x20, al
sti

mov     eax, dword [edi+EAX]
mov     edi, dword [edi+EDI]

iretd
```

涉及特权级转换的多进程调度

1. 如何从 ring0 跳转到 ring3
2. 中断发生时 CPU 自动做了什么
3. 中断时怎样切换进程 PCB

```
_clock_handler:
    pushad      ;esp no use
    push    ds
    push    es
    push    fs
    push    gs  ;now 17 reg in stack(2esp)

    mov     ax, ss
    mov     ds, ax
    inc     word [0xB8000]
    mov     al, 0x20
    out     0x20, al

    mov     eax, dword [proc_run]
    mov     ebx, dword [proc_ready]
    mov     dword [proc_run], ebx
    mov     dword [proc_ready], eax

    mov     eax, dword [proc_run]
    mov     esp, eax
    add     eax, STACKTOP
    mov     dword [tss+TSS_ESP0], eax

    pop     gs
    pop     fs
    pop     es
    pop     ds
    popad

    iretd

    mov     ax, ss
    mov     [tss+TSS_SS0], ax
    mov     eax, dword [proc_run]
    add     eax, STACKTOP
    mov     [tss+TSS_ESP0], eax
```

```
mov     eax, esp
mov     esp, process_a+STACKTOP
push    0xF     ;ss
push    0xF000  ;esp
push    0x1202  ;eflags
push    0x7     ;cs
push    0x10000+(_process_a-$$) ;eip
push    0       ;edi
push    0       ;esi
push    0       ;ebp
push    0       ;esp
push    0       ;ebx
push    0       ;edx
push    0       ;ecx
push    0       ;eax
push    0xF     ;ds
push    0xF     ;es
push    0xF     ;fs
push    0xF     ;gs
mov     esp, process_b+STACKTOP
push    0xF     ;ss
push    0xE000  ;esp
push    0x1202  ;eflags
push    0x7     ;cs
push    0x10000+(_process_b-$$) ;eip
push    0       ;edi
push    0       ;esi
push    0       ;ebp
push    0       ;esp
push    0       ;ebx
push    0       ;edx
push    0       ;ecx
push    0       ;eax
push    0xF     ;ds
push    0xF     ;es
push    0xF     ;fs
push    0xF     ;gs
mov     esp, eax

mov     ax, 24
```

```
    ltr     ax
    mov     ax, 32
    lldt    ax
    lidt    [idt_ptr]
    call    _init_8259a
    sti

    push    0xF
    push    0xF000
    push    0x7
    push    0x10000+(_process_a-$$)
    retf

process_a   equ ADDR
    times   4   dd 0    ;gs fs es ds
   times  8  dd 0   ;edi esi ebp ESP ebx edx ecx eax (pushad, popad-ignore esp)
    times   5   dd 0    ;eip cs eflags esp ss (auto stack)
process_b   equ ADDR
    times   17  dd 0
proc_run    equ ADDR
    dd      process_a
proc_ready  equ ADDR
    dd      process_b
tss         equ ADDR
    times   25 dd 0 ;1backlink 6esp/ss 17regs ldt
    dw      0   ;debug trap
    dw      104 ;I/O base
    db      0xFF;end of I/O

gdt     equ ADDR
    DESCRIPTOR  0, 0, 0 ;(H->L:||G|DB|0|AVL|+|0000|+|P|DPL2|S|+|TYPE4||)
    DESCRIPTOR  0, 0xFFFFF, ((1100b<<12)+(1001b<<4)+0xA);0xA exec/read
    DESCRIPTOR  0, 0xFFFFF, ((1100b<<12)+(1001b<<4)+0x2);0x2 read/write
    DESCRIPTOR  tss, 104,   ((1000b<<4)+0x9)    ;0x9 avail 386 tss
    DESCRIPTOR  ldt, 16,    ((1000b<<4)+0x2)    ;0x2 ldt
gdt_ptr equ ADDR
    dw      (5*8-1)
    dd      gdt
ldt     equ ADDR    ;same ldt for every process
    DESCRIPTOR  0, 0xFFFFF, ((1100<<12)+(1111b<<4)+0xA)
    DESCRIPTOR  0, 0xFFFFF, ((1100<<12)+(1111b<<4)+0x2)

%define TSS_ESP0    4
```

```
%define TSS_SS0     8
%define STACKTOP    4*17
%define GS          0
%define FS          4
%define ES          8
%define DS          12
%define EDI         16
%define ESI         20
%define EBP         24
%define ESP         28
%define EBX         32
%define EDX         36
%define ECX         40
%define EAX         44
%define EIP         48
%define CS          52
%define EFLAGS      56
%define ESP         60
%define SS          64
```