武汉大学国家网络安全学院教学实验报告

课程名称	操作系统设计与实践	实验日期	2022.10.9
实验名称	中断与异常	实验周次	第四周
姓名	学号	专业	班级
李心杨	2020302181022	信息安全	2
王宇骥	2020302181008	信息安全	2
林锟扬	2020302181032	信息安全	2
郑炳捷	2020302181024	信息安全	2

1 实验目的及实验内容

1.1 实验目的

理解中断与异常机制的实现机理。

1.2 实验内容

1. 理解中断与异常的机制。
2. 调试 8259A 的编程基本例程。
3. 调试时钟中断例程。
4. 实现一个自定义的中断向量，功能可自由设想。

2 实验环境及实验步骤

2.1 实验环境

- Ubuntu 16.04.1;
- VMWare Workstation 16 player;
- bochs 2.7。

2.2 实验步骤

1. 调试 8259A 的编程基本例程。
2. 调试时钟中断例程。
3. 实现一个自定义的中断向量。

3 实验过程分析

3.1 调试 8259A 的编程基本例程

3.1.1 代码阅读

相较于 pmtest8.asm，pmtest9a.asm 新增内容主要为建立 IDT 和初始化 8259A 两部分。

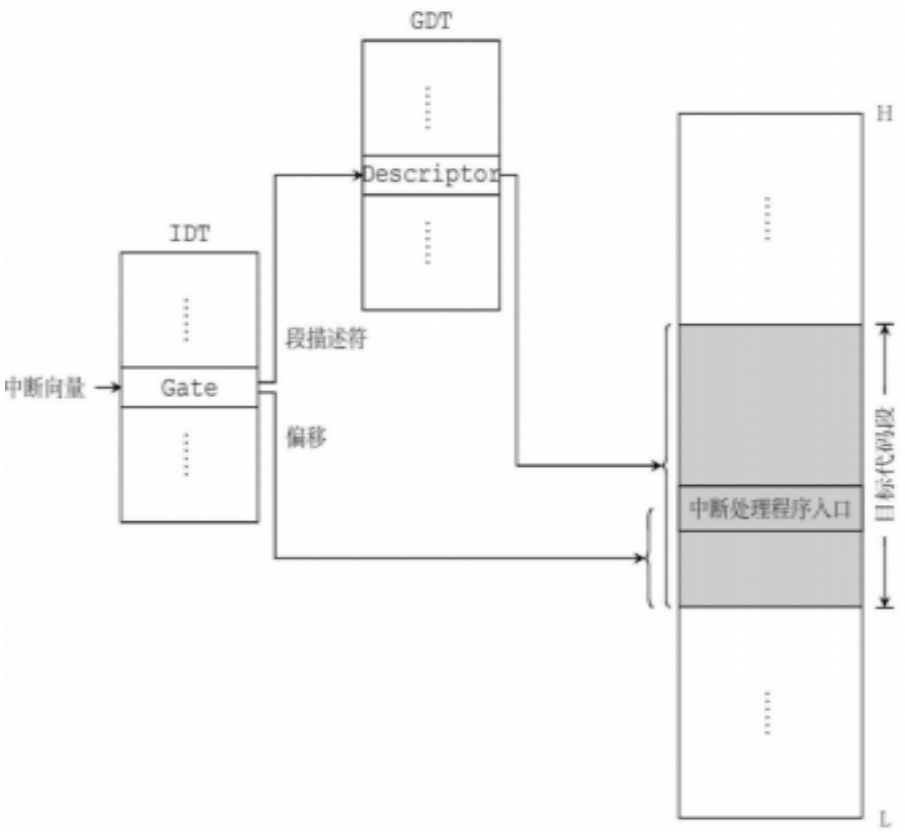

图 1: IDT 寻址方式

• 建立 IDT：IDT、IDTR 的初始化与寻址方式与前面实验中的 GDT、LDT 类似，寻址方式见图1。

```
; IDT
[SECTION .idt]
ALIGN   32
[BITS   32]
LABEL_IDT:
; 门                    目标选择子,             偏移, DCount, 属性
%rep 255
        Gate    SelectorCode32, SpuriousHandler, 0, DA_386IGate
%endrep

IdtLen      equ $ - LABEL_IDT
IdtPtr      dw  IdtLen - 1  ; 段界限
        dd  0       ; 基地址
```

```
; END of [SECTION .idt]
```

```
    ; 为加载 IDTR 作准备
    xor eax, eax
    mov ax, ds
    shl eax, 4
    add eax, LABEL_IDT          ; eax <- idt 基地址
    mov dword [IdtPtr + 2], eax ; [IdtPtr + 2] <- idt 基地址

    ; 加载 GDTR
    lgdt    [GdtPtr]

    ; 关中断
    cli

    ; 加载 IDTR
    lidt    [IdtPtr]
```

可以看到在 IDT 的初始化中，将 255 个描述符全部指向了同一个中断门 Selector-Code32:SpuriousHandler，而这个中断处理程序的内容是在特定位置打印一个 “!”：

```
_SpuriousHandler:
    SpuriousHandler equ _SpuriousHandler - $$
    mov ah, 0Ch             ; 0000: 黑底    1100: 红字
    mov al, '!'
    mov [gs:((80 * 0 + 75) * 2)], ax    ; 屏幕第 0 行, 第 75 列。
    xchg    bx, bx
    jmp $
    iretd
```

- 初始化保护模式下的 8259A：

```
; Init8259A ---------------------------------------
Init8259A:
    mov al, 011h
    out 020h, al    ; 主8259, ICW1.
    call    io_delay

    out 0A0h, al    ; 从8259, ICW1.
    call    io_delay

    mov al, 020h    ; IRQ0 对应中断向量 0x20
    out 021h, al    ; 主8259, ICW2.
    call    io_delay

    mov al, 028h    ; IRQ8 对应中断向量 0x28
    out 0A1h, al    ; 从8259, ICW2.
    call    io_delay

    mov al, 004h    ; IR2 对应从8259
    out 021h, al    ; 主8259, ICW3.
    call    io_delay

    mov al, 002h    ; 对应主8259的 IR2
    out 0A1h, al    ; 从8259, ICW3.
```

```
24      call    io_delay
25
26      mov al, 001h
27      out 021h, al    ; 主8259, ICW4.
28      call    io_delay
29
30      out 0A1h, al    ; 从8259, ICW4.
31      call    io_delay
32
33      mov al, 11111110b   ; 仅仅开启定时器中断
34      ;mov    al, 11111111b   ; 屏蔽主8259所有中断
35      out 021h, al    ; 主8259, OCW1.
36      call    io_delay
37
38      mov al, 11111111b   ; 屏蔽从8259所有中断
39      out 0A1h, al    ; 从8259, OCW1.
40      call    io_delay
41
42      ret
43  ; Init8259A ---------------------------------------
```

初始化的这些寄存器的结构在4.3.3一节中已经详细说明，此处不再多做解释。可以看到在初始化 ICW2 时，将 IRQ0~IRQ15 对应到了 20h~2Fh 的位置，并且屏蔽了除了时钟中断以外所有的中断。不过由于我们将所有 IDT 描述符全都指向同一个打印程序，所以这些设定暂时没有体现出用处，他们将在3.2中被用到。

3.1.2 调试程序

原始代码执行后结果如下图：

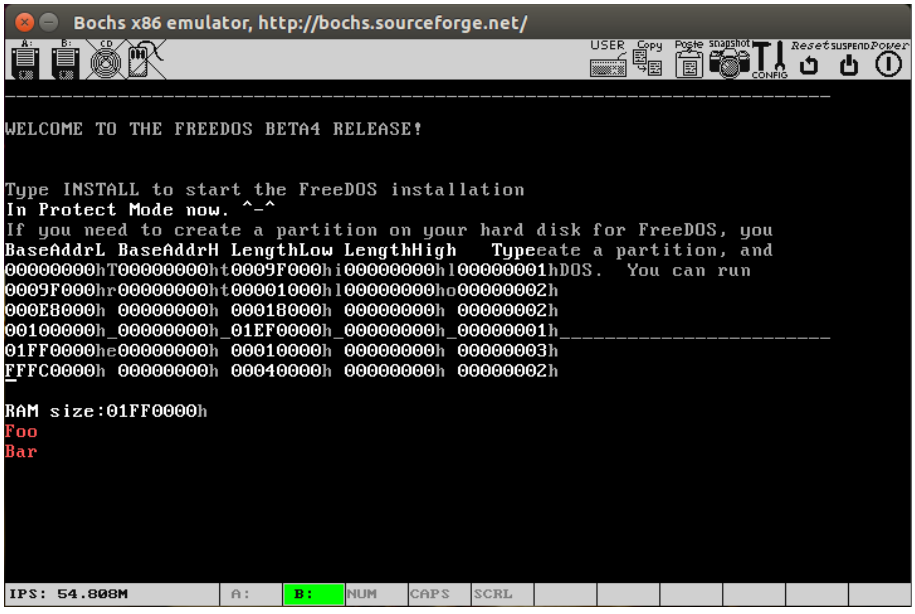

图 2: pmtest9a.asm 执行结果

在原始代码基础上进行以下修改：将关中断指令 cli 注释，并且在保护跳回实模式

之前增加指令 jmp $ 使得程序停留在保护模式下，此时程序执行结果如下：

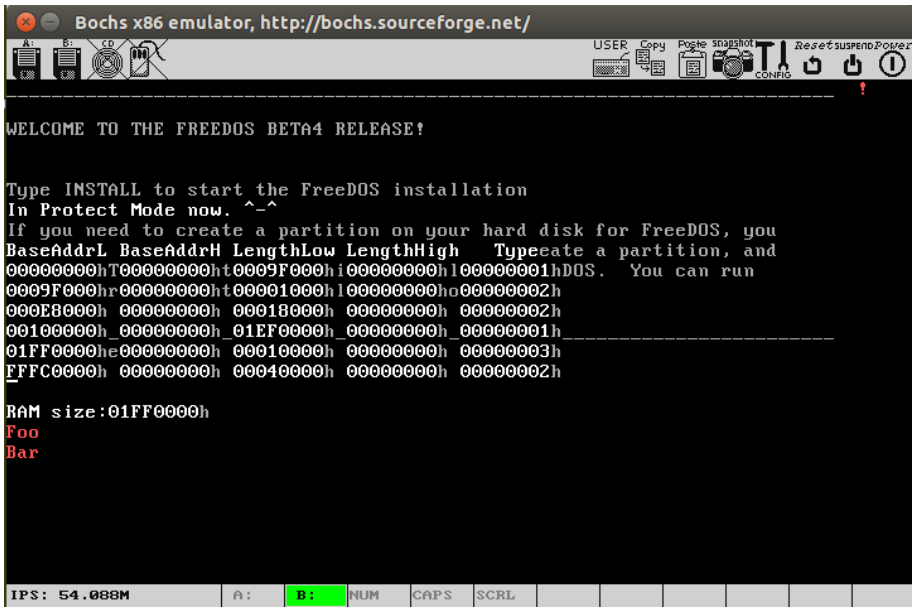

图 3: pmtest9a.asm 修改后的执行结果

在原始代码中，最后会返回到实模式，但是代码中并未在返回实模式后将 8259A 和应有的实模式中断处理还原回去，而仍然是保护模式下的设置，推测是由于在保护模式时中断处理程序并没有来得及执行，所以处于实模式的系统是无法正确识别保护模式下设置的中断处理程序并打印出的“!”的。第一次调试时，我们将断点设置在了 SpuriousHandler 中，但是并没有触发断点，推测是中断处理程序根本没有执行，从而发现了上述问题。故我们通过对 pmtest9a 作出如 Listing 1的修改，即在保护模式下开中断并在最后增加 jmp $ 来使系统停在保护模式后，得到符合预期的结果，成功打印出了红色“!”。

Listing 1: 使 pmtest9a 能正常显示叹号

```
1  --- a/i/pmtest9a.asm
2  +++ b/i/pmtest9a.asm
3  @@ -275,6 +275,9 @@ LABEL_SEG_CODE32:
4
5          call    PagingDemo              ; 演示改变页目录的效果
6  +       sti
7  +       jmp     $
8  +       ; 不返回实模式
9          ; 到此停止
10         jmp     SelectorCode16:0
```

需要注意，pmtest9a.asm 中并没有调用 Init8259A，所以该程序中 8259A 的状态并不是我们设置的，是默认的全嵌套方式。我们不能确定最终打印出“!”一定是通过时钟中断产生的。

3.2 调试时钟中断例程

3.2.1 代码阅读

相较于 pmtest9a，pmtest9 的变化如下：

- IDT 初始化新增中断处理程序：

```
; IDT
[SECTION .idt]
ALIGN   32
[BITS   32]
LABEL_IDT:
;门                          目标选择子,             偏移, DCount, 属性
%rep 32
        Gate    SelectorCode32, SpuriousHandler,        0, DA_386IGate
%endrep
.020h:  Gate    SelectorCode32,     ClockHandler,       0, DA_386IGate
%rep 95
        Gate    SelectorCode32, SpuriousHandler,        0, DA_386IGate
%endrep
.080h:  Gate    SelectorCode32,  UserIntHandler,        0, DA_386IGate

IdtLen      equ $ - LABEL_IDT
IdtPtr      dw  IdtLen - 1  ; 段界限
            dd  0           ; 基地址
; END of [SECTION .idt]
```

可以看到和 pmtest9a 相比，新增了 20h 号 ClockHandler 和 80h 号 UserIntHandler，后者功能为向特定位置打印字符“I”，前者功能为在响应时钟中断的时候使后者打印的字符对应的 ACSII 值自增，来起到改变字符显示时钟变化的功能。

```
_ClockHandler:
ClockHandler    equ _ClockHandler - $$
    inc byte [gs:((80 * 0 + 70) * 2)]   ; 屏幕第 0 行, 第 70 列。
    mov al, 20h
    out 20h, al             ; 发送 EOI
    iretd

_UserIntHandler:
UserIntHandler  equ _UserIntHandler - $$
    mov ah, 0Ch             ; 0000: 黑底    1100: 红字
    mov al, 'I'
    mov [gs:((80 * 0 + 70) * 2)], ax    ; 屏幕第 0 行, 第 70 列。
    iretd
```

- IDTR 和 IMG 的保存、实模式 8259A 的设置、使系统停在保护模式：

```
    ; 保存 IDTR
    sidt    [_SavedIDTR]

    ; 保存中断屏蔽寄存器(IMREG)值
    in  al, 21h
    mov [_SavedIMREG], al
```

在加载 GDTR 和 IDTR 之前先将 IDTR 和 IMG 保存。

Listing 2: SetRealmode8259A

```
; SetRealmode8259A ------------------------------------------
SetRealmode8259A:
    mov ax, SelectorData
    mov fs, ax

    mov al, 017h
    out 020h, al    ; 主8259, ICW1.
    call    io_delay

    mov al, 008h    ; IRQ0 对应中断向量 0x8
    out 021h, al    ; 主8259, ICW2.
    call    io_delay

    mov al, 001h
    out 021h, al    ; 主8259, ICW4.
    call    io_delay

    mov al, [fs:SavedIMREG] ;   恢复中断屏蔽寄存器(IMREG)的原值
    out 021h, al        ;
    call    io_delay

    ret
; SetRealmode8259A ------------------------------------------
```

设置好实模式下的中断处理机制。

```
    call    Init8259A

    int 080h
    sti
    jmp $
```

加入了 sti（虽然在 pmtest9 中一开始就没有关中断，但是在关掉中断的 pmtest9a 中可以通过注释掉 cli 或者新增 sti 来解决3.1.2中的错误）和 jmp $ 使系统停在保护模式以便执行完 80h 号中断之后能看到 20h 号时钟中断的执行。

3.2.2 调试程序

直接运行原始代码，可以看到执行结果如下：

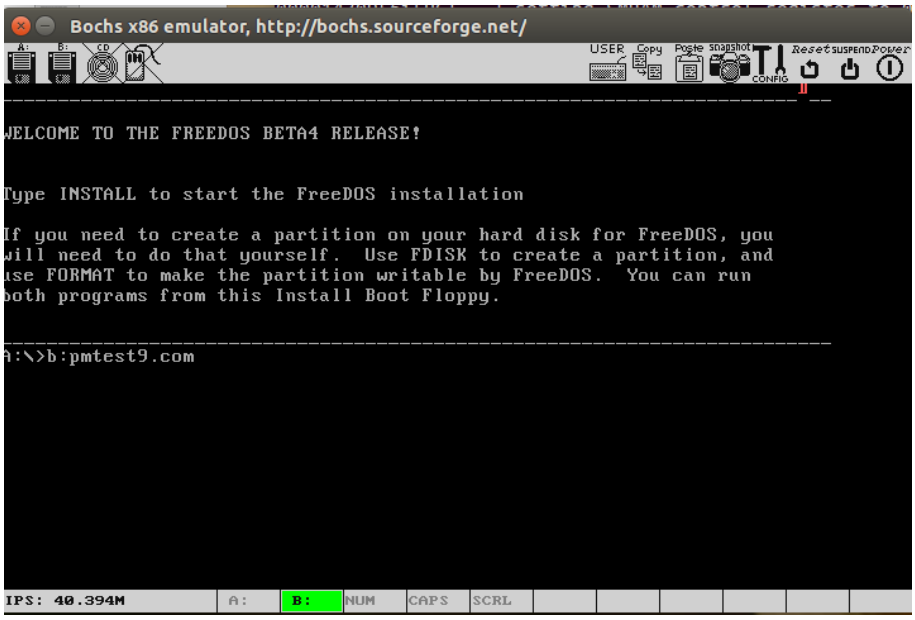

图 4: pmtest9 执行结果

不便于展示动图。事实上，第 0 行第 75 列处的字符是不停变化的，说明通过时钟中断 int 20H，实现了值的自增，进而以字符的方式显示在屏幕上。同时，注意到我们看到的字符的变化并不是连续的，猜测是屏幕刷新频率低于时钟中断触发频率导致的。

为了更好地理解程序，在 ClockHandler 和 UserIntHandler 中分别增加断点，重新编译、装载程序，并运行程序。停在第一个断点处时，执行结果如下：

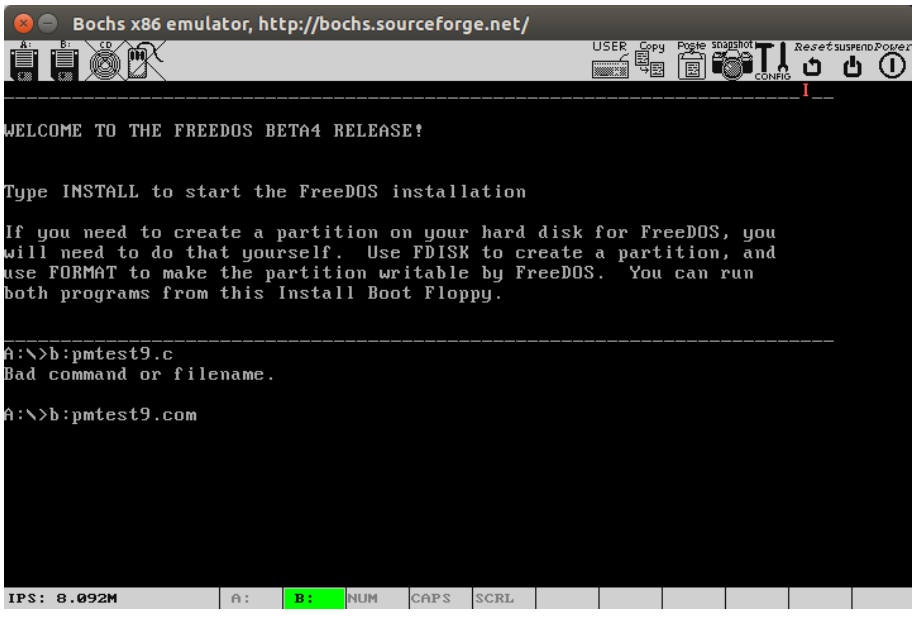

图 5: 第一个断点——停在 UserIntHandler

此时程序停在 UserIntHandler，向低 0 行第 70 列写入一个 I，屏幕上显示出对应的内容。继续执行程序，停在下一个断点。执行结果如下：

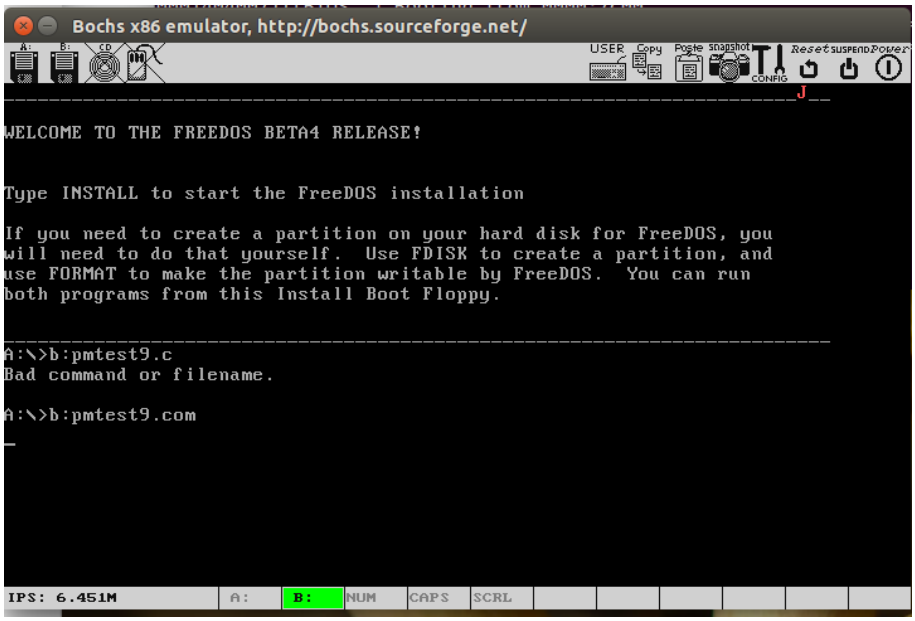

图 6: 第二个断点——停在 ClockHandler

事实上，程序一直在执行 jmp $，直到捕捉到下一个时钟中断，此时调用 int 20H，完成自增。可以看到屏幕上指定位置的字符从 I 变成了 J。继续调试，字符变成 K,L,M 等等。此处不再重复展示。

调试结果与我们对代码的分析是一致的。

3.3 实现一个自定义的中断向量

我们一共实现了四个自定义中断向量，可以参见代码中的 int.asm int2.asm int3.asm int4.asm 这四个文件。其中前两个是保护模式下的中断，后两个是实模式下的中断。

这四个中断的基本功能如下：

• int.asm：计时器中断。效果是屏幕上出现一个在 A 到 Z 间循环跳动的红色字符。一段时间后，循环结束，屏蔽中断，该字符会停止跳动，随后程序退出，返回 DOS。

• int2.asm：在 int.asm 的基础上，加上了键盘中断。因此除了不断跳动的字符以外，按下键盘时，屏幕上会出现通码；松开键盘时，屏幕上会出现断码。

• int3.asm：修改实模式中断向量表，在实模式下实现了上面的键盘中断。

• int4.asm：区别于 int3.asm，同样修改了中断向量表，但在自定义功能完成后，将控制权交还给原本的处理程序，由原本的处理程序继续完成中断。效果是，程序退出后，在 freedos 下打字，除了像往常一样能正常输入外，还能在右上角看到一个会动的红色字符。

3.3.1 保护模式下自定义中断

我们将中断处理程序同一放置在 handler.asm 这个文件中，只要在主程序中引用这个文件，即可正确设置中断处理程序。

首先，我们分析一下时钟中断的实现方式。原有时钟中断会不停的增加字符的值，造成字符值超过 ASCII 中字母的范围。所以我们为原有的时钟中断加入了循环功能，当字符增加到 Z 时，将其重新设置为 A。具体实现如 Listing 3所示，使用 cmp byte 比较字符，使用 jz 将字符重新设置为'A' 或者增加字符的值。

Listing 3: handler.asm

```
_ClockHandler:
    push eax
    cmp byte [gs:CharPos], Z
    je .2
    inc byte [gs:CharPos]
    jmp .exit
.2:
    mov byte [gs:CharPos], A
.exit:
    mov al, 20h
    out 20h, al
    pop eax
    iretd
    ClockHandler equ _ClockHandler - $$
```

除此之外，我们还添加了一个自定义键盘中断处理程序。这个键盘中断处理程序会读取触发键盘中断的键值，并展示其扫描码。具体实现如 Listing 4所示，键盘中断位于中断描述符表的第 21h 号。键盘中断发生时，说明键盘的数据已经就绪。我们只需要读取键盘的数据端口（60h）即可得到键盘输入的数据。DispAL 用于显示 al 中读取到的扫描码。ShowStr 定义在 utils.asm 中，可以显示一个给定字符，str32 定义为空格，故 ShowStr str32 用于显示空格。

Listing 4: handler.asm

```
_KeyBoardHandler:
    cli
    push eax

    in al, 60h
    call DispAL
    ShowStr str32

.exit:
    mov al, 20h
    out 20h, al
    pop eax
    iretd
KeyBoardHandler equ _KeyBoardHandler - $$
```

int2.asm 的执行结果如图所示：

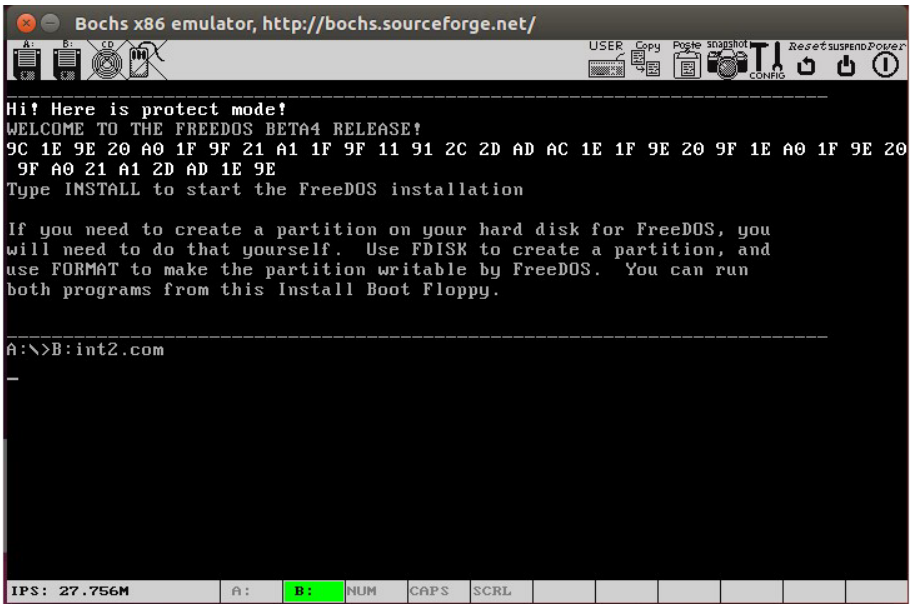

图 7: int2 程序执行结果

3.3.2 实模式下自定义中断

实模式下的自定义中断可以直接通过写入中断向量表实现。实模式下的键盘中断位于中断向量表的第 9 号。故我们编写了新的 9 号中断，并把中断向量表指向这个新的中断。新的中断处理程序如 Listing 5所示，由于需要在程序退出后，中断向量表仍然能够调用中断处理程序，所以我们需要将中断处理程序拷贝到内存中未被占用的位置，这里选择了 0x200 这个位置。中断处理程序的实现与保护模式下大致相同，可以循环显示从 A-Z 的红色字符。

Listing 5: int3.asm

```
1     _int9:
2     push eax
3     push gs
4     mov ax, 0b800h
5     mov gs, ax
6     in al, 60h ; 读取缓冲区
7     cmp byte [gs:((80 * 0 + 70) * 2)], Z
8     je .2
9     inc byte [gs:((80 * 0 + 70) * 2)]
10    jmp .exit
11 .2:
12    mov byte [gs:((80 * 0 + 70) * 2)], A
13 .exit:
14    mov al, 20h
15    out 20h, al ; 输出 EOI
16    pop gs
17    pop eax
18    iret
```

在此基础上，我们在 int4.asm 中添加了保存原有 9 号中断，以及重定向到原有 9

号中断的功能，使得在发生键盘中断以后，除了能够实现输入功能，还能够在右上角显示一个变化的红色字符。

```
_int9:
    push ax
    push gs
    ; 运行自定义的中断程序
    mov ax, 0b800h
    mov gs, ax
    mov byte [gs:(CharPos + 1)], 0Ch
    inc byte [gs:CharPos]
    jmp .exit
.exit:
    pop gs
    pop ax
    ; 由原本的中断程序接管
    ; jmp dword 60h:0d0d6h
    push word [cs:202h]
    push word [cs:200h]
    retf
```

int4.asm 的执行结果如图 8和图 9所示：

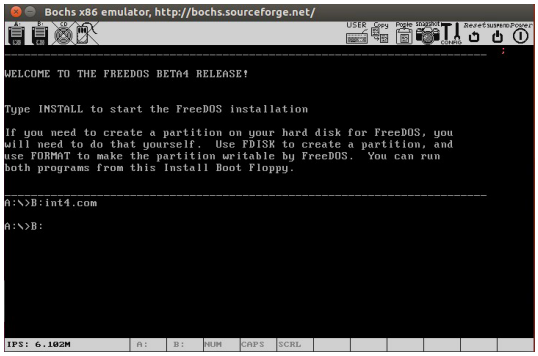

图 8: int4 运行结果 1

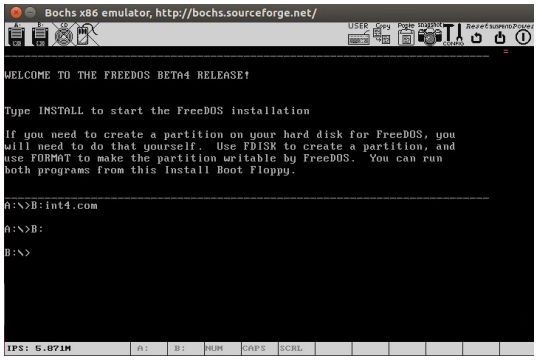

图 9: int4 运行结果 2

4 实验结果总结

4.1 中断和异常的概念

早期中断和异常都被称为中断，后来按照产生的原因进行了区分。异常和中断都是软件或者硬件发生了某种情形而通知处理器的行为。

4.1.1 异常

异常也叫内部中断，它不需要硬件支持。异常的引入表示 CPU 执行指令时本身出现的问题。异常可分为以下三种类型：

- Fault：一种可以被更正的异常。而且一旦被更正，程序可以不失连续性地继续执

行。当一个 fault 发生，处理器会把产生 fault 指令之前的状态保存起来，异常处理程序的返回地址将会是产生 fault 的指令，而不是其后的那条指令。

• Trap：是一种在发生 trap 的指令执行之后立即被报告的异常，它也允许程序或任务不失连续性地继续执行。异常处理程序的返回地址将会是产生 trap 的指令之后的那条指令。

• Abort：是一种不总是报告准确异常发生位置的异常，它不允许程序或者任务继续执行，而是用来报告严重错误的。

注意，异常是同步的，这是指异常发生的时候，CPU 立即处理本次异常，直到异常处理结束之后才能继续进行接下来的任务。

4.1.2 中断

这里的中断指的是外部中断。中断的引入是为了支持 CPU 和设备之间的并行操作。中断可以分为可屏蔽中断和不可屏蔽中断。

• 可屏蔽中断：指通过可屏蔽中断请求线 INTR 向 CPU 发出中断请求，CPU 可以通过在中断控制器中设置响应的屏蔽字来屏蔽他或不屏蔽他，被屏蔽的中断请求将不被送至 CPU。

• 不可屏蔽中断：指通过专门的不可屏蔽请求线 NMI 向 CPU 发出的中断请求，通常是非常紧急的硬件故障。

注意，外部中断是异步的，意思是所有中断来的信号都是记录在中断寄存器中的，当 CPU 执行完一道指令之后，如果是开中断状态，则会检查中断寄存器中有没有中断，如果有中断，就会选择一个中断优先级比较高的中断先处理，等到处理完中断再继续执行；如果是关中断，则不会检查，而直接执行下一条指令。

4.2 处理机制

4.2.1 实模式下的中断处理

实模式下，中断转移方法与 8086 相同，即通过中断向量号直接去中断向量表种找到中断处理程序入口，然后跳转到指定位置执行中断处理程序。示意图如下：

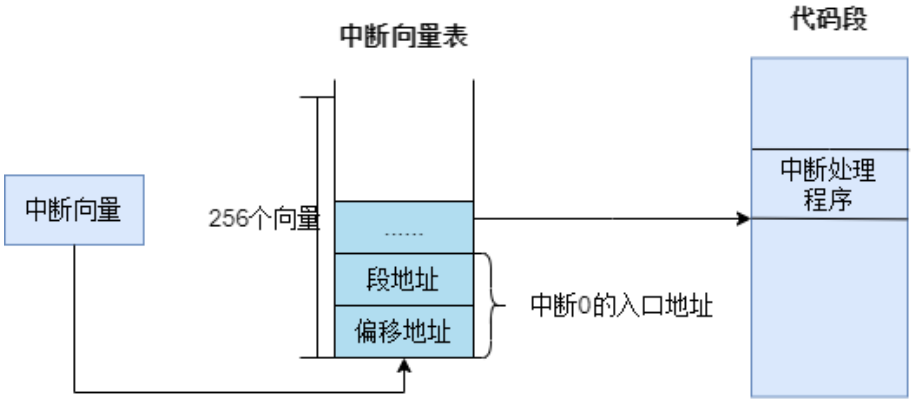

图 10: 实模式中断处理

4.2.2 保护模式下的中断处理

不同于实模式，在保护模式下，中断向量表被 IDT 代替。IDT 的作用是将每一个中断向量和一个描述符对应起来。IDT 的描述符有以下三类：中断门描述符、陷阱门描述符、任务门描述符。

- 中断门和陷阱门中断门和陷阱门的结构如下图所示：

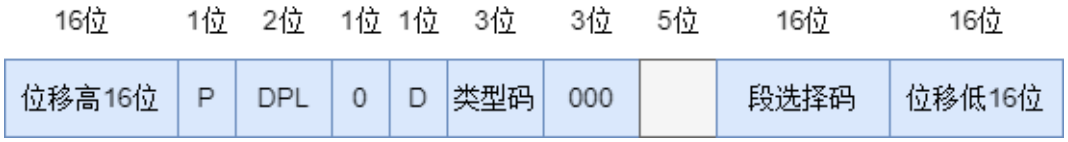

图 11: 中断门、陷阱门结构

其中灰色部分表示保留，不使用。其中段选择码和偏移用来定位中断处理程序，其余标志该描述符的属性。具体的处理机制如图12。

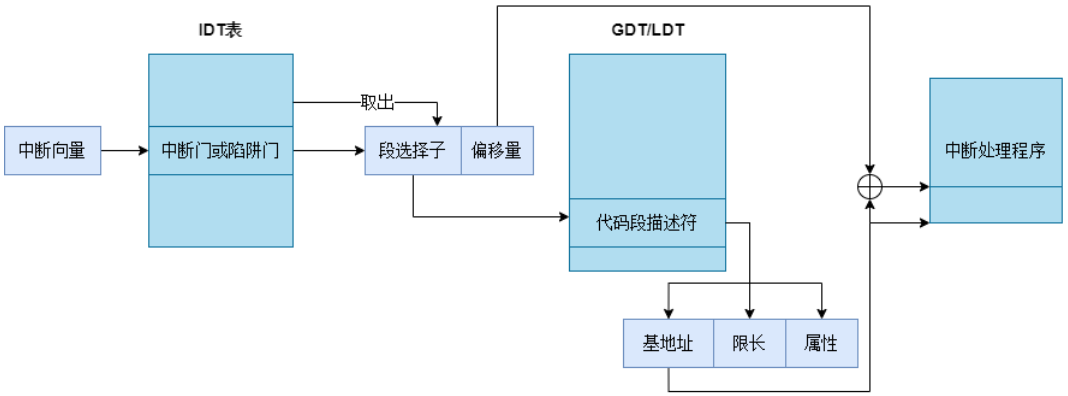

图 12: 中断门和陷阱门

注意，中断门和陷阱门的区别是对中断允许标志 IF 位的影响。中断门向量引起中断时会复位 IF，此时其他中断干扰会被屏蔽，最终通过 iret 从堆栈上恢复出 IF 的原值。

• 任务门

任务门的结构如下图所示：

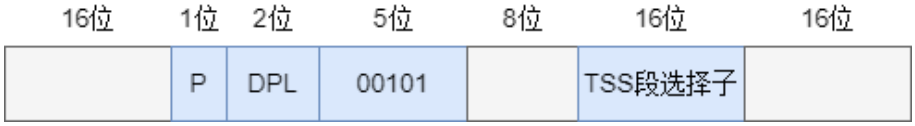

图 13: 任务门结构

同样，灰色部分表示空闲不使用。任务门不需要提供段内偏移，因为任务门不指向某一个子程序的入口，TSS 本身是作为一个段来对待的。任务门的处理机制如下图。

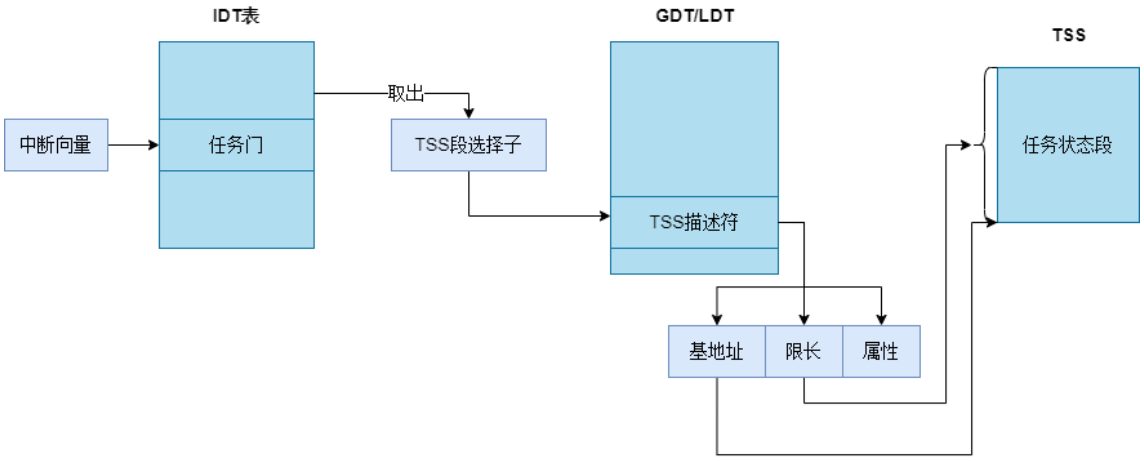

图 14: 任务门

根据 TSS 段的信息转入对应的中断或异常处理程序。

4.3 8259A 的工作原理

4.3.1 8259A 的内部结构

8259A 的内部结构如图15所示。

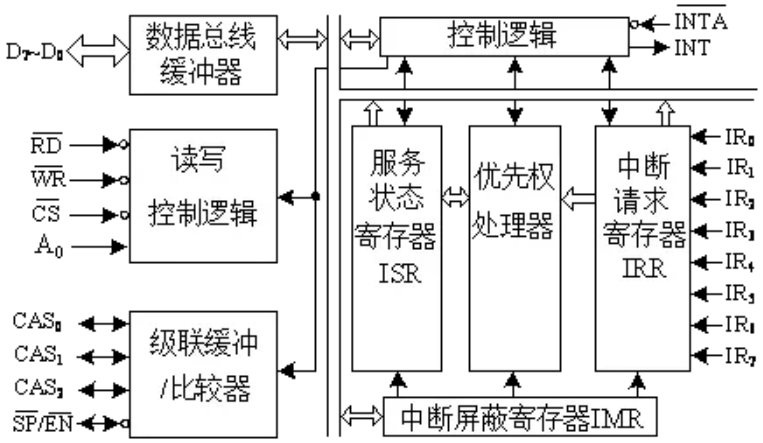

图 15: 8259A 内部结构

关于各个组成部分的解释如下：

• 中断请求寄存器 IRR(Interrupt Request Register)

外部 8 级中断请求信号从 IR7 IR0 脚上引入，有请求时相应位置为 1。多个中断请求可同时进入。中断请求信号可为高电平或上升沿触发，通过编程定义。

• 中断屏蔽寄存器 IMR(Interrupt Mask Register)

存放中断屏蔽信息，每 1 位与 1 个 IR 位对应，置 1 表示禁止对应中断请求进入系统。用来有选择地禁止某些设备请求中断。

• 服务状态寄存器 ISR(Interrupt Service Register)

保存正处理的中断请求。任一中断被响应而执行其服务程序时，相应位置 1，直到处理结束。若有多重中断的情况，会有多个位置为 1。

• 优先级处理器 PR(Priority Resolver)

判别请求寄存器 IRR 里中断的优先级，把优先级最高的中断请求选进服务寄存器 ISR 中。多重中断出现时，PR 判定新出现的中断能否去打断正在处理的中断，优先服务更高的中断级别。

• 控制电路

它包含一组初始化命令字寄存器 $ICW_1 \sim ICW4$ 和一组操作命令字寄存器 $OCW_1 \sim OCW_3$，管理 8259A 的全部工作。控制电路根据 IRR 设置和 PR 判定，决定控制信号，然后从 INT 脚向 CPU 发中断请求信号，接收 CPU 或总线控制器 8288 送来的中断响应信号 $\overline{INTA}$。中断响应时 ISR 相应位置 1，并发送中断类型号 n，经数据总线缓冲器送到 D7~D0；中断服务程序结束时，按编程规定方式结束中断。

• 数据总线缓冲器

是 8259A 与 CPU 的接口，CPU 经它向 8259A 写控制字，接收 8259A 送出的中断类型号，还可从中读出状态字（中断请求、屏蔽、服务寄存器的状态）和中断查询字。

• 读写控制逻辑

接收 CPU 的 $\overline{RD}$、$\overline{WR}$、地址、片选。一片 8259A 只占两个 I/O 地址，XT 机中 A0 接地址 A0，口地址为 20H、21H。当与 8086 连时，A0 脚接地址 A1，A0 的 0/1 选偶/奇

地址口。执行 OUT 指令时，$\overline{WR}$ 信号与 A0 配合，将控制字写入 ICW 和 OCW 寄存器；执行 IN 指令时，$\overline{RD}$ 信号与 A0 配合，将内部寄存器的内容经 D7 D0 送给 CPU。

• 级联缓冲/比较器

一片 8259A 最多引入 8 级中断，超过 8 级要用多片 8259A 构成主从关系，级联使用。当需要用两片 8259A：从片输出 INT 接主片 IR_i，主从片的 3 条级联信号线 CAS2∼CAS0 并接。

4.3.2 8259A 的工作方式

写入初始化命令字 ICW 和控制命令字 OCW，可以对 8259A 设置不同的工作方式。

• 设置优先级方式

优先级方式分为以下几种：

· 全嵌套方式：最基本方式，初始化后自动进入。

从各 IRi 脚引入的中断请求具有固定优先级，$IR_0 \rightarrow IR7$ 依次降低，IR_0 最高。8259A 初始化后自动进入此方式。处理过程中，高级中断打断低级中断，禁止低级或同级中断进入。

· 特殊全嵌方式

同全嵌套方式, 但允许同级中断进入（在主片看来，从片的 8 级中断为同级中断）。

· 优先级自动循环方式

各中断请求优先级相同，IR_i 服务完后成为最低级，IR_{i+1} 成最高级。初始优先级从高到低为 $IR_0 \rightarrow IR_7$。

· 优先级特殊循环方式

也称为设置最低优先级方式，与优先级自动循环方式类似，只是最低优先级由程序设置，并非 IR_7 最低。IR_i 设为最低后，$IR_i + 1$ 便是最高。

• 中断屏蔽方式

开中断情况下，可将中断屏蔽寄存器 IMR 的相应位置 1，来屏蔽某一级或某几级中断。代码实现上，可用 CLI 指令关中断，禁止可屏蔽中断进入。有两种屏蔽方式：普通屏蔽方式和特殊屏蔽方式。

• 结束中断方式

中断响应后，ISR 的相应位 IS_n 置 1，中断结束后应将 IS_n 清 0，表示结束中断。有 2 种结束中断方式：自动和非自动，后者又分普通结束和特殊结束（EOI 和 SEOI）。这里不再展开详述。

• 中断查询方式

使用一条 IN 指令读取中断查询字，就可查到 8259A 是否有中断请求以及哪个优先级最高。

4.3.3 8259A 的命令字设置

为使 8259A 按预定方式工作，必须对它编程，由 CPU 向其控制寄存器发各种控制命令。控制命令分为两类——初始化命令字和操作命令字。

- 初始化命令字

初始化命令字的写入流程如下图所示。

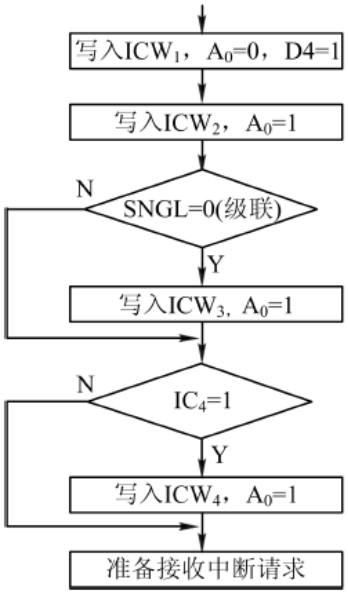

图 16: 初始化命令字写入流程

初始化命令字必须从 ICW_1 开始顺序写入规定端口，级联时要写入 ICW_3，无级联时无需写入。

· ICW_1——初始化字

ICW_1 的结构如下图：

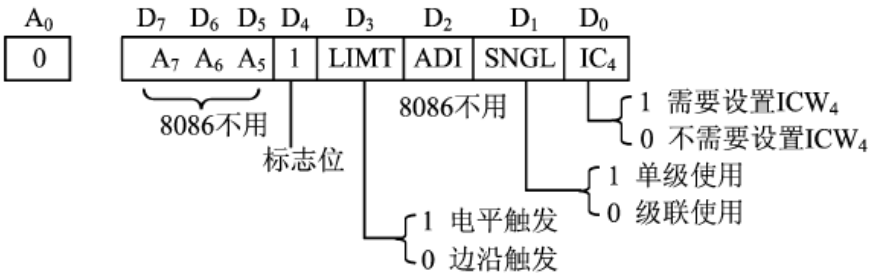

图 17: ICW1 结构

A_0=0，ICW_1 写入偶地址口；D_1 表示 SNGL 属性，标志是否使用级联。级联时 SNGL=0, 要写入 ICW_3。

· ICW_2——中断向量码

ICW_2 的结构如下图:

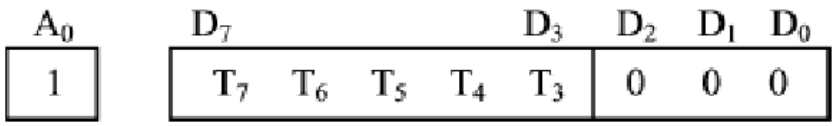

图 18: ICW2 结构

A_0=1，ICW_2 写入奇地址口；T7∼T3 位用于确定中断类型码 n 的高 5 位，低 3 位 D2∼D0 则由 8259A 根据从 IRi 上引入中断的引脚序号自动填入，从 IR0∼IR7 的序号依次为 000 111，其初值可以置为 0。ICW_2 的高 5 位内容是可以任选的，一旦高 5 位确定，一块芯片的 8 个中断请求信号 IR0∼IR7 的中断类型号也就确定了。

· ICW_3——级联控制字

ICW_3 只在级联时使用。ICW_3 的格式和 8259A 主从结构连接示意图如下：

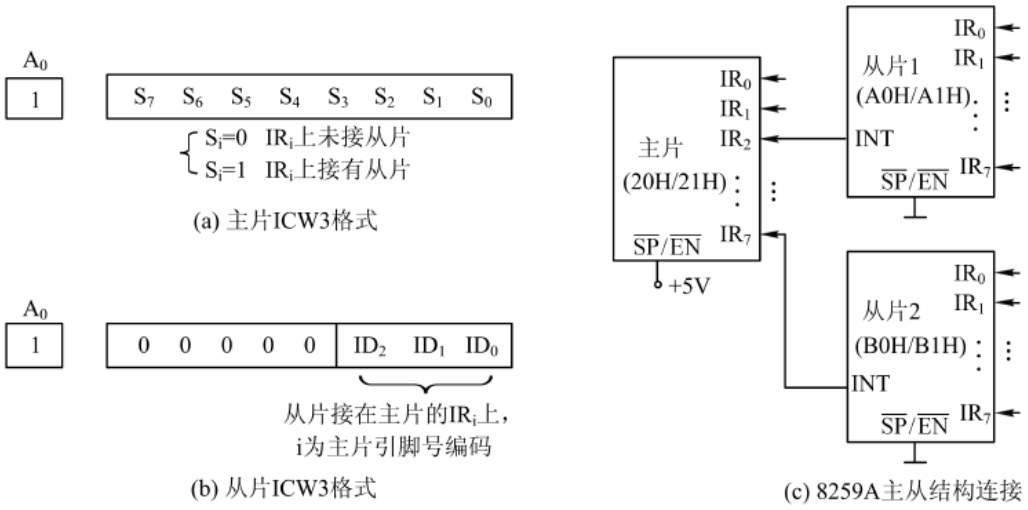

图 19: ICW3 结构以及主从结构

主片 ICW_3 格式：S_i=0，IR_i 上未接从片，S_i=1 接有从片。从片 ICW_3 格式。低 3 位指明从片接主片哪个引脚，$ID_2 \sim ID0 = 000 \sim 111$ 表示 $IR_0 \sim IR_7$ 引脚。

· ICW_4——中断结束方式字

8086 系统 ICW_4 必须设置，写入奇地址口。ICW_4 的结构如下：

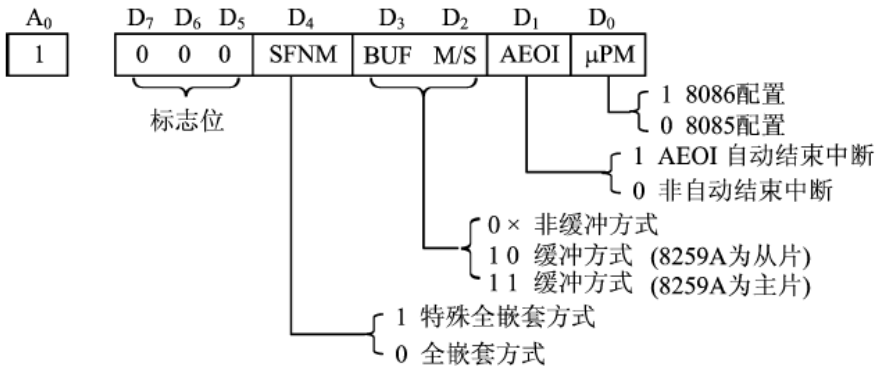

图 20: ICW4 结构

• 操作命令字

用来改变 8259A 的中断控制方式，屏蔽中断源，以及读出 8259A 的工作状态（IRR,ISR,IMR），初始化完成后任意状态皆可写入，顺序也没有严格要求，但是对端口地址有规定：OCW_1 奇地址端口（A0=1），OCW_2,OCW_3 必须为偶地址端口。

· OCW_1——中断屏蔽字

直接对中断屏蔽寄存器 IMR 的各位进行置 1 或清 0。当 M_i 位置 1，相应 IRi 的中断请求将被屏蔽，清 0 则允许中断。其结构如下：

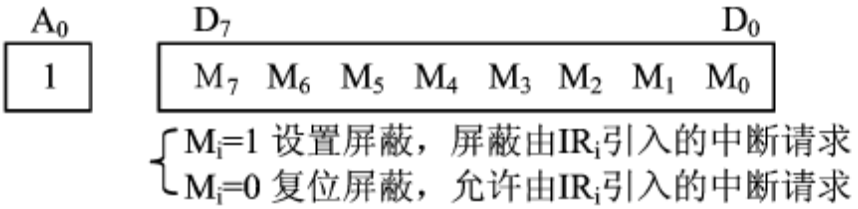

图 21: OCW1 结构

· OCW_2——优先级循环和中断结束

OCW_2 是设置优先级循环方式和中断结束方式的命令字。其结构如下：

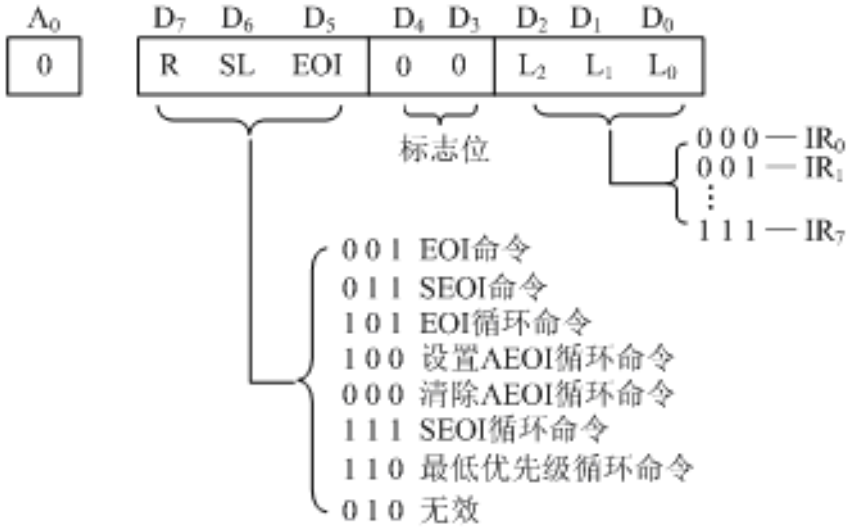

图 22: OCW2 结构

· OCW_3——屏蔽方式和状态读出控制字

OCW_3 的结构如下图：

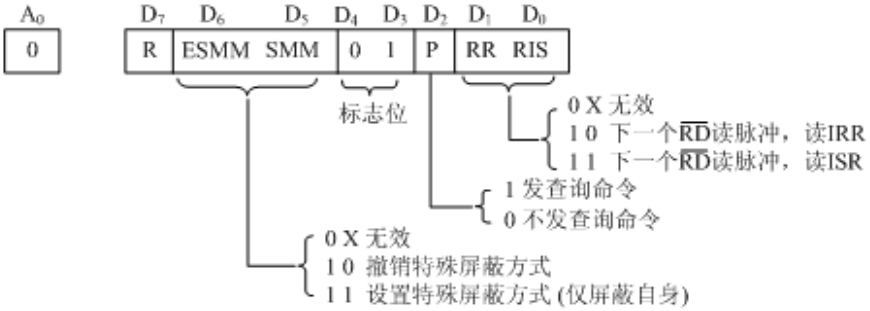

图 23: OCW3 结构

4.3.4 8259A 工作原理概述

先通过 $ICW_1 \sim ICW_4$ 来初始化 8259A，使芯片处于一个规定的基本工作方式。初始化完成后，开中断，一个外部中断请求信号通过中断请求线 IRQ，传输到 IMR（中断屏蔽寄存器），IMR 根据所设定的中断屏蔽字（OCW_1），决定是将其丢弃还是接受。

如果可以接受，则 8259A 将 IRR（中断请求暂存寄存器）中代表此 IRQ 的位置位，以表示此 IRQ 有中断请求信号，并同时向 CPU 的 $\overline{INTR}$ 管脚发送一个信号，但 CPU 这时可能正在执行一条指令，因此 CPU 不会立即响应，而当这 CPU 正忙着执行某条指令时，还有可能有其余的 IRQ 线送来中断请求，这些请求都会接受 IMR 的挑选，如果没有被屏蔽，那么这些请求也会被放到 IRR 中，也即 IRR 中代表它们的 IRQ 的相应位会被置 1。

当 CPU 执行完一条指令时后，会检查一下 $\overline{INTR}$ 管脚是否有信号，如果发现有信号，就会转到中断服务，此时，CPU 会立即向 8259A 芯片的 INTA（中断应答）管脚

发送一个信号。当芯片收到此信号后，判优部件开始工作，它在 IRR 中，挑选优先级最高的中断，将中断请求送到 ISR（中断服务寄存器），也即将 ISR 中代表此 IRQ 的位置位，并将 IRR 中相应位置零，表明此中断正在接受 CPU 的处理。同时，将它的编号写入中断向量寄存器 IVR 的低三位。这时，CPU 还会送来第二个 INTA 信号，当收到此信号后，芯片将 IVR 中的内容，也就是此中断的中断号送上通向 CPU 的数据线。

4.4 中断号的处理向量初始化

本节从 linux 源码角度阐述中断号的处理向量初始化。

在 linux 系统中，中断一共有 256 个，0 19 主要用于异常与陷阱，20 31 是 intel 保留，未使用。32 255 作为外部中断进行使用。特别的，0x80 中断用于系统调用。机器上电时，BIOS 会初始化一个中断向量表，当交接给 linux 内核后，内核会自己新建立一个中断向量表，之后完全使用自己的中断向量表，舍弃 BIOS 的中断向量表。

中断的初始化大体上分为两个部分，第一个部分为汇编代码的中断向量表的初次初始化，第二部分为 C 语言代码，又分为异常与陷阱的初始化和中断的初始化。

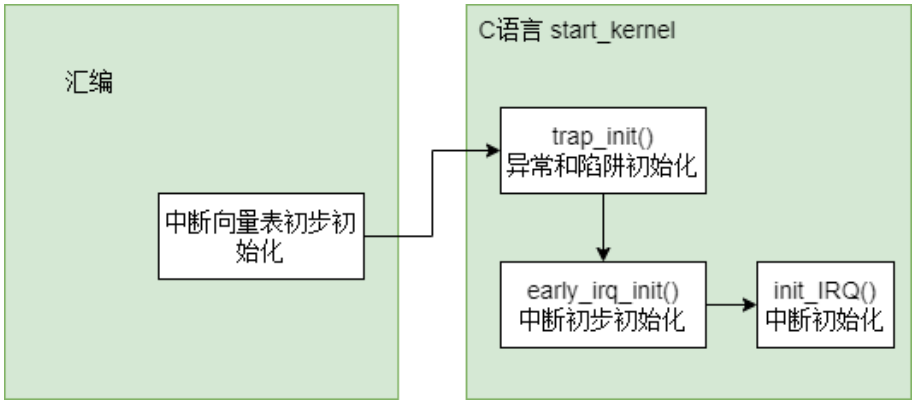

图 24: 中断初始化

在汇编的中断向量表初始化过中，主要对整个中断向量表进行了初始化，其主要工作是:· 所有的门描述符的权限位为 0;· 所有的门描述符的段选择符为 ___KERNEL_CS;· 0∼31 的门描述符的中断处理程序为 early_idt_handlers[i](0 <= i <= 31);· 其他的门描述符的中断处理程序为 ignore_int。

trap_init() 所做的异常与陷阱初始化，就是修改中断向量表的前 19 项 (异常和中断)，主要修改他们的中断处理函数入口和权限位，特殊的如任务门还会设置它们的段选择符。在 trap_init() 中就已经把所有的异常和陷阱都初始化完成了，并会把新的中断向量表地址放入 IDTR 寄存器，开始使用新的中断向量表。

在 early_irq_init() 中，主要工作是初始化整个中断描述符表，将数组中的每个中断描述符中的必要变量进行初始化。

最后在 init_IRQ() 中，主要工作是初始化中断向量表中的所有中断门描述符，对于一般的中断，内核将它们的中断处理函数入口设置为 interrupt[i]，而一些特殊的中断会在

apic_intr_init() 中进行设置。之后，init_IRQ() 会初始化内部和外部的中断控制器，最后将一般的中断使用的中断控制器设置为 8259A，中断处理函数为 handle_level_irq(电平触发)。

4.5 建立 IDT

参考3.2节中关于 IDT 的内容，IDT 是按地址顺序从 0h 号开始顺序设置描述符的，也就是说假设你要设置 20h 号中断，你只需要设置好第 20 个描述符，将其指向 32 位代码段中的中断处理程序。

4.6 控制时钟中断

可以通过修改 8259A 的状态来控制时钟中断。

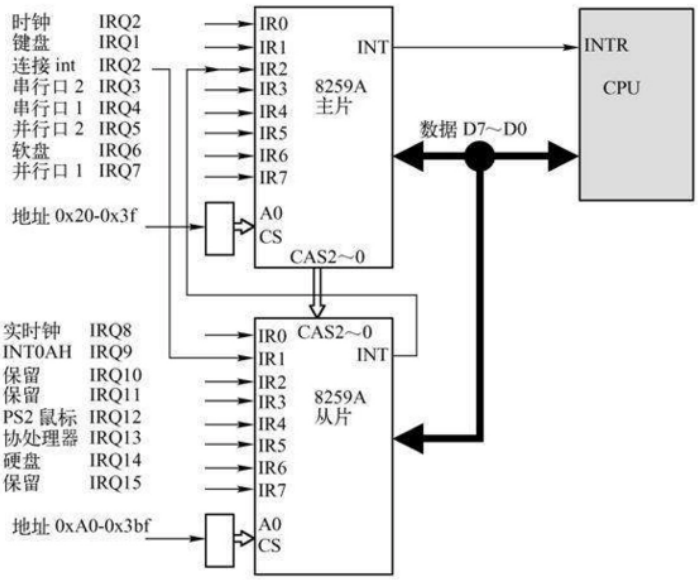

图 25: 主从 8259A

从主从 8259A 结构中可以看到，时钟中断处在 8259A 主片的 IR0 端口。通过 OCW_1 可以设置主片的 IR_0 是否屏蔽，从而决定系统对时钟中断是否响应。

时钟中断的中断处理程序位于 IDT 的第 20h 号，修改 IDT 中对应位置即可将时钟中断的处理定位到自定义程序。

Q：为什么时钟中断时候，没有看到 int 的指令？

int 指令触发的是软中断。而时钟中断是由定时器触发的中断，即硬件中断。时钟中断是由 8259A 直接发送给 CPU 进行处理的，不需要 int 指令来触发。

4.7 IOPL 的作用与基本机理

4.7.1 IOPL 的作用

IOPL 是 I/O 保护机制的关键之一，它能够控制 I/O 的权限。IOPL 保存在寄存器 eflags 的 12、13 位。

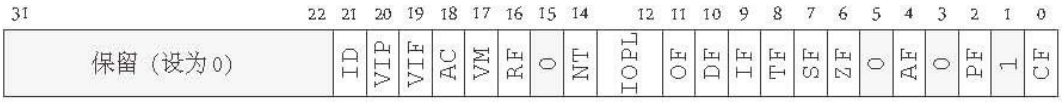

图 26: eflags 结构

指令 in,ins,out,outs,cli,sti 必须在 CPL≤IOPL 的条件下才能执行。上述指令是 I/O 敏感指令，如果低特权级的指令试图访问这些指令会导致常规保护错误。

4.7.2 IOPL 的基本机理

处理器不限制 0 特权级程序的 I/O 访问，它总是允许的。但是，可以限制低特权级程序的 I/O 访问权限。这是很重要的，操作系统的功能之一是设备管理，它可能不希望应用程序拥有私自访问外设的能力。

可以改变 IOPL 的指令只有 popf 和 iretd，但只有运行在 ring0 的程序才能将其改变。运行在低特权级下的程序无法改变 IOPL，不过，如果试图那样做的话并不会产生任何异常，只是 IOPL 不会改变，仍然保持原样。

另一个与 I/O 操作特权级有关的概念是 I/O 位图。I/O 位图基址是一个以 TSS 的地址为基址的偏移，指向的便是 I/O 许可位图。之所以叫做位图，是因为它的每一位表示一个字节的端口地址是否可用。如果某一位为 0，则表示此位对应的端口号可用，为 1 则不可用。由于每一个任务都可以有单独的 TSS，所以每一个任务可以有它单独的 I/O 许可位图。

IO 位图与 IOPL 是或的关系，即只要满足一个条件即可。

指令	功能	条件
cli	清除 IF 位	CPL<=IOPL
sti	设置 IF 位	CPL<=IOPL
in	从 IO 读数据	CPL<=IOPL 或 IO 位图许可
ins	从 IO 读字符串	CPL<=IOPL 或 IO 位图许可
out	向 IO 写数据	CPL<=IOPL 或 IO 位图许可
outs	向 IO 写字符串	CPL<=IOPL 或 IO 位图许可

5 各人实验贡献与体会

本次实验主要探究了异常与中断的实现。在汇编语言课程中，我们就已经对实模式下的中断有了一定的认识，本次重点关注了保护模式下的异常中断处理机制。通过对 pmtest9a.asm 的调试，熟悉了 IDT 的建立以及保护模式下中断调用；通过对 pmtest9.asm 的调试，理解了时钟中断的触发机理。最后，基于对中断原理的理解，我们编写了实模式和保护模式下自定义中断（键盘中断和计时器中断）。在实验过程中，组员之间积极讨论，对原理有了更深刻的理解。

实验分工如下：李心杨负责撰写自定义中断程序的说明，协助其他同学解决代码中的问题；王宇骥对中断原理进行总结，完善报告细节；林锟扬负责编写自定义中断程序；郑炳捷负责调试 pmtest9a.asm 并完成对应部分实验报告。

参考资料汇总

1. 配套教材《Orange's 一个操作系统的实现》。

2. 微机系统课程 PPT 《8.2 8259A 的原理》。

3. 任务门、中断门、陷阱门和调用门 - silenccfly -博客园。

4. 操作系统——认识认识保护模式（三）中断。

5. 汇编学习笔记 (21) - IO 保护。

教师评语		
姓名	**学号**	**分数**
李心杨	2020302181022	
王宇骥	2020302181008	
林锟扬	2020302181032	
郑炳捷	2020302181024	
教师签名： 年 月 日		